MODERN C++ ALGORITHMS

A GRADUATE-LEVEL COMPANION

PREPARED BY
Ayman Alheraki

DRAFT

Modern C++ Algorithms: A Graduate-Level Companion

Prepared by Ayman Alheraki

simplifycpp.org

August 2025

Contents

Contents **2**

Author's Preface **26**

I Foundations (C++-centric) 28

1 Preface & How to Use This Book **30**

1.1 Target Audience and Prerequisites (C++17/20/23) 30

1.1.1 Prerequisites 32

1.1.2 Positioning of This Book 33

1.2 Coding Standards Used in Examples (formatter, naming, header structure) 34

1.2.1 Code Formatting and Style 34

1.2.2 Naming Conventions 35

1.2.3 Header and Source Structure 38

1.2.4 Modern C++ Practices 41

1.3 Build & Run: CMake Minimal Template, Compiler Flags, Sanitizers, and Test Runner Setup 42

1.3.1 Minimal CMake Template 42

1.3.2 Recommended Compiler Flags 43

1.3.3 Sanitizers 44
1.3.4 Test Runner Setup 45
1.3.5 Recommended Workflow 46

2 Algorithmic Thinking with C++ 47

2.1 What is an Algorithm? C++ Examples as First-Class Citizens 47
2.1.1 Defining Algorithms 47
2.1.2 Algorithms as First-Class Citizens in C++ 48
2.1.3 Algorithmic Thinking in C++ 50
2.1.4 Summary 51
2.2 Complexity Notation (Big-O / Θ / Ω) Illustrated with C++ Microbenchmarks 53
2.2.1 Introduction to Complexity Notation 53
2.2.2 Microbenchmarks in C++ 54
2.2.3 Visualizing Complexity 56
2.2.4 Space Complexity 56
2.2.5 Modern C++ Techniques for Complexity Analysis 57
2.2.6 Key Takeaways 58
2.3 Practical Measurement: `chrono`, `std::execution`, CPU Cycles, and Pitfalls 59
2.3.1 Measuring Time with `<chrono>` 59
2.3.2 Parallel Execution with `std::execution` 60
2.3.3 Measuring CPU Cycles 61
2.3.4 Common Measurement Pitfalls 62
2.3.5 Recommended Workflow for Reliable Benchmarks 63
2.3.6 Summary 64

3 Essential C++ Tools for Algorithm Developers **65**

3.1 The Standard Library Overview Relevant to Algorithms (Containers, Iterators, Algorithms Header) . . . 65

3.1.1 Containers . . . 66

3.1.2 Iterators . . . 68

3.1.3 The `<algorithm>` Header . . . 69

3.1.4 Best Practices for Algorithm Developers . . . 70

3.2 Modern C++ Features That Change Algorithm Design: `ranges`, `concepts`, `span`, `string_view` . . . 72

3.2.1 Ranges (`std::ranges`) . . . 72

3.2.2 Concepts (`std::concepts`) . . . 73

3.2.3 `std::span` . . . 74

3.2.4 `std::string_view` . . . 75

3.2.5 Combined Modern Patterns . . . 76

3.2.6 Summary . . . 77

3.3 Unit Testing & Benchmarking in C++: GoogleTest, Catch2, `benchmark` Library, `valgrind`, Sanitizers . . . 78

3.3.1 Unit Testing . . . 78

3.3.2 Benchmarking . . . 80

3.3.3 Memory and Runtime Analysis . . . 81

3.3.4 Integrating Testing and Benchmarking . . . 83

3.3.5 Key Takeaways . . . 83

II Linear & Basic Structures (with C++ implementations) 84

4 Arrays & Vectors **86**

4.1 Static Array vs `std::vector` — Memory and Performance Tradeoffs . . . 86

4.1.1 Static Arrays . . . 86
4.1.2 `std::vector` . . . 88
4.1.3 Memory Layout and Cache Effects . . . 89
4.1.4 Performance Tradeoffs . . . 90
4.1.5 Guidelines for Algorithm Developers . . . 91
4.1.6 Summary . . . 92
4.2 In-Place Algorithms: Sliding Window, Two Pointers, Partitioning in C++ 93
4.2.1 Sliding Window Technique . . . 93
4.2.2 Two-Pointer Technique . . . 94
4.2.3 Partitioning (In-Place Reordering) . . . 96
4.2.4 Best Practices for In-Place Algorithms . . . 98
4.2.5 Summary . . . 98
4.3 Exercises: In-Place Rotation, Subarray Sums, Prefix/Suffix Arrays . . . 100
4.3.1 In-Place Array Rotation . . . 100
4.3.2 Subarray Sums . . . 101
4.3.3 Prefix and Suffix Arrays . . . 103
4.3.4 Suggested Exercises . . . 104
4.3.5 Summary . . . 105

5 Linked Lists 106
5.1 Single/Doubly Linked List Implementations in Modern C++ (Smart Pointers vs Raw Pointers) . . . 106
5.1.1 Singly Linked List . . . 106
5.1.2 Doubly Linked List . . . 109
5.1.3 Raw Pointers vs Smart Pointers — Tradeoffs . . . 112
5.1.4 Summary . . . 113
5.2 Common Algorithms: Reverse, Detect Cycle (Floyd), Merge Lists, Remove Nth Node from End . . . 114

5.2.1 Reversing a Singly Linked List . . . 114
5.2.2 Cycle Detection (Floyd's Tortoise and Hare Algorithm) . . . 115
5.2.3 Merging Two Sorted Linked Lists . . . 117
5.2.4 Removing the N-th Node from the End . . . 118
5.2.5 Summary of Common Linked List Algorithms . . . 120
5.3 Exercises and Tests: Memory-Leak Free Implementations, Iterator Support 121
5.3.1 Memory-Leak Free Implementations . . . 121
5.3.2 Iterator Support . . . 123
5.3.3 Testing Linked Lists . . . 125
5.3.4 Suggested Exercises . . . 126
5.3.5 Summary . . . 126

6 Stacks, Queues, Deques, and Priority Queues 128
6.1 STL Wrappers vs Custom Implementations: `std::stack`, `std::queue`, `std::deque`, `std::priority_queue` . . . 128
6.1.1 STL Wrappers Overview . . . 129
6.1.2 Example: `std::stack` . . . 129
6.1.3 Example: `std::queue` . . . 130
6.1.4 Example: `std::deque` . . . 131
6.1.5 Example: `std::priority_queue` . . . 132
6.1.6 Custom Implementations . . . 132
6.1.7 When to Use STL vs Custom . . . 134
6.1.8 Summary . . . 135
6.2 Use-Cases and Algorithmic Patterns (Expression Parsing, BFS, Sliding Window Optimums) . . . 136
6.2.1 Expression Parsing with Stacks . . . 136
6.2.2 Breadth-First Search (BFS) with Queues . . . 137
6.2.3 Sliding Window Optimizations with Deques . . . 139

6.2.4 Priority Queues in Algorithmic Patterns . . . 140
6.2.5 Summary of Patterns and Use-Cases . . . 141
6.3 Exercises: Monotonic Queue, K-Largest Using Heaps . . . 143
6.3.1 Monotonic Queue Exercise . . . 143
6.3.2 K-Largest Elements Using Heaps . . . 145
6.3.3 Suggested Exercises . . . 146
6.3.4 Key Takeaways . . . 147

7 Hashing and Unordered Containers 148
7.1 `std::unordered_map/set` Internals, Collision Behavior, Custom Hashers . 148
7.1.1 Internals of `std::unordered_map` and `std::unordered_set` . . . 148
7.1.2 Collision Behavior . . . 149
7.1.3 Custom Hash Functions . . . 150
7.1.4 Load Factor and Rehashing . . . 151
7.1.5 Performance Considerations . . . 152
7.1.6 Summary . . . 153
7.2 Hash-Based Algorithms: Frequency Counting, Two-Sum, Caching Strategies . . . 154
7.2.1 Frequency Counting . . . 154
7.2.2 Two-Sum Problem . . . 155
7.2.3 Caching Strategies (Memoization & LRU Cache) . . . 156
7.2.4 Best Practices . . . 157
7.2.5 Summary . . . 158
7.3 Exercises: Implement LRU Cache, Robin-Hood/Linear-Probing Sketch . . 159
7.3.1 Exercise: Implement LRU Cache . . . 159
7.3.2 Exercise: Robin-Hood and Linear-Probing Sketch . . . 161
7.3.3 Suggested Exercises . . . 164
7.3.4 Summary . . . 164

III Trees & Balanced Trees **165**

8 Binary Trees & Tree Traversals **167**

8.1 Node Representation, Recursive vs Iterative Traversal, Iterator Adapters . 167

8.1.1 Node Representation 167

8.1.2 Recursive Traversal 169

8.1.3 Iterative Traversal 170

8.1.4 Iterator Adapters for Trees 171

8.1.5 Summary 173

8.2 Algorithms — Preorder/Inorder/Postorder, Level-Order, Tree Serialization/Deserialization 174

8.2.1 Depth-First Traversals 174

8.2.2 Breadth-First Traversal (Level-Order) 177

8.2.3 Tree Serialization & Deserialization 178

8.2.4 Summary 180

8.3 Exercises — Reconstruct Tree from Traversals, Subtree Checks 182

8.3.1 Reconstructing a Tree from Traversals 182

8.3.2 Subtree Checks 185

8.3.3 Suggested Exercises 187

8.3.4 Summary 188

9 Binary Search Trees & Augmented Trees **189**

9.1 BST Operations, Invariants, Performance Edge Cases 190

9.1.1 The BST Invariant 190

9.1.2 Core BST Operations 191

9.1.3 Performance Considerations 193

9.1.4 Edge Cases to Address in Implementations 195

9.1.5 Summary 195

9.2 Augmented Trees for Range Queries and Order Statistics (`order_of_key`) 196
9.2.1 Motivation for Augmented Trees . . . 196
9.2.2 Core Augmentation: Subtree Size . . . 196
9.2.3 Order Statistics . . . 197
9.2.4 Range Queries . . . 199
9.2.5 Handling Duplicates . . . 199
9.2.6 Performance Considerations . . . 200
9.2.7 Practical Applications . . . 201
9.2.8 Summary . . . 201
9.3 Exercises — *kth Smallest, Interval Trees* . . . 202
9.3.1 Exercise: K-th Smallest Element in a BST . . . 202
9.3.2 Exercise: Interval Trees . . . 204
9.3.3 Testing and Benchmarking . . . 206
9.3.4 Summary . . . 207

10 Self-Balancing Trees (AVL, Red-Black) 208
10.1 AVL Rotations in C++ — Code Walkthrough . . . 208
10.1.1 Balance Factor and Rotation Trigger . . . 208
10.1.2 Node Structure in Modern C++ . . . 209
10.1.3 Single Rotations . . . 210
10.1.4 Double Rotations . . . 211
10.1.5 Rotation Integration in Insertions . . . 212
10.1.6 Walkthrough Example . . . 213
10.1.7 Key Insights . . . 214
10.1.8 Exercises . . . 214
10.2 Red-Black Tree Principles and Relation to `std::map` / `std::set` . . . 215
10.2.1 Red-Black Tree Properties . . . 215
10.2.2 Core Operations and Rebalancing . . . 215

10.2.3 Relation to `std::map` and `std::set` . . . 216
10.2.4 C++ Implementation Highlights . . . 217
10.2.5 Rotations in Red-Black Trees . . . 218
10.2.6 Comparison with AVL Trees . . . 218
10.2.7 Practical Takeaways . . . 219
10.3 Exercises — Implement an AVL with Unit Tests; Compare Against `std::set` Performance . . . 219
10.3.1 Exercise 1: Implement an AVL Tree . . . 220
10.3.2 Exercise 2: Unit Testing . . . 221
10.3.3 Exercise 3: Performance Comparison Against `std::set` . . . 222
10.3.4 Optional Extensions . . . 224
10.3.5 Summary . . . 224

11 B-Trees and External-Memory Structures 225
11.1 B-Tree Node Layout, Block I/O Considerations (C++ Structures for Disk-Backed Nodes) . . . 225
11.1.1 B-Tree Node Structure . . . 225
11.1.2 Disk Block Considerations . . . 226
11.1.3 Advantages of This Layout . . . 227
11.1.4 C++ Considerations . . . 228
11.1.5 Summary . . . 229
11.2 Practical Uses — Simple On-Disk Key-Value Store Prototype . . . 230
11.2.1 Design Overview . . . 230
11.2.2 Disk Node Structure . . . 231
11.2.3 Basic Operations . . . 231
11.2.4 Performance Considerations . . . 234
11.2.5 Extensions . . . 234
11.2.6 Summary . . . 234

11.3 Exercise — Small B-Tree Library Sketch with Tests 236
11.3.1 Library Structure 236
11.3.2 Key Operations 237
11.3.3 Unit Testing 239
11.3.4 Optional Extensions 240
11.3.5 Learning Outcomes 240

IV Graphs (Implemented in C++) 241

12 Graph Representations in C++ 243
12.1 Adjacency list/matrix, edge lists, compressed sparse row (CSR) for performance 243
12.1.1 Adjacency List 243
12.1.2 Edge List 245
12.1.3 Compressed Sparse Row (CSR) 246
12.1.4 Comparison Table 248
12.1.5 Choosing the Right Representation 249
12.2 Weighted graphs, directed/undirected, memory-oriented designs 250
12.2.1 Weighted Graphs 250
12.2.2 Directed vs. Undirected Graphs 252
12.2.3 Memory-Oriented Graph Designs 253
12.2.4 Performance and Trade-Offs 254

13 Traversal & Search 256
13.1 Depth-First Search (DFS) & Breadth-First Search (BFS) with Iterator-Based C++ APIs 256
13.1.1 Depth-First Search (DFS) 256
13.1.2 Breadth-First Search (BFS) 259

13.1.3 Iterator-Based API Design for Traversal . . . 260
13.1.4 Comparison: Recursive vs Iterative DFS vs BFS . . . 261
13.1.5 Best Practices in Modern C++ . . . 262
13.2 Applications of Graph Traversal . . . 263
13.2.1 Connected Components . . . 263
13.2.2 Cycle Detection . . . 265
13.2.3 Topological Sort . . . 267
13.2.4 Summary of Applications . . . 269

14 Shortest Paths **271**
14.1 Dijkstra's Algorithm . . . 271
14.1.1 Basic Dijkstra Algorithm . . . 271
14.1.2 Priority Queue Optimization . . . 272
14.1.3 Iterators and Modern C++ Features . . . 275
14.1.4 Performance Considerations . . . 275
14.1.5 Example Usage . . . 276
14.2 Bellman-Ford, SPFA Notes, and C++ Pitfalls . . . 278
14.2.1 Bellman-Ford Algorithm . . . 278
14.2.2 SPFA (Shortest Path Faster Algorithm) . . . 280
14.2.3 C++ Pitfalls to Avoid . . . 281
14.2.4 Comparison of Bellman-Ford vs SPFA . . . 282
14.2.5 Best Practices in Modern C++ . . . 283
14.3 A* Algorithm with C++ Heuristics and Custom Comparators . . . 284
14.3.1 Algorithm Overview . . . 284
14.3.2 Basic C++ Implementation Using `std::priority_queue` . . . 284
14.3.3 Custom Comparators in C++ . . . 287
14.3.4 Heuristic Design in C++ . . . 287
14.3.5 Performance and Pitfalls in C++ . . . 288

14.3.6 Example: A* on a 2D Grid . . . 289
14.3.7 Best Practices in Modern C++ . . . 289
14.4 Exercises — Multi-Source SSSP and Path Reconstruction Templates . . . 290
14.4.1 Multi-Source Single-Source Shortest Paths (SSSP) . . . 290
14.4.2 Path Reconstruction Templates . . . 293
14.4.3 Additional Exercises . . . 294
14.4.4 Best Practices in Modern C++ . . . 295

15 Minimum Spanning Trees & Union-Find 296
15.1 Kruskal's Algorithm with Efficient DSU . . . 296
15.1.1 Algorithm Overview . . . 296
15.1.2 Efficient DSU Implementation . . . 297
15.1.3 Kruskal's Algorithm Using DSU . . . 299
15.1.4 Example Usage . . . 300
15.1.5 Best Practices in Modern C++ . . . 301
15.1.6 Exercises . . . 302
15.2 Prim's Algorithm — Binary Heap vs Fibonacci Heap . . . 303
15.2.1 Prim's Algorithm Overview . . . 303
15.2.2 Prim Using Binary Heap (Standard Approach) . . . 304
15.2.3 Fibonacci Heap: Theoretical Advantage . . . 305
15.2.4 Comparison: Binary Heap vs Fibonacci Heap . . . 306
15.2.5 C++ Implementation Notes and Best Practices . . . 307
15.2.6 Exercises . . . 307
15.3 Exercises — MST Variants and Dynamic Connectivity . . . 308
15.3.1 MST Variants . . . 308
15.3.2 Dynamic Connectivity . . . 309
15.3.3 Best Practices and C++ Tips . . . 311
15.3.4 Additional Exercises for Mastery . . . 312

16 Network Flow & Advanced Graphs **313**

16.1 Ford-Fulkerson, Edmonds-Karp, Dinic — C++ Implementations and Performance Tradeoffs . . . 314

16.1.1 Ford-Fulkerson Method . . . 314

16.1.2 Edmonds-Karp Algorithm . . . 316

16.1.3 Dinic's Algorithm . . . 318

16.1.4 Performance Trade-offs . . . 321

16.1.5 Exercises . . . 322

16.2 Matching Algorithms and Min-Cost Max-Flow . . . 324

16.2.1 Bipartite Matching — Hopcroft–Karp Algorithm . . . 324

16.2.2 Flows with Capacities and Costs — Min-Cost Max-Flow (MCMF) 329

16.2.3 Performance Trade-offs . . . 332

16.2.4 Exercises . . . 333

16.3 Exercises — Bipartite Matching and Project Allocation Simulation . . . 334

16.3.1 Bipartite Matching Exercises . . . 334

16.3.2 Project Allocation Simulation . . . 335

16.3.3 Advanced Extensions . . . 338

16.3.4 Suggested Exercise Sequence . . . 338

V Design Paradigms & Algorithmic Techniques **339**

17 Divide and Conquer **341**

17.1 Merge Sort, Quicksort, and Recursion Patterns in C++ . . . 341

17.1.1 Merge Sort . . . 341

17.1.2 Quicksort . . . 343

17.1.3 Recursion Patterns in C++ . . . 345

17.1.4 Comparative Summary . . . 347

17.1.5 Exercises . . . 347
17.2 Parallel Divide-and-Conquer with `std::execution` and Thread Pools . . . 348
17.2.1 Parallel Divide-and-Conquer: Conceptual Overview . . . 348
17.2.2 `std::execution` in Parallel Divide-and-Conquer . . . 348
17.2.3 Thread Pools in Divide-and-Conquer . . . 349
17.2.4 Hybrid Approach: Execution Policies + Custom Thread Pools . . . 352
17.2.5 Performance Considerations . . . 353
17.2.6 Exercises . . . 353
17.2.7 Summary . . . 354
17.3 Exercises — Median of Medians, Parallel Mergesort . . . 355
17.3.1 Exercise: Median of Medians . . . 355
17.3.2 Exercise: Parallel Merge Sort . . . 357
17.3.3 Summary . . . 360

18 Dynamic Programming (DP) 361
18.1 Memoization vs. Tabulation — Idiomatic C++ Patterns . . . 362
18.1.1 Memoization (Top-Down Dynamic Programming) . . . 362
18.1.2 Tabulation (Bottom-Up Dynamic Programming) . . . 363
18.1.3 Comparing Memoization vs. Tabulation in C++ . . . 365
18.1.4 Advanced Idiomatic Patterns . . . 365
18.1.5 Summary . . . 367
18.2 DP on Sequences, Trees, and Graphs — Common Templates and Optimizations (Space Reduction) . . . 368
18.2.1 DP on Sequences . . . 368
18.2.2 DP on Trees . . . 370
18.2.3 DP on Graphs . . . 371
18.2.4 Space Reduction Techniques . . . 373
18.2.5 Summary . . . 374

18.3 Exercises: Knapsack Variants, Longest Increasing Subsequence with Patience Sorting (O(n log n)) . . . 375
18.3.1 Knapsack Variants . . . 375
18.3.2 Longest Increasing Subsequence (LIS) . . . 377
18.3.3 C++ Patterns for Efficiency . . . 378
18.3.4 Exercises . . . 378

19 Greedy Algorithms & Matroid Concepts 380
19.1 Greedy Correctness Proofs and C++ Greedy Idioms . . . 380
19.1.1 Greedy Algorithm Correctness Proofs . . . 381
19.1.2 C++ Greedy Idioms . . . 382
19.1.3 Typical Greedy Patterns in Practice . . . 384
19.1.4 Key Takeaways for Graduate-Level Readers . . . 385
19.2 Huffman Coding with Heaps and `std::priority_queue` Customization . . . 386
19.2.1 Problem Setting . . . 386
19.2.2 Greedy Insight . . . 386
19.2.3 Heap-Based Algorithm . . . 387
19.2.4 C++ Implementation with `std::priority_queue` . . . 387
19.2.5 Performance Analysis . . . 390
19.2.6 C++ Idioms and Customization . . . 390
19.2.7 Broader Connections . . . 391
19.2.8 Key Takeaways . . . 391
19.3 Exercises: Activity Selection, Interval Scheduling . . . 393
19.3.1 Activity Selection Problem . . . 393
19.3.2 Interval Scheduling Problem . . . 395
19.3.3 Exercises for the Reader . . . 397
19.3.4 Key Takeaways . . . 398

20 Randomized Algorithms & Probabilistic Methods **399**

20.1 Random Number Generation in C++ (`<random>`), Reproducible Experiments, Seeds . . . 399

20.1.1 Engines: Generating Pseudo-Randomness . . . 400

20.1.2 Distributions: Mapping Randomness . . . 401

20.1.3 Seeds: Ensuring Reproducibility . . . 402

20.1.4 Idiomatic Patterns in C++ . . . 402

20.1.5 Reproducible Experiments in Algorithm Design . . . 403

20.1.6 Summary . . . 403

20.2 QuickSelect, Hashing with Randomness, Monte Carlo Estimators . . . 404

20.2.1 QuickSelect: Randomized Selection . . . 404

20.2.2 Hashing with Randomness . . . 406

20.2.3 Monte Carlo Estimators . . . 407

20.2.4 Idiomatic C++ Considerations . . . 408

20.2.5 Key Takeaways . . . 409

20.3 Exercises: Randomized Algorithms for Median, Bloom Filter Sketch . . . 410

20.3.1 Randomized Median Selection . . . 410

20.3.2 Bloom Filter Sketch . . . 411

20.3.3 Learning Objectives . . . 414

20.3.4 Suggested Practice . . . 415

21 Approximation Algorithms & NP-Hard Problems **416**

21.1 Common Approximation Strategies Implemented in C++ . . . 416

21.1.1 Greedy Approximation . . . 417

21.1.2 Linear Programming Relaxation . . . 418

21.1.3 Randomized Rounding . . . 419

21.1.4 Local Search Heuristics . . . 419

21.1.5 PTAS / FPTAS Approaches . . . 420

21.1.6 Idiomatic C++ Patterns for Approximation 421
21.1.7 Exercises 421
21.1.8 Key Takeaways 422
21.2 Local Search, Greedy Approximation, PTAS Examples Where Applicable . 423
21.2.1 Local Search Heuristics 423
21.2.2 Greedy Approximation 425
21.2.3 Polynomial-Time Approximation Schemes (PTAS) 426
21.2.4 Idiomatic C++ Patterns 427
21.2.5 Exercises 428
21.2.6 Key Takeaways 428
21.3 Exercises: Vertex Cover Approximation, Traveling Salesman Heuristics . . 429
21.3.1 Vertex Cover Approximation Exercises 429
21.3.2 Traveling Salesman Problem (TSP) Heuristics 431
21.3.3 Learning Objectives 434
21.3.4 Suggested Practice 434

VI Performance, Concurrency & Low-level Concerns (C++ focused) 435

22 Memory & Cache-aware Algorithm Design 437

22.1 Data Layout, Locality, and Structure-of-Arrays vs Array-of-Structures . . . 437
22.1.1 Memory Locality and Cache Basics 438
22.1.2 Array-of-Structures (AoS) 438
22.1.3 Structure-of-Arrays (SoA) 439
22.1.4 Performance Implications 440
22.1.5 Hybrid Approaches 441
22.1.6 C++ Techniques for Cache Awareness 441

22.1.7 Exercises . . . 442
22.1.8 Key Takeaways . . . 442
22.2 Algorithms Optimized for Cache (Blocking, Tiling) with C++ Examples . 444
22.2.1 Cache Optimization Principles . . . 444
22.2.2 Blocking / Tiling Technique . . . 444
22.2.3 Example: Matrix Multiplication . . . 445
22.2.4 Tiling for Multi-Dimensional Arrays . . . 446
22.2.5 C++ Idiomatic Patterns . . . 447
22.2.6 Exercises . . . 448
22.2.7 Key Takeaways . . . 448

23 Parallel & Concurrent Algorithms 449
23.1 Threading Primitives in C++ (std::thread, Atomics, Mutexes) and Lock-Free Ideas . . . 449
23.1.1 The Role of Threads in Modern C++ . . . 449
23.1.2 Synchronization Primitives . . . 450
23.1.3 Lock-Free and Wait-Free Ideas . . . 454
23.1.4 Guidelines for Using Concurrency Primitives . . . 455
23.1.5 Summary . . . 456
23.2 Parallel Algorithms (`std::execution`) and Work-Stealing Patterns . . . 457
23.2.1 Parallel Algorithms in C++17 and Beyond . . . 457
23.2.2 Benefits of Parallel STL . . . 458
23.2.3 Limitations and Considerations . . . 459
23.2.4 Work-Stealing Patterns . . . 459
23.2.5 Example: Work-Stealing in Practice . . . 460
23.2.6 Combining Parallel STL and Work-Stealing . . . 461
23.2.7 Guidelines for Use . . . 461
23.2.8 Summary . . . 462

23.3 Exercises: Parallel Prefix Sum, Concurrent Queues 463
23.3.1 Parallel Prefix Sum (Scan) 463
23.3.2 Concurrent Queues 465
23.3.3 Exercise Variations 468
23.3.4 Summary 468

24 Metaprogramming & Compile-time Algorithms 470

24.1 Template Metaprogramming Basics for Algorithmic Tasks 471
24.1.1 Compile-Time Computation with Templates 471
24.1.2 Type Lists: Computation with Types 473
24.1.3 Applications of Template Metaprogramming 475
24.1.4 Modern TMP vs Historical TMP 475
24.1.5 Summary 475
24.2 Concepts & `constexpr` Algorithms in C++20/23 (constexpr Sorting, Compile-time DP) 477
24.2.1 Concepts and Constrained Algorithms 477
24.2.2 Expanded `constexpr` in C++20/23 478
24.2.3 Example: `constexpr` Sorting 478
24.2.4 Compile-time Dynamic Programming 480
24.2.5 Practical Applications of `constexpr` Algorithms 481
24.2.6 Limitations and Best Practices 481
24.3 Exercises: Static-Sequence Algorithms, `consteval` Usage 483
24.3.1 Static-Sequence Algorithms 483
24.3.2 `consteval` Usage 485
24.3.3 Exercise Ideas 486
24.3.4 Best Practices for Exercises 487

25 Profiling, Benchmarking & Optimization Workflow **488**

25.1 Using Profilers (gprof, perf), Sanitizers (ASAN, UBSAN), and Compiler Flags . . . 488

25.1.1 Profilers . . . 488

25.1.2 Sanitizers . . . 490

25.1.3 Compiler Flags . . . 491

25.1.4 Recommended Workflow . . . 493

25.2 Micro-optimizations vs Algorithmic Improvements — Case Studies in C++ 494

25.2.1 Micro-optimizations . . . 494

25.2.2 Algorithmic Improvements . . . 495

25.2.3 Case Study: Searching in C++ . . . 496

25.2.4 Case Study: Matrix Multiplication . . . 497

25.2.5 Guiding Principles . . . 497

25.3 Exercises — Profile and Improve Small C++ Projects . . . 499

25.3.1 Exercise: Profiling a Naive Sorting Benchmark . . . 499

25.3.2 Exercise: Memory Leak Detection with ASAN . . . 500

25.3.3 Exercise: Cache-Aware Optimization . . . 500

25.3.4 Exercise: Micro-Optimization vs Algorithmic Improvement . . . 501

25.3.5 Exercise: Multithreaded vs Single-Threaded Performance . . . 502

25.3.6 Project: Profile-and-Improve a Small CLI Tool . . . 502

25.3.7 Recommended Workflow for Each Exercise . . . 503

VII Capstone Projects **505**

26 Project A — High-performance Graph Library **507**

26.1 Design Goals, API, Iterators, Memory Layout (CSR) . . . 507

26.1.1 Design Goals . . . 508

26.1.2 API Design . . . 509
26.1.3 Iterators . . . 510
26.1.4 Memory Layout: Compressed Sparse Row (CSR) . . . 511
26.2 Implementations — SSSP, MST, Centrality Measures . . . 514
26.2.1 Single-Source Shortest Path (SSSP) . . . 514
26.2.2 Minimum Spanning Tree (MST) . . . 516
26.2.3 Centrality Measures . . . 517
26.2.4 Performance Considerations . . . 518
26.3 Tests & Benchmarks Against Common Datasets . . . 520
26.3.1 Testing Strategy . . . 520
26.3.2 Benchmarking Strategy . . . 521
26.3.3 Example Benchmarking Workflow . . . 522
26.3.4 Automation and Reproducibility . . . 524
26.3.5 Example Observations . . . 524

27 Project B — Mini Compiler / Interpreter 525
27.1 Lexing and Parsing with Modern C++ (Recursive Descent, Parser Combinators) . . . 525
27.1.1 Lexical Analysis (Lexer) . . . 525
27.1.2 Syntax Analysis (Parser) . . . 528
27.1.3 Modern C++ Features Applied . . . 531
27.1.4 Best Practices . . . 531
27.2 AST Transformations, Control-Flow Algorithms, Simple Optimization Passes . . . 533
27.2.1 AST Representation . . . 533
27.2.2 AST Transformations . . . 534
27.2.3 Control-Flow Algorithms . . . 535
27.2.4 Simple Optimization Passes . . . 536

27.2.5 Modern C++ Techniques Applied 537
27.3 Exercises — Generate Three-Address Code, Simple Register Allocation . . 539
27.3.1 Three-Address Code (TAC) Generation 539
27.3.2 Simple Register Allocation 541
27.3.3 Combined Exercise Workflow 543
27.3.4 Learning Outcomes 543

28 Project C — Algorithmic Trading Backtester (example of time-series algorithms) 545
28.1 Streaming Data Algorithms, Sliding Windows, Online Learning Sketches . 545
28.1.1 Streaming Data Algorithms 546
28.1.2 Sliding Window Techniques 547
28.1.3 Online Learning Sketches 548
28.1.4 Integrating Streaming Algorithms into the Backtester 550
28.1.5 Summary 551
28.2 Backtesting Engine Design and Performance Constraints 552
28.2.1 Core Design Goals 552
28.2.2 Engine Architecture 553
28.2.3 Performance Constraints 555
28.2.4 Modern C++ Techniques Applied 556
28.2.5 Example Engine Loop 556
28.2.6 Summary 557
28.3 Exercises — Implement Moving Average Crossover Strategy, Evaluate Latency 558
28.3.1 Moving Average Crossover Strategy 558
28.3.2 Integrating with the Backtesting Engine 559
28.3.3 Evaluating Latency 560
28.3.4 Advanced Extensions 561

28.3.5 Learning Outcomes 561

VIII Testing, Reproducibility & Research Practices 563

29 Testing Algorithm Correctness in C++ 565

29.1 Property-Based Testing, Fuzzing Inputs, Determinism in Tests 565

29.1.1 Property-Based Testing 565

29.1.2 Fuzzing Inputs 567

29.1.3 Determinism in Tests 569

29.1.4 Advantages of Property-Based Testing and Fuzzing 569

29.1.5 Summary 570

29.2 Using GoogleTest / QuickCheck-Style Libraries, CI Integration 571

29.2.1 GoogleTest for Unit Testing 571

29.2.2 QuickCheck-Style Property-Based Testing 572

29.2.3 Continuous Integration (CI) Integration 573

29.2.4 Combining Unit Tests and Property-Based Tests 574

29.2.5 Summary 575

30 Reproducible Experiments & Data Sets 576

30.1 Dataset Management, Synthetic Data Generators (C++), Seeding, and Reporting Standards 576

30.1.1 Dataset Management 577

30.1.2 Synthetic Data Generators in C++ 578

30.1.3 Seeding and Determinism 579

30.1.4 Reporting Standards 580

30.1.5 Summary 581

30.2 Publishing Code and Experiments — Packaging with CMake, Docker, and Minimal Reproducibility Checklist 583

30.2.1 Packaging C++ Experiments with CMake 583
30.2.2 Containerization with Docker 584
30.2.3 Minimal Reproducibility Checklist 586
30.2.4 Best Practices for Publishing 587
30.2.5 Summary 587

Appendices **588**

Appendix A – C++ Cheat Sheet for Algorithm Developers 588
Appendix B – Common Code Templates 597
Appendix C – Advanced Data Structures 603
Appendix D – CMake Template and CI Example 610
Appendix E – Recommended Reading & Research Papers 616
Appendix F – Solution Sketches & Sample Outputs 623

Author's Preface

The design and efficiency of algorithms lie at the heart of every powerful software system. Understanding algorithms not only refines a programmer's logic but also strengthens their ability to craft optimized, elegant, and scalable solutions.
Modern C++ Algorithms: A Graduate-Level Companion is written for advanced learners who seek to master algorithmic thinking and its implementation using Modern C++. It is not an introductory text on programming; rather, it serves as a comprehensive companion for graduate-level students, researchers, and professional developers who already possess a strong foundation in C++ and aim to deepen their knowledge of both the theoretical foundations and practical design of algorithms.
This book bridges the gap between abstract algorithmic theory and real-world implementation, illustrating how the expressive power of Modern C++ — through templates, generic programming, and the STL — enables elegant and efficient solutions to complex computational problems.
It is my hope that this work inspires readers to think algorithmically, write efficiently, and innovate with confidence in the ever-evolving landscape of software development.
The book is still undergoing careful review and testing of every part. It remains subject to correction, and this is a draft version intended for those who wish to contribute by sharing their opinions, pointing out any inaccuracies, or suggesting improvements.
For more discussions and valuable content about
Modern C++ Algorithms: A Graduate-Level Companion,

I invite you to follow me on **LinkedIn**:

https://linkedin.com/in/aymanalheraki

You can also visit the company website:

https://simplifycpp.org

Wishing everyone success and prosperity.

Ayman Alheraki

Part I

Foundations (C++-centric)

Chapter 1

Preface & How to Use This Book

1.1 Target Audience and Prerequisites (C++17/20/23)

This book is written for advanced learners who seek to master algorithmic thinking and its implementation using Modern C++. It is not an introductory text on programming; rather, it is designed as a companion for graduate-level students, researchers, and professional developers who already have a solid foundation in C++ and wish to deepen their expertise in both theoretical and practical aspects of algorithms.
The **target audience** includes:

1. **Graduate and postgraduate students in computer science, mathematics, or engineering**

    - These readers will benefit from the rigorous approach to algorithm design, analysis, and implementation, especially in areas involving optimization, graph theory, and parallelization.

- The book is designed to complement advanced coursework, seminar projects, and thesis research.

2. **Researchers and academics**

- Those conducting research in areas such as high-performance computing, numerical optimization, computational geometry, or machine learning systems will find ready-to-use C++ implementations alongside theoretical insights.
- The focus on Modern C++ standards ensures that the techniques are relevant for current and future research.

3. **Professional software developers and systems engineers**

- Developers working in domains like finance, scientific computing, embedded systems, or large-scale distributed software will find value in mastering efficient algorithm design with Modern C++ idioms.
- Emphasis on efficiency, concurrency, and real-world constraints prepares readers for the challenges of production-level codebases.

4. **Enthusiasts of Modern C++**

- Readers who are already proficient in earlier C++ standards and want to transition their thinking to C++17, C++20, and C++23 idioms will gain a structured path toward adopting concepts, ranges, coroutines, and modules in algorithmic contexts.

1.1.1 Prerequisites

To maximize the benefit from this book, readers are expected to meet the following prerequisites:

1. **C++ Knowledge**
    - A strong understanding of C++ up to at least the **C++17 standard** is required.
    - Familiarity with advanced constructs like templates, smart pointers, lambda expressions, and move semantics is assumed.
    - Exposure to **C++20/23 features** such as concepts, ranges, coroutines, and the expanded standard library is highly recommended. These features will be used throughout the book to simplify and modernize algorithm implementation.
2. **Mathematical Foundations**
    - Readers should have working knowledge of **discrete mathematics, linear algebra, probability, and basic number theory**.
    - Understanding of asymptotic notation (Big-O, Big-Theta, Big-Omega) and algorithmic complexity is essential.
3. **Computer Science Foundations**
    - Prior coursework or experience with **data structures (arrays, linked lists, stacks, queues, trees, graphs, hash tables)** and their trade-offs is assumed.
    - Basic familiarity with recursion, dynamic programming, and divide-and-conquer strategies is necessary.

4. **Development Tools**

   - Readers should be comfortable with **compilers supporting C++20/23**, such as GCC, Clang, or MSVC.
   - Basic familiarity with **CMake** or modern build systems is assumed, as the examples will make use of such tools for modular compilation and testing.

5. **Optional but Helpful Background**

   - Knowledge of parallel programming models (threads, OpenMP, or GPU programming) will be helpful in later chapters dealing with concurrency and performance.
   - Familiarity with software engineering practices such as version control, unit testing, and benchmarking will make it easier to apply the book's lessons in practical contexts.

1.1.2 Positioning of This Book

Unlike introductory algorithm texts that focus purely on pseudocode or language-agnostic explanations, this book emphasizes **real-world C++ implementations aligned with the latest standards**. Each algorithm is not only presented with its theoretical foundation but also translated into **idiomatic C++17/20/23 code** that leverages modern features for clarity, efficiency, and maintainability.
By setting high prerequisites, the book ensures that the content can dive directly into graduate-level algorithmic strategies without repeating elementary concepts. This allows readers to engage with advanced material such as graph flows, optimization heuristics, and concurrency-aware algorithms at a professional depth.

1.2 Coding Standards Used in Examples (formatter, naming, header structure)

To ensure clarity, consistency, and professionalism across all code examples, this book adheres to a strict set of coding standards. These conventions are chosen to balance readability, maintainability, and alignment with modern C++ practices, making it easier for readers to follow the examples and adopt the same standards in their own projects.

1.2.1 Code Formatting and Style

All code snippets are formatted using a consistent automated formatter, aligned with the widely used **LLVM/Clang-Format style**. The primary goals are readability, uniformity, and modern conventions.

- **Indentation**:
  - Four spaces are used per indentation level.
  - Tabs are not used.
- **Line Length**:
  - Code lines are kept within 100–120 characters to improve readability in both printed and digital formats.
- **Braces**:
  - Opening braces are placed on the same line as the declaration or statement:

```
if (condition) {
    // code
} else {
    // alternative
}
```

- **Whitespace**:
  - One space after keywords (`if`, `for`, `while`, `switch`).
  - Spaces around operators (`=`, `+`, `-`, `*`, `/`, `&&`, `||`).
  - No trailing whitespace at the end of lines.
- **Comments**:
  - Use `//` for short, explanatory inline comments.
  - Use `/** ... */` (Doxygen style) for documenting functions, classes, and templates, making examples suitable for automatic documentation tools.

1.2.2 Naming Conventions

Naming conventions follow widely accepted C++ community standards, with clarity and semantic meaning prioritized.

- **Variables and Functions**:
  - Use **camelCase** for function names and local variables.
  - Example:

```
int computeDistance(int x, int y);
double averageValue(const std::vector<int>& values);
```

- **Classes, Structs, and Types**:
  - Use **PascalCase** for class and struct names.
  - Example:

```
class GraphSolver {
public:
    void run();
};
```

- **Constants and Enums**:
  - Use **ALL_CAPS_WITH_UNDERSCORES** for global constants.
  - Strongly typed enums (`enum class`) use **PascalCase** for enumerators.
  - Example:

```
template <typename T>
class Stack { /* ... */ };
```

- **Templates and Generic Parameters**:
  - Template parameters use **PascalCase**, often a single uppercase letter or descriptive word.

- Example:

```
namespace algo::graph {
    void dijkstra();
}
```

- **Namespaces**:

  - Namespaces in C++ organize code and prevent name conflicts. Use concise, lowercase names for namespaces, and employ nested namespaces for logical grouping.

  - Example:

```
#include "graph_solver.hpp"
#include <iostream>
#include <vector>
#include <queue>

namespace graph {
    namespace solver {
        class GraphSolver {
        public:
            void solve() {
                std::cout << "Solving graph..." << std::endl;
            }
        };
    }
}
```

1.2.3 Header and Source Structure

Each example adheres to a clean and consistent file organization, ensuring modularity and easy integration into larger projects.

- **Include Order**:

  Maintain a consistent include order to improve readability and reduce dependency issues:

  1. **Corresponding header file** (`.h` or `.hpp`).
  2. **Standard library headers**.
  3. **Third-party library headers** (if applicable).
  4. **Project-specific headers**.

  Example:

```
#include "graph_solver.hpp"   // Corresponding header
#include <iostream>           // Standard library
#include <vector>
#include <queue>
#include <boost/graph/adjacency_list.hpp> // Third-party (example)
#include "utils/logger.hpp"   // Project-specific
```

- **Header File Structure**:

  – Use `#pragma once` to prevent multiple inclusion.

  – Keep declarations (interfaces, templates, constants) in header files, and place definitions in `.cpp` files whenever possible.

Example (graph_solver.hpp):

```
// graph_solver.hpp
#pragma once
#include <vector>

class GraphSolver {
public:
    GraphSolver(int vertices);
    void addEdge(int u, int v, int weight);
    void runDijkstra(int source);

private:
    int vertexCount;
    std::vector<std::vector<int>> adjacencyMatrix;
};
```

Source File Structure:

- Source files should contain the function and method definitions, keeping headers lean and focused on declarations.

Example:

```
// graph_solver.cpp
#include "graph_solver.hpp"
#include <queue>
#include <limits>
#include <iostream>

GraphSolver::GraphSolver(int vertices)
    : vertexCount(vertices),
```

```
      adjacencyMatrix(vertices, std::vector<int>(vertices, 0)) {}

void GraphSolver::addEdge(int u, int v, int weight) {
    adjacencyMatrix[u][v] = weight;
}

void GraphSolver::runDijkstra(int source) {
    std::vector<int> dist(vertexCount, std::numeric_limits<int>::max());
    dist[source] = 0;

    using Node = std::pair<int, int>; // (distance, vertex)
    std::priority_queue<Node, std::vector<Node>, std::greater<>> pq;
    pq.push({0, source});

    while (!pq.empty()) {
        int d = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        if (d > dist[u]) continue;

        for (int v = 0; v < vertexCount; ++v) {
            if (adjacencyMatrix[u][v] > 0) {
                int newDist = dist[u] + adjacencyMatrix[u][v];
                if (newDist < dist[v]) {
                    dist[v] = newDist;
                    pq.push({newDist, v});
                }
            }
        }
    }
```

```
    std::cout << "Shortest distances from source " << source << ":\n";
    for (int i = 0; i < vertexCount; ++i) {
        std::cout << "Vertex " << i << ": ";
        if (dist[i] == std::numeric_limits<int>::max())
            std::cout << "INF\n";
        else
            std::cout << dist[i] << "\n";
    }
}
```

1.2.4 Modern C++ Practices

To reinforce best practices, all code examples:

- Prefer **`constexpr`, `noexcept`, `[[nodiscard]]`, and `auto`** where applicable.
- Use **RAII (Resource Acquisition Is Initialization)** for resource management.
- Favor **`std::unique_ptr` and `std::shared_ptr`** over raw pointers.
- Use **ranges, concepts, and structured bindings** (C++20/23) when demonstrating algorithms.
- Avoid deprecated or unsafe constructs (e.g., `new/delete` in favor of smart pointers, `printf` in favor of `std::format`).

This standardized approach ensures that every code example is:

- Readable and immediately understandable.
- Idiomatic to modern C++ (C++17/20/23).
- Suitable for integration into production or research codebases.

1.3 Build & Run: CMake Minimal Template, Compiler Flags, Sanitizers, and Test Runner Setup

This book emphasizes not only algorithmic design but also reproducible and professional software engineering practices. To that end, all examples are structured around **CMake-based builds**, with carefully chosen compiler flags, runtime sanitizers, and lightweight testing frameworks. This ensures that every code sample is easy to compile, portable across platforms, and robust against common runtime errors.

1.3.1 Minimal CMake Template

CMake is the de facto build system generator for modern C++ projects. Its declarative syntax, wide compiler support, and integration with IDEs make it an ideal choice for academic and professional work.
A minimal project template used throughout this book looks as follows:

```
cmake_minimum_required(VERSION 3.20)
project(ModernCppAlgorithms VERSION 1.0 LANGUAGES CXX)

# Set the standard explicitly
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

# Enable warnings
if (MSVC)
    add_compile_options(/W4 /permissive-)
else()
```

```
    add_compile_options(-Wall -Wextra -Wpedantic -Werror)
endif()

# Define an executable
add_executable(example main.cpp)

# Link libraries (if needed)
# target_link_libraries(example PRIVATE some_library)
```

This template enforces the C++23 standard, though examples are compatible with C++17 and C++20 as well. Extensions are disabled to maintain portability and encourage standard-compliant code.

1.3.2 Recommended Compiler Flags

Compiler warnings and optimizations are crucial for producing reliable code. The following flags are encouraged throughout the book:

- **Clang/GCC**:
  - `-Wall -Wextra -Wpedantic -Werror` — enables strict warnings and treats them as errors.
  - `-O2` or `-O3` — enables optimizations.
  - `-g` — includes debug symbols.
  - `-fsanitize=address,undefined` (when enabled) — runtime sanitizers.
- **MSVC**:
  - `/W4` — high warning level.
  - `/permissive-` — enforces strict ISO compliance.

– `/std:c++20` or `/std:c++latest` — ensures correct language version.

Build configurations are typically separated into **Debug** (with sanitizers and debug symbols) and **Release** (with optimizations).
Example workflow:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug
cmake --build build
./build/example
```

1.3.3 Sanitizers

To minimize undefined behavior and catch subtle runtime issues, examples in this book make extensive use of sanitizers. These are runtime checks supported by Clang and GCC that detect memory and logic errors:

- **AddressSanitizer (ASan)**: Detects memory leaks, buffer overflows, and use-after-free.
- **UndefinedBehaviorSanitizer (UBSan)**: Catches undefined behavior such as division by zero, invalid casts, or signed integer overflow.
- **ThreadSanitizer (TSan)**: Helps detect data races in multithreaded algorithms.

CMake configuration snippet for sanitizers:

```
if (CMAKE_CXX_COMPILER_ID MATCHES "Clang|GNU")
    add_compile_options(-fsanitize=address,undefined -fno-omit-frame-pointer)
    add_link_options(-fsanitize=address,undefined -fno-omit-frame-pointer)
endif()
```

For multithreaded sections of the book, ThreadSanitizer can be enabled separately:

```
add_compile_options(-fsanitize=thread)
add_link_options(-fsanitize=thread)
```

1.3.4 Test Runner Setup

Every algorithm in this book can be validated with small, reproducible tests. To provide structure and reproducibility, the **CTest framework** (bundled with CMake) is used as the default test runner.
Minimal testing setup in `CMakeLists.txt`:

```
enable_testing()

add_executable(test_example test_example.cpp)
add_test(NAME ExampleTest COMMAND test_example)
```

Test code can be simple assert-based programs:

```
#include <cassert>
#include "algorithm.hpp"

int main() {
    assert(computeDistance(0, 3) == 3);
    assert(computeDistance(-1, 2) == 3);
    return 0;
}
```

For more advanced examples, lightweight frameworks such as **Catch2** or **GoogleTest** can be integrated. However, the book defaults to minimal test runners to keep the focus on algorithms, not external dependencies.

1.3.5 Recommended Workflow

1. **Clone or write the example code**.
2. **Configure the build system**:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug
```

3. **Compile the code**:

```
cmake --build build
```

4. **Run the example**:

```
./build/example
```

5. **Run the test suite**:

```
ctest --test-dir build
```

This workflow ensures every algorithm is validated in a reproducible, portable, and professional way.

Chapter 2

Algorithmic Thinking with C++

2.1 What is an Algorithm? C++ Examples as First-Class Citizens

An **algorithm** is a finite, well-defined sequence of steps designed to solve a specific problem or perform a computation. In computer science, algorithms are the blueprint for any computational process, serving as the bridge between theoretical problem-solving and practical implementation.
In the context of C++, algorithms are **first-class citizens**, meaning they are not merely abstract concepts but entities that can be directly represented, manipulated, and executed in code. Modern C++ provides multiple mechanisms—functions, function objects, templates, ranges, and lambdas—to encode algorithms in a clear, type-safe, and reusable manner.

2.1.1 Defining Algorithms

At its core, an algorithm has several defining characteristics:

1. **Input**: One or more well-defined values or data structures.
2. **Output**: A result that is produced after finite steps.
3. **Determinism**: Given the same input, the output is predictable and repeatable.
4. **Finiteness**: An algorithm must terminate after a finite number of steps.
5. **Effectiveness**: Each step must be executable using a finite, well-defined operation.

Example in C++: computing the factorial of a number recursively:

```cpp
#include <iostream>

int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

int main() {
    int value = 5;
    std::cout << "Factorial of " << value << " is " << factorial(value) << '\n';
}
```

Here, `factorial` is a complete, self-contained algorithm: it has input (`n`), output (the factorial), determinism, and a finite number of steps.

2.1.2 Algorithms as First-Class Citizens in C++

C++ allows algorithms to be represented and manipulated in ways that treat them as **first-class entities**:

- **Functions**: The most basic building blocks of algorithms.

```
int add(int a, int b) { return a + b; }
```

- **Function Objects (Functors)**: Objects that behave like functions, allowing stateful algorithms.

```
struct Multiply {
    int factor;
    Multiply(int f) : factor(f) {}
    int operator()(int x) const { return x * factor; }
};
```

- **Lambdas**: Inline, anonymous algorithms, widely used with modern STL containers and algorithms.

```
auto square = [](int x) { return x * x; };
std::cout << square(4);  // outputs 16
```

- **Templates and Generic Algorithms**: C++ templates allow algorithms to operate on any compatible type.

```
template <typename T>
T sum(T a, T b) { return a + b; }
```

- **STL Algorithms**: The Standard Template Library provides pre-built, high-performance algorithms such as `std::sort`, `std::accumulate`, `std::transform`, and `std::find`, which treat algorithms as generic, composable entities.

```cpp
#include <vector>
#include <algorithm>
#include <numeric>
#include <iostream>

int main() {
    std::vector<int> v{1, 2, 3, 4, 5};
    int total = std::accumulate(v.begin(), v.end(), 0);
    std::cout << "Sum: " << total << '\n';

    std::sort(v.begin(), v.end(), [](int a, int b){ return b < a; });  //
    ↪  descending
}
```

By treating algorithms as **first-class citizens**, C++ enables a high degree of **reusability, composability, and abstraction**. Functions, templates, and STL algorithms can be passed, stored, and invoked like data, allowing flexible algorithm design and higher-level program reasoning.

2.1.3 Algorithmic Thinking in C++

Algorithmic thinking is the practice of designing, analyzing, and implementing algorithms efficiently. In C++, this involves:

1. **Choosing the right data structure**: Efficient algorithms rely on data organization, e.g., arrays for random access, linked lists for dynamic insertion, hash tables for fast lookup.

2. **Deciding the approach**: Whether to use recursion, iteration, divide-and-conquer, or dynamic programming.

3. **Leveraging Modern C++ features**: Using ranges, concepts, lambdas, and STL algorithms to simplify implementation while maintaining performance.

4. **Evaluating complexity**: Understanding the time and space complexity of an algorithm and optimizing where necessary.

Example: Using `std::transform` with a lambda for element-wise operation:

```
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    std::vector<int> data{1, 2, 3, 4, 5};
    std::vector<int> squared(data.size());

    std::transform(data.begin(), data.end(), squared.begin(),
                   [](int x) { return x * x; });

    for (auto val : squared)
        std::cout << val << " ";  // Output: 1 4 9 16 25
}
```

Here, `std::transform` abstracts iteration, and the lambda defines the algorithm concisely. This demonstrates how Modern C++ elevates algorithms to first-class, composable, and reusable entities.

2.1.4 Summary

- An algorithm is a **well-defined, finite sequence of steps** to solve a problem.
- C++ enables algorithms to be **first-class citizens** via functions, lambdas, templates, and STL utilities.

- Modern C++ encourages **reusable, composable, and type-safe algorithmic design**.
- Algorithmic thinking in C++ combines problem-solving, data structures, language features, and complexity analysis to implement efficient and maintainable solutions.

By mastering this mindset, readers are prepared to engage with both classical and advanced algorithmic patterns presented in later chapters.

2.2 Complexity Notation (Big-O / Θ / Ω) Illustrated with C++ Microbenchmarks

Understanding algorithmic complexity is fundamental to algorithmic thinking. Complexity notations provide a formal framework to **measure, compare, and reason about algorithm performance** in terms of time and space. In C++, these theoretical concepts can be **experimentally validated using microbenchmarks**, bridging the gap between theory and practice.

2.2.1 Introduction to Complexity Notation

Algorithmic complexity describes how an algorithm's **resource usage grows** with input size. The most common measures are:

1. **Big-O Notation (O)**:
    - Represents the **upper bound** on runtime or space.
    - Provides a worst-case guarantee: the algorithm will not exceed this growth rate.
    - Example: Insertion sort has $O(n^2)$ worst-case complexity.
2. **Big-Theta Notation (Θ)**:
    - Represents the **tight bound**, where the growth rate is both an upper and lower bound.
    - Provides an exact asymptotic behavior.
    - Example: Merge sort has $\Theta(n \log n)$ complexity in both average and worst cases.

3. **Big-Omega Notation (Ω)**:
    - Represents the **lower bound**, i.e., the minimum amount of work an algorithm will do.
    - Example: Searching an unordered list has $\Omega(1)$ for best-case scenario (if the target is first).

These notations help **abstract away machine-specific constants**, focusing on the scalability of algorithms as input size increases.

2.2.2 Microbenchmarks in C++

Microbenchmarks provide empirical evidence of complexity. In Modern C++, we can leverage `<chrono>` to measure execution time precisely.

Example 1: Linear Search (O(n) worst-case)

```
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm>

int linearSearch(const std::vector<int>& v, int key) {
    for (size_t i = 0; i < v.size(); ++i) {
        if (v[i] == key) return static_cast<int>(i);
    }
    return -1;
}

int main() {
    const int N = 1'000'000;
    std::vector<int> data(N);
```

```
    std::iota(data.begin(), data.end(), 0);

    auto start = std::chrono::high_resolution_clock::now();
    int index = linearSearch(data, N - 1);  // worst-case
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Index: " << index << ", Time: " << elapsed.count() << "s\n";
}
```

- Observed runtime increases linearly with `N`.
- Matches theoretical O(n) behavior.

Example 2: Binary Search (O(log n) worst-case)

```
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm>

int main() {
    const int N = 1'000'000;
    std::vector<int> data(N);
    std::iota(data.begin(), data.end(), 0);

    auto start = std::chrono::high_resolution_clock::now();
    int index = std::binary_search(data.begin(), data.end(), N - 1) ? N - 1 : -1;
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Found: " << index << ", Time: " << elapsed.count() << "s\n";
}
```

- Time grows logarithmically with `N`.
- Demonstrates how algorithm choice dramatically affects performance.

2.2.3 Visualizing Complexity

Readers are encouraged to **plot runtime vs input size** to visually validate complexity. Example workflow:

1. Vary `N` (input size) over multiple orders of magnitude.
2. Measure execution time using `<chrono>`.
3. Plot `N` on the x-axis and runtime on the y-axis.
4. Compare linear, logarithmic, quadratic, and cubic algorithms to understand their growth curves.

This practice highlights **practical differences** between O(n), O(log n), and $O(n^2)$ algorithms, showing how even small asymptotic differences can dominate performance for large inputs.

2.2.4 Space Complexity

C++ algorithms are also evaluated in terms of **memory usage**. Common examples:

- **In-place algorithms**: Modify the input array, minimal extra space (e.g., in-place quicksort, O(log n) space for recursion).
- **Auxiliary-space algorithms**: Require additional structures (e.g., merge sort, O(n) extra space).

Microbenchmarks can include **memory profiling** using tools such as Valgrind or AddressSanitizer to detect excessive allocations or leaks.

2.2.5 Modern C++ Techniques for Complexity Analysis

- **`std::chrono`**: High-resolution timing.
- **`std::vector` and other STL containers**: Provide predictable performance characteristics.
- **Lambdas and `std::function`**: Facilitate benchmarking multiple algorithms concisely.

Example: Comparing two algorithms using a lambda-based timer:

```
auto measure = [](auto func) {
    auto start = std::chrono::high_resolution_clock::now();
    func();
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration<double>(end - start).count();
};

double linear_time = measure([&] { linearSearch(data, N - 1); });
double binary_time = measure([&] { std::binary_search(data.begin(), data.end(), N -
↪  1); });

std::cout << "Linear: " << linear_time << "s, Binary: " << binary_time << "s\n";
```

This approach promotes **reproducible, maintainable benchmarks**, consistent with professional C++ standards.

2.2.6 Key Takeaways

- Big-O, Θ, and Ω notations provide **formal tools to describe algorithm performance**.
- C++ enables **direct empirical validation** of complexity through microbenchmarks.
- Modern C++ features such as lambdas, STL algorithms, and `<chrono>` make complexity analysis **concise and reproducible**.
- Understanding both theoretical and practical behavior is essential for algorithmic thinking and for selecting the right algorithm for real-world problems.

2.3 Practical Measurement: `chrono`, `std::execution`, CPU Cycles, and Pitfalls

In algorithmic development, **practical measurement** complements theoretical complexity analysis. While Big-O notation predicts scaling behavior, actual performance depends on hardware, compiler optimizations, and memory hierarchy. Modern C++ offers tools to **measure execution time precisely, leverage parallel execution policies, and analyze low-level CPU usage**.

2.3.1 Measuring Time with `<chrono>`

The `<chrono>` library provides high-resolution clocks to measure execution time with precision. Typical usage involves capturing timestamps before and after a function call:

```
#include <chrono>
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v(1'000'000);
    std::iota(v.begin(), v.end(), 0);

    auto start = std::chrono::high_resolution_clock::now();
    std::sort(v.begin(), v.end(), std::greater<>());  // sample algorithm
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double, std::milli> elapsed = end - start;
    std::cout << "Sorting took: " << elapsed.count() << " ms\n";
}
```

Key points:

- **`high_resolution_clock`** provides maximum available precision.
- `duration` can be expressed in **seconds, milliseconds, microseconds, or nanoseconds**.
- Always **warm up the CPU cache** before measuring to avoid skewed results.

2.3.2 Parallel Execution with `std::execution`

C++17 introduced **parallel algorithms** in the Standard Library. Execution policies allow simple parallelization without explicit threads:

```
#include <vector>
#include <algorithm>
#include <execution>
#include <numeric>
#include <iostream>

int main() {
    std::vector<int> data(10'000'000, 1);

    auto start = std::chrono::high_resolution_clock::now();
    int sum = std::reduce(std::execution::par, data.begin(), data.end());
    auto end = std::chrono::high_resolution_clock::now();

    std::cout << "Parallel sum: " << sum
              << ", Time: "
              << std::chrono::duration<double, std::milli>(end - start).count()
              << " ms\n";
}
```

- `std::execution::seq` — sequential execution (default).
- `std::execution::par` — parallel execution using multiple threads.
- `std::execution::par_unseq` — parallel and vectorized execution.

Note: The effectiveness of parallel execution depends on input size and system hardware; small datasets may see worse performance due to threading overhead.

2.3.3 Measuring CPU Cycles

For **low-level performance analysis**, measuring CPU cycles can reveal precise computational cost. On x86 architectures, the `RDTSC` instruction or high-level wrappers (e.g., `std::chrono::steady_clock` with cycle calibration) are used.
Example using `<chrono>` to approximate cycles:

```cpp
#include <chrono>
#include <iostream>

int main() {
    constexpr long N = 10'000'000;
    auto start = std::chrono::steady_clock::now();

    volatile long sum = 0;
    for (long i = 0; i < N; ++i) sum += i;

    auto end = std::chrono::steady_clock::now();
    auto elapsed = std::chrono::duration<double>(end - start).count();

    std::cout << "Elapsed time: " << elapsed << " s\n";
}
```

Tips:

- `volatile` prevents the compiler from optimizing away loops.
- For precise CPU cycle counting on multiple platforms, consider **`<x86intrin.h>`** (`__rdtsc`) on x86.
- Always run **multiple iterations** to smooth out noise from context switches or thermal throttling.

2.3.4 Common Measurement Pitfalls

1. **Compiler Optimizations**:
    - Aggressive optimization may remove entire code sections (dead code elimination).
    - Mitigation: Use `volatile` or store results in variables used after the loop.
2. **Caching Effects**:
    - Memory hierarchy (L1/L2/L3 cache, RAM) dramatically affects runtime.
    - Solution: Repeat measurements, shuffle input, or warm up caches before measurement.
3. **Context Switching and Background Processes**:
    - Multitasking can skew results; run benchmarks in isolated environments.
4. **Small Sample Sizes**:
    - Measuring very fast algorithms may produce noise due to timer resolution.
    - Solution: Repeat the function many times and average results.

5. **Incorrect Use of Parallel Algorithms**:
    - Parallel execution policies introduce thread overhead; small arrays may perform worse than sequential.
    - Always profile for your dataset size.

2.3.5 Recommended Workflow for Reliable Benchmarks

1. **Choose the right clock**: `high_resolution_clock` or `steady_clock` for monotonic time measurement.
2. **Warm up caches**: Run the function once before timing.
3. **Repeat measurements**: Use loops or microbenchmark libraries (Google Benchmark) for averaging.
4. **Record system info**: CPU frequency, cores, OS, and compiler optimizations for reproducibility.
5. **Compare algorithm variants**: Sequential vs parallel, naive vs optimized, in-place vs extra memory.

Example: averaging multiple runs:

```
double measure(auto func, int runs = 10) {
    double total = 0.0;
    for (int i = 0; i < runs; ++i) {
        auto start = std::chrono::high_resolution_clock::now();
        func();
        auto end = std::chrono::high_resolution_clock::now();
        total += std::chrono::duration<double, std::milli>(end - start).count();
    }
```

```
    return total / runs;
}
```

2.3.6 Summary

- `<chrono>` provides high-precision timing for empirical performance evaluation.
- `std::execution` enables safe and concise parallelization of STL algorithms.
- CPU cycles offer low-level insight into algorithm cost but require careful handling.
- Awareness of pitfalls—compiler optimizations, caching, and background processes—is critical for reliable measurement.
- Combining **theoretical complexity analysis** with **practical benchmarks** equips C++ developers to make informed algorithmic choices in real-world applications.

Chapter 3

Essential C++ Tools for Algorithm Developers

3.1 The Standard Library Overview Relevant to Algorithms (Containers, Iterators, Algorithms Header)

A deep understanding of the **C++ Standard Library** is essential for algorithm developers. Modern C++ offers a rich ecosystem of **containers, iterators, and algorithmic functions** that allow developers to implement complex operations efficiently and safely. Leveraging these facilities enables higher productivity and clearer, more maintainable code while adhering to best practices.

3.1.1 Containers

Containers are data structures provided by the Standard Library that store collections of objects. Choosing the right container is critical for algorithm performance and complexity guarantees.

1.1 Sequence Containers

- **`std::vector`**
  - Dynamic array with contiguous memory.
  - O(1) random access; amortized O(1) insertion at the end.
  - Ideal for algorithms requiring indexing or contiguous memory access.

```
std::vector<int> v{1, 2, 3, 4, 5};
v.push_back(6);
```

- **`std::deque`**
  - Double-ended queue supporting O(1) insertion at both ends.
  - Slightly less cache-friendly than `vector` due to segmented memory.
- **`std::list`**
  - Doubly linked list with O(1) insertion/deletion anywhere, O(n) traversal.
  - Rarely used in modern C++ unless frequent mid-list modifications are necessary.
- **`std::array`**

  - Fixed-size array known at compile time.
  - Offers stack allocation and compile-time bounds checking.

```
std::array<int, 5> arr = {1, 2, 3, 4, 5};
```

1.2 Associative Containers

- Provide fast lookup, insertion, and deletion based on keys.
  - **`std::set` / `std::map`**
    - Balanced binary tree implementations.
    - O(log n) insert, search, delete.

```
std::set<int> s{1, 3, 5};
s.insert(2);
```

  - **`std::unordered_set` / `std::unordered_map`**
    - Hash-table-based containers.
    - O(1) average-case access; O(n) worst-case.
    - Useful for constant-time lookups.

1.3 Container Selection Guidelines

- **Random access and iteration** → `vector`, `array`.
- **Frequent insertions/deletions in the middle** → `list`.
- **Unique sorted elements** → `set` / `map`.
- **Fast average-case lookup** → `unordered_set` / `unordered_map`.

3.1.2 Iterators

Iterators are abstractions that allow algorithms to **access and traverse containers uniformly**, decoupling the algorithm from container type.

2.1 Iterator Categories

Category	Capabilities	Common Containers
InputIterator	Read-only, single-pass	`istream_iterator`
OutputIterator	Write-only, single-pass	`ostream_iterator`
ForwardIterator	Multi-pass, read/write	`forward_list`, `unordered_set`
BidirectionalIterator	Forward + backward traversal	`list`, `set`
RandomAccessIterator	Constant-time access by index	`vector`, `deque`, `array`

2.2 Example Usage

```
std::vector<int> v{1, 2, 3, 4, 5};
for (auto it = v.begin(); it != v.end(); ++it) {
    *it *= 2;
}
```

- Iterators provide a **uniform interface** for algorithms like `std::sort`, `std::find`, or `std::accumulate`, allowing them to operate on any compatible container.

3.1.3 The <algorithm> Header

The <algorithm> header provides a vast collection of **generic algorithms** that work with iterators. These algorithms are **highly optimized**, type-safe, and often parallelizable (C++17 onward with execution policies).

3.1 Categories of Algorithms

1. **Non-modifying sequence operations**
   - std::all_of, std::any_of, std::none_of
   - std::find, std::find_if
   - Example:

```
std::vector<int> v{1, 2, 3, 4};
if (std::any_of(v.begin(), v.end(), [](int x){ return x % 2 == 0; })) {
    std::cout << "Contains even number\n";
}
```

2. **Modifying sequence operations**
   - std::copy, std::transform, std::fill, std::remove
   - Example:

```
std::vector<int> v{1,2,3,4,5};
std::transform(v.begin(), v.end(), v.begin(), [](int x){ return x*x; });
```

3. **Sorting and related operations**
   - std::sort, std::stable_sort, std::partial_sort, std::nth_element

4. **Set operations** (require sorted ranges)
    - `std::set_union`, `std::set_intersection`
5. **Numeric operations**
    - `std::accumulate`, `std::inner_product`, `std::adjacent_difference`

3.2 Modern C++ Enhancements

- C++17: parallel execution policies (`std::execution::par`) for `std::for_each`, `std::sort`, `std::transform`.
- C++20: ranges library (`std::ranges`) allows cleaner algorithm expressions with pipelines and views:

```
#include <ranges>
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v{1,2,3,4,5};
    for (int x : v | std::ranges::views::transform([](int n){ return n*n; })) {
        std::cout << x << ' ';
    }
}
```

3.1.4 Best Practices for Algorithm Developers

1. Prefer **STL algorithms** over writing custom loops; they are well-tested and optimized.

2. Always pair algorithms with **appropriate iterators** to maximize flexibility.
3. Use **execution policies and ranges** in modern C++ to write clear and potentially parallel code.
4. Choose containers that match **algorithm access patterns** to avoid unnecessary complexity or overhead.

By mastering containers, iterators, and `<algorithm>` utilities, algorithm developers can implement high-performance, maintainable solutions while leveraging the **full power of Modern C++**.

3.2 Modern C++ Features That Change Algorithm Design: `ranges`, `concepts`, `span`, `string_view`

Modern C++ (C++17, C++20, and C++23) introduces a set of powerful features that **fundamentally change how algorithms are written, composed, and optimized**. These features improve **abstraction, safety, and performance** without sacrificing the fine-grained control that C++ developers expect.

3.2.1 Ranges (`std::ranges`)

Introduced in C++20, **ranges** provide a composable abstraction for sequences of elements. Unlike traditional iterator-based algorithms, ranges enable **pipeline-style operations** that are more readable and expressive.
Key advantages:

- Concise syntax: algorithms can be chained as pipelines.
- Lazy evaluation: operations like `filter` or `transform` are evaluated only when iterated.
- Improved safety: eliminates common iterator errors.

Example: filtering and transforming a vector

```
#include <ranges>
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v{1, 2, 3, 4, 5, 6};
```

```
    auto even_squares = v
        | std::ranges::views::filter([](int x){ return x % 2 == 0; })
        | std::ranges::views::transform([](int x){ return x * x; });

    for (int x : even_squares)
        std::cout << x << " ";  // Output: 4 16 36
}
```

Impact on algorithm design:

- Eliminates manual loops for transformations.
- Supports composable and modular algorithm pipelines.
- Encourages functional programming patterns in C++.

3.2.2 Concepts (`std::concepts`)

Concepts, introduced in C++20, provide **compile-time constraints** on template parameters, allowing algorithms to be **self-documenting and safer**.
Example: constraining a generic sum function

```
#include <concepts>
#include <vector>
#include <numeric>
#include <iostream>

template <std::integral T>
T sum_elements(const std::vector<T>& v) {
    return std::accumulate(v.begin(), v.end(), T{0});
}
```

```
int main() {
    std::vector<int> v{1, 2, 3};
    std::cout << sum_elements(v);  // Output: 6
}
```

Impact on algorithm design:

- Prevents incorrect template instantiations.
- Improves **compile-time diagnostics**, making errors easier to understand.
- Encourages **type-safe generic algorithms**.

3.2.3 `std::span`

`std::span` (C++20) represents a **non-owning view over a contiguous sequence** of elements. It allows algorithms to operate on arrays, vectors, or subarrays without copying data.

Example: passing a subarray to an algorithm

```
#include <span>
#include <vector>
#include <algorithm>
#include <iostream>

void double_elements(std::span<int> s) {
    for (int &x : s) x *= 2;
}

int main() {
    std::vector<int> v{1, 2, 3, 4, 5};
```

```
    double_elements(v);  // doubles all elements
    for (int x : v) std::cout << x << " ";  // Output: 2 4 6 8 10

    int arr[] = {10, 20, 30};
    double_elements(arr);  // works with C-style arrays
}
```

Impact on algorithm design:

- Algorithms can accept **generic contiguous sequences**.
- Reduces copying overhead and improves memory efficiency.
- Integrates seamlessly with STL algorithms.

3.2.4 `std::string_view`

`std::string_view` (C++17) is a **non-owning, lightweight view of a string**. It allows string algorithms to **work on substrings without copying**.
Example: splitting and searching substrings efficiently

```
#include <string_view>
#include <iostream>

void print_words(std::string_view sv) {
    size_t start = 0, end;
    while ((end = sv.find(' ', start)) != std::string_view::npos) {
        std::cout << sv.substr(start, end - start) << '\n';
        start = end + 1;
    }
    std::cout << sv.substr(start) << '\n';
}
```

```
int main() {
    std::string text = "Modern C++ algorithms are powerful";
    print_words(text);
}
```

Impact on algorithm design:

- Avoids unnecessary string copying.
- Allows safe, read-only access to substrings.
- Optimizes string-intensive algorithms like parsing, searching, and tokenization.

3.2.5 Combined Modern Patterns

Modern C++ features often **work together** to simplify algorithm design:

```
#include <ranges>
#include <string_view>
#include <vector>
#include <iostream>
#include <algorithm>

int main() {
    std::vector<std::string> words{"Modern", "C++", "algorithms", "are", "powerful"};

    auto filtered = words
        | std::ranges::views::filter([](const auto& w){ return w.size() > 3; })
        | std::ranges::views::transform([](std::string_view sv){ return sv.substr(0,
        ↪  3); });
```

```
    for (auto w : filtered)
        std::cout << w << " ";  // Output: Mod algo pow
}
```

- `std::string_view` avoids copying strings.
- `ranges` compose transformations and filters cleanly.
- Concepts (if applied in templates) can enforce constraints on element types.

3.2.6 Summary

Modern C++ features fundamentally change algorithm design by:

1. **Ranges**: Enabling composable, pipeline-style operations.
2. **Concepts**: Enforcing compile-time constraints for safer templates.
3. **Span**: Allowing generic, non-owning views of contiguous sequences.
4. **String_view**: Optimizing string handling and parsing algorithms.

By adopting these features, algorithm developers can write **concise, efficient, and type-safe algorithms** that leverage the full power of C++17, C++20, and beyond.

3.3 Unit Testing & Benchmarking in C++: GoogleTest, Catch2, `benchmark` Library, `valgrind`, Sanitizers

Reliable algorithms require **rigorous testing and performance evaluation**. Modern C++ developers rely on unit testing frameworks, benchmarking libraries, and runtime analysis tools to ensure correctness, efficiency, and safety. This section provides an overview of essential tools and best practices.

3.3.1 Unit Testing

Unit tests verify that individual components of an algorithm behave as expected. In C++, popular frameworks include **GoogleTest** and **Catch2**.

1.1 GoogleTest (gtest)

GoogleTest is widely adopted for enterprise-grade C++ projects. Key features:

- **Assertions**: `EXPECT_EQ`, `ASSERT_TRUE`, `EXPECT_NEAR` for floating-point comparisons.
- **Test Fixtures**: Reusable setup/teardown for complex objects.
- **Parameterized Tests**: Test algorithms with multiple inputs systematically.

Example: Testing a sorting algorithm

```
#include <gtest/gtest.h>
#include <vector>
#include <algorithm>
```

```
std::vector<int> sort_vector(std::vector<int> v) {
    std::sort(v.begin(), v.end());
    return v;
}

TEST(SortTest, HandlesUnsortedInput) {
    std::vector<int> input{4, 2, 5, 1};
    std::vector<int> expected{1, 2, 4, 5};
    EXPECT_EQ(sort_vector(input), expected);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

1.2 Catch2

Catch2 is a **header-only testing library** suitable for lightweight projects.
Advantages:

- No separate build required for the framework.
- Simple syntax for assertions: `REQUIRE`, `CHECK`.
- Supports BDD-style testing with `SCENARIO` and `GIVEN/WHEN/THEN`.

Example:

```
#define CATCH_CONFIG_MAIN
#include <catch2/catch.hpp>
#include <vector>
#include <algorithm>
```

```
std::vector<int> sort_vector(std::vector<int> v) {
    std::sort(v.begin(), v.end());
    return v;
}

TEST_CASE("SortVector correctly sorts integers") {
    std::vector<int> input{4, 2, 5, 1};
    std::vector<int> expected{1, 2, 4, 5};
    REQUIRE(sort_vector(input) == expected);
}
```

Best practices:

- Unit tests should be **small, isolated, and deterministic**.
- Use fixtures to reduce code duplication for shared data.
- Integrate with **CI/CD pipelines** to ensure automated testing.

3.3.2 Benchmarking

Measuring performance is critical to validate algorithm efficiency beyond asymptotic complexity.

2.1 Google Benchmark Library

The **benchmark library** by Google allows fine-grained measurement of function execution time.

Example: Benchmarking vector sorting

```
#include <benchmark/benchmark.h>
#include <vector>
```

```cpp
#include <algorithm>
#include <numeric>

static void BM_SortVector(benchmark::State& state) {
    std::vector<int> v(state.range(0));
    std::iota(v.begin(), v.end(), 0);

    for (auto _ : state) {
        std::shuffle(v.begin(), v.end(), std::mt19937{std::random_device{}()});
        benchmark::DoNotOptimize(v);
        std::sort(v.begin(), v.end());
    }
}

BENCHMARK(BM_SortVector)->Range(8, 1<<20);
BENCHMARK_MAIN();
```

- Supports **parameterized benchmarks** and **reporting average, min, max, and standard deviation**.
- Provides **DoNotOptimize** to prevent compiler optimizations from invalidating results.

2.2 Manual Microbenchmarking

For small-scale experiments, `<chrono>` can be used as described in earlier chapters, but libraries like Google Benchmark provide **repeatability, statistical significance, and integration**.

3.3.3 Memory and Runtime Analysis

Ensuring algorithms are memory-safe and free from undefined behavior is as important as correctness.

3.1 Valgrind

- Popular on Linux for **memory leak detection** and **undefined behavior analysis**.
- Typical usage:

```
valgrind --leak-check=full ./my_program
```

- Reports allocations that were not freed, helping detect **memory leaks** in complex algorithmic implementations.

3.2 Sanitizers

Modern compilers (Clang/GCC) provide runtime sanitizers:

- **AddressSanitizer (ASan)**: Detects buffer overflows, use-after-free, and invalid memory access.
- **UndefinedBehaviorSanitizer (UBSan)**: Catches undefined behavior such as signed integer overflow or invalid type casting.
- **ThreadSanitizer (TSan)**: Detects data races in concurrent algorithms.

Example: Compile and run with AddressSanitizer:

```
g++ -fsanitize=address -fno-omit-frame-pointer -O1 main.cpp -o main
./main
```

Sanitizers are particularly valuable when benchmarking **parallel algorithms** or working with **manual memory management**.

3.3.4 Integrating Testing and Benchmarking

- Combine **unit tests** (GoogleTest/Catch2) with **benchmarking** to verify correctness **and** performance.
- Use **sanitizers** during development to catch memory or concurrency errors before production.
- Structure your project to **separate algorithm implementations from test and benchmark code**, enabling maintainable and reproducible results.

Project Layout Example:

```
/src       -> algorithm implementations
/tests     -> unit tests (gtest or Catch2)
/bench     -> benchmarks (Google Benchmark)
/CMakeLists.txt -> separate targets for building tests and benchmarks
```

3.3.5 Key Takeaways

- Unit testing ensures **algorithm correctness** and prevents regressions.
- Benchmarking verifies **performance expectations** against theoretical complexity.
- Tools like **Valgrind and sanitizers** catch memory errors and undefined behavior early.
- Modern C++ testing and benchmarking libraries are **highly integrated** with contemporary workflows, making them essential for algorithm developers.

By combining these tools, developers can produce **robust, efficient, and maintainable algorithms**, ready for real-world applications.

Part II

Linear & Basic Structures (with C++ implementations)

Chapter 4

Arrays & Vectors

4.1 Static Array vs `std::vector` — Memory and Performance Tradeoffs

In C++, both **static arrays** and **`std::vector`** are foundational linear containers. Choosing between them requires understanding **memory layout, performance characteristics, and flexibility tradeoffs**. This section provides a detailed comparison, highlighting when each container is appropriate for algorithm implementation.

4.1.1 Static Arrays

A static array is a **fixed-size, contiguous block of memory** allocated either on the stack or as a global/static variable.

Syntax example:

```
int arr[5] = {1, 2, 3, 4, 5};
```

Characteristics:

- **Fixed size**: Must be known at compile time (for stack allocation).
- **Contiguous memory**: Elements are stored sequentially, optimizing cache usage.
- **No overhead**: Minimal metadata; access and iteration are very fast.
- **Stack vs heap allocation**:
  - Stack arrays: fast allocation/deallocation, limited by stack size (~1MB typical).
  - Heap arrays (dynamic allocation): `new int[N]` allows runtime sizing but requires manual memory management.

Performance considerations:

- **Access speed**: O(1) due to contiguous memory and no bounds checking.
- **Insertion/deletion**: Expensive for elements in the middle; requires manual shifting.
- **Memory footprint**: Minimal; no extra allocation overhead.

Example: Sum of static array elements

```
int sum = 0;
for (int i = 0; i < 5; ++i) sum += arr[i];
```

Limitations:

- Fixed size prevents dynamic growth.
- No built-in resizing or automatic memory management.
- Unsafe if size is exceeded (undefined behavior).

4.1.2 `std::vector`

`std::vector` is a **dynamic array container** from the C++ Standard Library that **manages memory automatically**.
Syntax example:

```
#include <vector>
std::vector<int> v{1, 2, 3, 4, 5};
v.push_back(6);  // dynamically resizes if needed
```

Characteristics:

- **Dynamic resizing**: Automatically grows as elements are added.
- **Contiguous memory**: Like arrays, elements are sequentially stored, which benefits cache locality.
- **Automatic memory management**: Allocates and deallocates memory on the heap.
- **Rich interface**: Provides iterators, size/empty queries, and STL-compatible algorithms.

Performance considerations:

- **Access speed**: O(1) random access, similar to static arrays.

- **Insertion/deletion at end**: Amortized O(1) thanks to exponential capacity growth.
- **Insertion/deletion elsewhere**: O(n), as elements may need to shift.
- **Memory overhead**: Extra metadata (size, capacity) and potential over-allocation to optimize resizing.

Example: Dynamic resizing impact

```
std::vector<int> v;
for (int i = 0; i < 1000; ++i) v.push_back(i);  // vector automatically resizes
```

- Each resize may allocate a new block (usually 2× current capacity) and copy elements.
- Amortized cost remains O(1) for `push_back`, but worst-case for individual insertions is O(n).

4.1.3 Memory Layout and Cache Effects

Feature	Static Array	`std::vector`
Memory location	Stack (fast) / Heap (manual)	Heap (automatic)
Contiguous	Yes	Yes
Metadata	None	Stores size, capacity, allocator pointer

Feature	Static Array	`std::vector`
Cache friendliness	Excellent	Excellent (mostly)
Overhead	Minimal	Slight overhead for capacity management
Resizing	Not supported	Supported, may trigger reallocations

Cache considerations:

- Both containers benefit from **sequential access**, optimizing prefetching.
- Excessive `push_back` causing repeated reallocation can introduce **cache misses**.

4.1.4 Performance Tradeoffs

1. **Static arrays**:
    - Best for **fixed-size, high-performance scenarios** where memory overhead must be minimal.
    - Example: low-level numerical algorithms, embedded systems.
2. `std::vector`:
    - Best for **dynamic-size scenarios**, generic programming, or when using STL algorithms.
    - Example: data structures requiring flexible growth, such as queues, stacks, or graphs.

Microbenchmark insight:

- Iterating over static arrays is slightly faster than `std::vector` due to the absence of metadata.
- For large, dynamically changing datasets, `std::vector` often outperforms manual heap arrays due to **automated allocation strategies** and **exception safety**.

4.1.5 Guidelines for Algorithm Developers

- **Use static arrays** when:
  - Size is known at compile time.
  - Maximum performance and minimal memory overhead are critical.
  - Avoiding dynamic memory allocation is important (e.g., embedded systems).
- **Use `std::vector`** when:
  - Dataset size varies at runtime.
  - STL algorithms or iterator-based generic code are desired.
  - Safety and maintainability are more important than the tiny static overhead.
- **Hybrid approach**:
  - Use `std::array` (fixed-size STL array, stack-allocated) for known sizes.
  - Use `std::vector` when dynamic growth or integration with STL algorithms is required.

4.1.6 Summary

- **Static arrays**: minimal overhead, fixed size, optimal cache performance.
- **`std::vector`**: dynamic, safe, STL-compatible, slightly more overhead but far more flexible.
- Choosing the right container is a **tradeoff between memory efficiency, performance, and flexibility**.
- Understanding these tradeoffs is essential when implementing linear structures or algorithms in Modern C++.

4.2 In-Place Algorithms: Sliding Window, Two Pointers, Partitioning in C++

In-place algorithms are techniques that **manipulate arrays or vectors without allocating significant extra memory**, making them ideal for high-performance applications. Common in algorithmic problem solving, these methods **maximize efficiency** by reducing memory usage while maintaining linear or near-linear time complexity. This section examines **sliding window**, **two-pointer**, and **partitioning** techniques in C++ with practical examples.

4.2.1 Sliding Window Technique

The sliding window approach is used to **process a subset of elements in a contiguous sequence efficiently**. It is widely applied in **sum, maximum/minimum, and substring problems.**

Core idea: Maintain a window of elements, updating it incrementally rather than recomputing values repeatedly.

Example: Maximum Sum of a Subarray of Size k

```
#include <vector>
#include <iostream>

int maxSubarraySum(const std::vector<int>& v, int k) {
    int n = v.size();
    if (n < k) return -1;
    int sum = 0;
    for (int i = 0; i < k; ++i) sum += v[i];

    int maxSum = sum;
```

```
    for (int i = k; i < n; ++i) {
        sum += v[i] - v[i - k];  // slide window forward
        if (sum > maxSum) maxSum = sum;
    }
    return maxSum;
}

int main() {
    std::vector<int> v{1, 2, 3, 4, 5, 6};
    std::cout << "Max sum of size 3: " << maxSubarraySum(v, 3) << "\n";  // Output:
    ↪ 15
}
```

Key points:

- Time complexity: O(n)
- Space complexity: O(1)
- Efficient for large arrays where recomputation is expensive

Applications: maximum sum subarrays, minimum window substring, dynamic range queries.

4.2.2 Two-Pointer Technique

The two-pointer method is commonly used for **sorted arrays** or sequences to **find pairs, triplets, or subarrays satisfying a condition**.
Core idea: Use two indices to traverse the array from different directions.
Example: Finding a pair with a given sum in a sorted vector

```cpp
#include <vector>
#include <iostream>
#include <algorithm>

bool hasPairWithSum(std::vector<int>& v, int target) {
    std::sort(v.begin(), v.end());  // ensure sorted
    int left = 0, right = v.size() - 1;

    while (left < right) {
        int sum = v[left] + v[right];
        if (sum == target) return true;
        else if (sum < target) ++left;
        else --right;
    }
    return false;
}

int main() {
    std::vector<int> v{2, 4, 3, 5, 7};
    std::cout << std::boolalpha << hasPairWithSum(v, 10) << "\n";  // Output: true
}
```

Key points:

- Time complexity: O(n) for sorted arrays (sorting O(n log n) if needed)
- Space complexity: O(1)
- Can be extended to **triplets or sliding windows** in subarray problems

Applications: two-sum, three-sum, moving average, subarray problems, interval overlap detection.

4.2.3 Partitioning (In-Place Reordering)

Partitioning rearranges elements in an array based on a **pivot or condition**. It is foundational in **quick sort, Dutch National Flag problem, and other in-place algorithms**.

3.1 Lomuto Partition Scheme (for quicksort)

```
#include <vector>
#include <iostream>
#include <algorithm>

int lomutoPartition(std::vector<int>& v, int low, int high) {
    int pivot = v[high];
    int i = low - 1;

    for (int j = low; j < high; ++j) {
        if (v[j] <= pivot) {
            ++i;
            std::swap(v[i], v[j]);
        }
    }
    std::swap(v[i + 1], v[high]);
    return i + 1;
}

void quickSort(std::vector<int>& v, int low, int high) {
    if (low < high) {
        int pi = lomutoPartition(v, low, high);
        quickSort(v, low, pi - 1);
        quickSort(v, pi + 1, high);
    }
}
```

```
int main() {
    std::vector<int> v{10, 7, 8, 9, 1, 5};
    quickSort(v, 0, v.size() - 1);
    for (int x : v) std::cout << x << " ";  // Output: 1 5 7 8 9 10
}
```

3.2 Dutch National Flag Problem (Three-Way Partitioning)

- Useful when **categorizing array elements** into three groups (e.g., 0, 1, 2).

```
#include <vector>
#include <iostream>

void dutchNationalFlag(std::vector<int>& v) {
    int low = 0, mid = 0, high = v.size() - 1;

    while (mid <= high) {
        if (v[mid] == 0) std::swap(v[low++], v[mid++]);
        else if (v[mid] == 1) ++mid;
        else std::swap(v[mid], v[high--]);
    }
}

int main() {
    std::vector<int> v{2, 0, 2, 1, 1, 0};
    dutchNationalFlag(v);
    for (int x : v) std::cout << x << " ";  // Output: 0 0 1 1 2 2
}
```

Key points:

- Time complexity: O(n)

- Space complexity: O(1)
- Efficient for in-place sorting or categorization problems

4.2.4 Best Practices for In-Place Algorithms

1. **Minimize extra memory allocation**: Use in-place swaps and indices instead of auxiliary arrays.
2. **Validate bounds**: Off-by-one errors are common; always check index boundaries.
3. **Combine techniques**: Sliding window + two-pointers or partitioning can solve complex problems efficiently.
4. **Leverage STL utilities**: Functions like `std::partition`, `std::rotate`, and `std::stable_partition` simplify in-place operations while remaining efficient.

Example using STL partition:

```
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    std::vector<int> v{1, 4, 2, 5, 3};
    std::partition(v.begin(), v.end(), [](int x){ return x % 2 == 0; });
    for (int x : v) std::cout << x << " ";  // Output: 4 2 1 5 3 (evens first)
}
```

4.2.5 Summary

- **Sliding window**: Optimizes range-based computations with O(1) space.

- **Two pointers**: Efficiently processes sorted sequences or paired conditions.
- **Partitioning**: Reorders arrays in-place, foundational for quicksort and classification problems.
- **In-place design**: Reduces memory usage, improves cache locality, and often increases performance for large datasets.

Understanding and applying these **in-place techniques** equips developers to implement high-performance algorithms on arrays and vectors with minimal memory overhead.

4.3 Exercises: In-Place Rotation, Subarray Sums, Prefix/Suffix Arrays

Practical exercises solidify the understanding of arrays and vectors while reinforcing in-place algorithm techniques. This section presents **hands-on C++ exercises** covering **rotation, subarray sums, and prefix/suffix arrays**, including hints, explanations, and sample implementations.

4.3.1 In-Place Array Rotation

Problem: Rotate an array of size `n` by `k` positions to the right **in-place**.
Example:
Input: `[1, 2, 3, 4, 5]`, `k = 2` → Output: `[4, 5, 1, 2, 3]`
Solution Concept:

- Reverse the whole array.
- Reverse the first `k` elements.
- Reverse the remaining `n-k` elements.

C++ Implementation:

```cpp
#include <vector>
#include <algorithm>
#include <iostream>

void rotateArray(std::vector<int>& v, int k) {
    int n = v.size();
    k %= n;  // handle k > n
    std::reverse(v.begin(), v.end());
```

```
    std::reverse(v.begin(), v.begin() + k);
    std::reverse(v.begin() + k, v.end());
}

int main() {
    std::vector<int> v{1, 2, 3, 4, 5};
    rotateArray(v, 2);
    for (int x : v) std::cout << x << " ";  // Output: 4 5 1 2 3
}
```

Key Takeaways:

- Time complexity: O(n)
- Space complexity: O(1) (in-place)
- Reversing subarrays is a common in-place technique.

4.3.2 Subarray Sums

Problem: Compute the sum of all contiguous subarrays efficiently.
Example:
Input: `[1, 2, 3]` → Subarray sums: `[1, 3, 6, 2, 5, 3]`
C++ Implementation (Naive):

```
#include <vector>
#include <iostream>

void allSubarraySums(const std::vector<int>& v) {
    int n = v.size();
    for (int start = 0; start < n; ++start) {
        int sum = 0;
```

```
        for (int end = start; end < n; ++end) {
            sum += v[end];
            std::cout << sum << " ";
        }
    }
}

int main() {
    std::vector<int> v{1, 2, 3};
    allSubarraySums(v);  // Output: 1 3 6 2 5 3
}
```

Optimized Approach Using Prefix Sum:

```
#include <vector>
#include <iostream>

std::vector<int> computePrefixSum(const std::vector<int>& v) {
    std::vector<int> prefix(v.size() + 1, 0);
    for (size_t i = 0; i < v.size(); ++i) prefix[i + 1] = prefix[i] + v[i];
    return prefix;
}

int main() {
    std::vector<int> v{1, 2, 3};
    auto prefix = computePrefixSum(v);

    for (size_t start = 0; start < v.size(); ++start) {
        for (size_t end = start; end < v.size(); ++end) {
            int sum = prefix[end + 1] - prefix[start];
            std::cout << sum << " ";  // Output: 1 3 6 2 5 3
        }
```

```
    }
}
```

Key Takeaways:

- Prefix sums reduce repeated computation of subarray sums.
- Time complexity: $O(n^2)$ for all subarrays, $O(n)$ to compute prefix sum.
- Space complexity: $O(n)$ for prefix array.

4.3.3 Prefix and Suffix Arrays

Prefix and suffix arrays are widely used to **accelerate cumulative calculations** and **range queries**.
Problem: Compute **prefix and suffix sums** of an array.
C++ Implementation:

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v{1, 2, 3, 4, 5};
    int n = v.size();

    std::vector<int> prefix(n), suffix(n);

    prefix[0] = v[0];
    for (int i = 1; i < n; ++i) prefix[i] = prefix[i - 1] + v[i];

    suffix[n - 1] = v[n - 1];
    for (int i = n - 2; i >= 0; --i) suffix[i] = suffix[i + 1] + v[i];
```

```
    std::cout << "Prefix sums: ";
    for (int x : prefix) std::cout << x << " ";  // Output: 1 3 6 10 15

    std::cout << "\nSuffix sums: ";
    for (int x : suffix) std::cout << x << " ";  // Output: 15 14 12 9 5
}
```

Applications:

- Range sum queries
- Sliding window problems
- Dynamic programming optimizations

Key Takeaways:

- Prefix and suffix arrays allow **O(1) range sum queries** after O(n) preprocessing.
- Crucial for algorithm optimization in competitive programming and real-world datasets.

4.3.4 Suggested Exercises

1. **Rotate array left by `k` positions in-place**.
2. **Find the maximum subarray sum of size `k`** using sliding window technique.
3. **Compute prefix product array** (similar to prefix sum but with multiplication).

4. **Range query exercise**: Given a prefix array, compute sums of multiple ranges efficiently.

5. **In-place rotation of subarray**: Rotate a portion of the array without additional memory.

4.3.5 Summary

- **In-place rotation** teaches efficient index manipulation and array reversal techniques.
- **Subarray sums** introduce naive and prefix-sum optimizations, highlighting tradeoffs between computation and memory.
- **Prefix and suffix arrays** provide a foundation for cumulative computations and range queries, widely used in advanced algorithms.
- Practicing these exercises strengthens **algorithmic thinking** while reinforcing **memory-efficient, in-place C++ programming**.

Chapter 5

Linked Lists

5.1 Single/Doubly Linked List Implementations in Modern C++ (Smart Pointers vs Raw Pointers)

Linked lists are **fundamental dynamic data structures** that support efficient insertion and deletion operations, especially when compared to contiguous containers like arrays or vectors. Modern C++ provides tools to implement linked lists safely and efficiently using **raw pointers** or **smart pointers**. This section explores both approaches, highlighting trade-offs and best practices.

5.1.1 Singly Linked List

A **singly linked list** consists of nodes where each node contains **data** and a **pointer to the next node**.

Node structure with raw pointers:

```
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};
```

Basic operations using raw pointers:

```
#include <iostream>

struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

// Insert at head
void insertHead(Node*& head, int val) {
    Node* newNode = new Node(val);
    newNode->next = head;
    head = newNode;
}

// Print list
void printList(Node* head) {
    while (head) {
        std::cout << head->data << " ";
        head = head->next;
    }
}

int main() {
```

```
    Node* head = nullptr;
    insertHead(head, 3);
    insertHead(head, 2);
    insertHead(head, 1);
    printList(head);  // Output: 1 2 3
}
```

Considerations with raw pointers:

- Manual memory management (`delete`) is required to avoid leaks.
- Error-prone in complex algorithms or exception-prone code.
- High performance, minimal overhead.

Singly linked list with `std::unique_ptr` (smart pointers):
Modern C++ encourages **automatic memory management** using smart pointers, especially `std::unique_ptr` for exclusive ownership.

```
#include <memory>
#include <iostream>

struct Node {
    int data;
    std::unique_ptr<Node> next;
    Node(int val) : data(val), next(nullptr) {}
};

void insertHead(std::unique_ptr<Node>& head, int val) {
    auto newNode = std::make_unique<Node>(val);
    newNode->next = std::move(head);
    head = std::move(newNode);
```

```
}

void printList(const std::unique_ptr<Node>& head) {
    Node* current = head.get();
    while (current) {
        std::cout << current->data << " ";
        current = current->next.get();
    }
}

int main() {
    std::unique_ptr<Node> head = nullptr;
    insertHead(head, 3);
    insertHead(head, 2);
    insertHead(head, 1);
    printList(head);  // Output: 1 2 3
}
```

Advantages of smart pointers:

- Automatic cleanup, eliminates memory leaks.
- Exception-safe, avoids dangling pointers.
- `std::unique_ptr` enforces **single ownership**, aligning with RAII principles.

5.1.2 Doubly Linked List

A **doubly linked list (DLL)** allows traversal in both directions. Each node contains **data**, a **pointer to the next node**, and a **pointer to the previous node**.
DLL with raw pointers:

```
struct DNode {
    int data;
    DNode* next;
    DNode* prev;
    DNode(int val) : data(val), next(nullptr), prev(nullptr) {}
};

void insertHead(DNode*& head, int val) {
    DNode* newNode = new DNode(val);
    newNode->next = head;
    if (head) head->prev = newNode;
    head = newNode;
}
```

Key operations: insertion, deletion, and bidirectional traversal are slightly more complex due to the extra `prev` pointer.

DLL with smart pointers (`std::unique_ptr` + raw `prev` pointer):

Smart pointers simplify memory management but require care with **backward links**. A common pattern is:

- Use `std::unique_ptr` for `next` (exclusive ownership).
- Use raw or `std::weak_ptr` for `prev` to avoid **cyclic references**.

```
#include <memory>
#include <iostream>

struct DNode {
    int data;
    std::unique_ptr<DNode> next;
    DNode* prev;
```

```
    DNode(int val) : data(val), next(nullptr), prev(nullptr) {}
};

void insertHead(std::unique_ptr<DNode>& head, int val) {
    auto newNode = std::make_unique<DNode>(val);
    newNode->next = std::move(head);
    if (newNode->next) newNode->next->prev = newNode.get();
    head = std::move(newNode);
}

void printList(const std::unique_ptr<DNode>& head) {
    DNode* current = head.get();
    while (current) {
        std::cout << current->data << " ";
        current = current->next.get();
    }
}

int main() {
    std::unique_ptr<DNode> head = nullptr;
    insertHead(head, 3);
    insertHead(head, 2);
    insertHead(head, 1);
    printList(head);  // Output: 1 2 3
}
```

Key points:

- Using `unique_ptr` for `next` ensures safe automatic deallocation.
- `prev` must remain a raw pointer to avoid circular ownership and memory leaks.
- Doubly linked lists require careful handling during insertions and deletions.

5.1.3 Raw Pointers vs Smart Pointers — Tradeoffs

Feature	**Raw Pointers**	**Smart Pointers (`unique_ptr`)**
Memory safety	Manual, error-prone	Automatic, RAII-compliant
Performance	Minimal overhead	Slight overhead for ownership semantics
Exception safety	Low	High
Ownership	Programmer-managed	Enforced by `unique_ptr`
Cyclic references	Easy	Must avoid cycles (use `weak_ptr`)
Use cases	Low-level/high-performance	Most modern C++ code, safer algorithms

Best Practices:

- Prefer **smart pointers** for most modern C++ code.
- Use raw pointers **only when ownership is explicit or performance-critical**.
- Always handle **edge cases** (empty list, single-node list) in both implementations.
- When designing library-like structures, consider exposing **iterators** rather than raw node pointers.

5.1.4 Summary

- **Singly linked lists**: simpler, unidirectional, easy to implement with smart pointers.
- **Doubly linked lists**: support bidirectional traversal, require careful handling of `prev` pointers.
- **Raw pointers**: minimal overhead but manual memory management is error-prone.
- **Smart pointers**: automatic memory management, exception-safe, recommended in modern C++.
- Understanding these trade-offs is critical for **high-performance and safe algorithm design**.

5.2 Common Algorithms: Reverse, Detect Cycle (Floyd), Merge Lists, Remove Nth Node from End

Linked lists are a versatile linear data structure, and mastering **common algorithms** is essential for efficient manipulation and problem solving. This section covers **in-place reversal, cycle detection using Floyd's algorithm, merging sorted lists, and removing the N-th node from the end** in modern C++, with practical examples and performance analysis.

5.2.1 Reversing a Singly Linked List

Problem: Reverse the order of nodes in a singly linked list **in-place**.
Algorithm: Iterate through the list, reversing `next` pointers one by one while maintaining previous and current pointers.

```
#include <memory>
#include <iostream>

struct Node {
    int data;
    std::unique_ptr<Node> next;
    Node(int val) : data(val), next(nullptr) {}
};

std::unique_ptr<Node> reverseList(std::unique_ptr<Node> head) {
    std::unique_ptr<Node> prev = nullptr;
    while (head) {
        std::unique_ptr<Node> next = std::move(head->next);
```

```
        head->next = std::move(prev);
        prev = std::move(head);
        head = std::move(next);
    }
    return prev;
}

void printList(const std::unique_ptr<Node>& head) {
    Node* current = head.get();
    while (current) {
        std::cout << current->data << " ";
        current = current->next.get();
    }
}

int main() {
    auto head = std::make_unique<Node>(1);
    head->next = std::make_unique<Node>(2);
    head->next->next = std::make_unique<Node>(3);

    head = reverseList(std::move(head));
    printList(head);  // Output: 3 2 1
}
```

Time Complexity: O(n)
Space Complexity: O(1) — in-place operation
Applications: Undo operations, stack emulation, algorithmic challenges.

5.2.2 Cycle Detection (Floyd's Tortoise and Hare Algorithm)

Problem: Detect if a singly linked list contains a **cycle**.
Algorithm:

- Use two pointers, `slow` and `fast`.
- `slow` moves one step at a time, `fast` moves two steps.
- If `slow` and `fast` meet, a cycle exists.

```
struct NodeRaw {
    int data;
    NodeRaw* next;
    NodeRaw(int val) : data(val), next(nullptr) {}
};

bool hasCycle(NodeRaw* head) {
    NodeRaw* slow = head;
    NodeRaw* fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return true;
    }
    return false;
}

int main() {
    NodeRaw* a = new NodeRaw(1);
    NodeRaw* b = new NodeRaw(2);
    NodeRaw* c = new NodeRaw(3);
    a->next = b; b->next = c; c->next = a;  // cycle

    std::cout << std::boolalpha << hasCycle(a) << "\n";  // Output: true
}
```

Time Complexity: O(n)

Space Complexity: O(1)
Applications: Detect loops in linked lists, memory leak detection, graph traversal analogues.

5.2.3 Merging Two Sorted Linked Lists

Problem: Merge two sorted singly linked lists into a single sorted list.
Algorithm: Iteratively compare nodes of both lists and append the smaller node to the merged list.

```
#include <memory>
#include <iostream>

struct Node {
    int data;
    std::unique_ptr<Node> next;
    Node(int val) : data(val), next(nullptr) {}
};

std::unique_ptr<Node> mergeSortedLists(std::unique_ptr<Node> l1,
↪  std::unique_ptr<Node> l2) {
    auto dummy = std::make_unique<Node>(0);
    Node* tail = dummy.get();

    while (l1 && l2) {
        if (l1->data < l2->data) {
            tail->next = std::move(l1);
            tail = tail->next.get();
            l1 = std::move(tail->next);
        } else {
            tail->next = std::move(l2);
            tail = tail->next.get();
```

```
            l2 = std::move(tail->next);
        }
    }
    tail->next = l1 ? std::move(l1) : std::move(l2);
    return std::move(dummy->next);
}
```

Time Complexity: O(n + m) — linear in total number of nodes
Space Complexity: O(1) — in-place node re-linking
Applications: Merge sort on linked lists, priority queues, data stream merging.

5.2.4 Removing the N-th Node from the End

Problem: Remove the N-th node from the end of a singly linked list in a **single pass**.
Algorithm:

- Use two pointers separated by `n` nodes.
- Move both pointers together until the fast pointer reaches the end.
- Remove the target node using pointer manipulation.

```
struct NodeRaw {
    int data;
    NodeRaw* next;
    NodeRaw(int val) : data(val), next(nullptr) {}
};

NodeRaw* removeNthFromEnd(NodeRaw* head, int n) {
    NodeRaw dummy(0);
    dummy.next = head;
```

```
    NodeRaw* first = &dummy;
    NodeRaw* second = &dummy;

    for (int i = 0; i <= n; ++i) first = first->next;

    while (first) {
        first = first->next;
        second = second->next;
    }

    NodeRaw* toDelete = second->next;
    second->next = second->next->next;
    delete toDelete;
    return dummy.next;
}

int main() {
    NodeRaw* head = new NodeRaw(1);
    head->next = new NodeRaw(2);
    head->next->next = new NodeRaw(3);
    head->next->next->next = new NodeRaw(4);
    head = removeNthFromEnd(head, 2);

    for (NodeRaw* cur = head; cur; cur = cur->next)
        std::cout << cur->data << " ";
    // Output: 1 2 4

}
```

Time Complexity: O(n)

Space Complexity: O(1) — single-pass, constant extra memory

Applications: Linked list manipulations in real-time systems, competitive

programming.

5.2.5 Summary of Common Linked List Algorithms

Algorithm	Time Complexity	Space Complexity	Notes
Reverse List	O(n)	O(1)	In-place reversal, iterative or recursive variants
Detect Cycle (Floyd)	O(n)	O(1)	Tortoise and Hare algorithm, fast/slow pointers
Merge Sorted Lists	O(n + m)	O(1)	Efficient merging by re-linking nodes, in-place
Remove N-th Node	O(n)	O(1)	Single-pass with two-pointer technique

Key Takeaways:

- Many linked list operations can be implemented **in-place** to optimize memory usage.
- Two-pointer techniques recur in multiple algorithms, from reversal to cycle detection and deletion.
- Using **smart pointers** can ensure **safe memory management**, while raw pointers may be preferred for **performance-critical code**.

5.3 Exercises and Tests: Memory-Leak Free Implementations, Iterator Support

Building robust linked list implementations in modern C++ requires not only correctness but also **memory safety** and **ease of traversal**. This section provides exercises and guidance to ensure **memory-leak free designs** using smart pointers and implementing **iterators** for standard-compliant traversal.

5.3.1 Memory-Leak Free Implementations

Memory management is a critical concern with linked lists. Using **raw pointers** without careful deletion can easily lead to **memory leaks** or **dangling pointers**. Modern C++ provides **smart pointers (`std::unique_ptr`)** to automatically manage ownership and cleanup.

Exercise 1: Implement a singly linked list using `std::unique_ptr`
Requirements:

- Implement `push_front`, `push_back`, `pop_front`, and `pop_back` operations.
- Ensure no manual `delete` is needed.
- Write tests to verify memory is released when nodes are removed.

```
#include <memory>
#include <iostream>

struct Node {
    int data;
    std::unique_ptr<Node> next;
```

```
    Node(int val) : data(val), next(nullptr) {}
};

class SinglyLinkedList {
    std::unique_ptr<Node> head;
public:
    void push_front(int val) {
        auto newNode = std::make_unique<Node>(val);
        newNode->next = std::move(head);
        head = std::move(newNode);
    }

    void pop_front() {
        if (head) head = std::move(head->next);
    }

    void print() const {
        Node* current = head.get();
        while (current) {
            std::cout << current->data << " ";
            current = current->next.get();
        }
        std::cout << "\n";
    }
};

int main() {
    SinglyLinkedList list;
    list.push_front(10);
    list.push_front(20);
    list.push_front(30);
    list.print();  // Output: 30 20 10
```

```
    list.pop_front();
    list.print();  // Output: 20 10
}
```

Key Points:

- No manual `delete` is required.
- Memory automatically deallocates when `head` goes out of scope.
- Using `unique_ptr` ensures exception safety.

5.3.2 Iterator Support

Providing **iterator support** allows linked lists to integrate with **range-based for loops** and standard algorithms (`std::for_each`, `std::find_if`).

Exercise 2: Add iterator support to a singly linked list

```
#include <memory>
#include <iostream>

struct Node {
    int data;
    std::unique_ptr<Node> next;
    Node(int val) : data(val), next(nullptr) {}
};

class SinglyLinkedList {
    std::unique_ptr<Node> head;
public:
    void push_front(int val) {
```

```
        auto newNode = std::make_unique<Node>(val);
        newNode->next = std::move(head);
        head = std::move(newNode);
    }

    struct Iterator {
        Node* current;
        Iterator(Node* node) : current(node) {}
        int& operator*() { return current->data; }
        Iterator& operator++() { current = current->next.get();
        return *this;
    }
        bool operator!=(const Iterator& other) const { return current !=
        ↪  other.current; }
    };

    Iterator begin() { return Iterator(head.get()); }
    Iterator end() { return Iterator(nullptr); }
};

int main() {
    SinglyLinkedList list;
    list.push_front(10);
    list.push_front(20);
    list.push_front(30);

    for (int val : list) std::cout << val << " ";

    // Output: 30 20 10
}
```

Benefits of Iterator Support:

- Seamless integration with **C++ standard algorithms**.
- Enables **range-based for loops**.
- Improves **code readability and maintainability**.

5.3.3 Testing Linked Lists

Exercise 3: Unit testing linked list operations

- Use **Catch2 or GoogleTest** for verifying correctness.
- Include tests for:
    1. `push_front` and `pop_front`
    2. Iterator traversal
    3. Memory safety (tools like `valgrind` or sanitizers)

Example GoogleTest Skeleton:

```
#include <gtest/gtest.h>
#include "SinglyLinkedList.h"

TEST(SinglyLinkedListTest, PushFrontPopFront) {
    SinglyLinkedList list;
    list.push_front(10);
    list.push_front(20);

    int vals[] = {20, 10};
    int idx = 0;
    for (int val : list) EXPECT_EQ(val, vals[idx++]);
```

```
    list.pop_front();
    vals[0] = 10;
    idx = 0;
    for (int val : list) EXPECT_EQ(val, vals[idx++]);
}
```

Memory Leak Checks:

- Compile with sanitizers: `-fsanitize=address -fno-omit-frame-pointer`
- Use `valgrind` on Linux/macOS for runtime memory leak detection.

5.3.4 Suggested Exercises

1. Implement a **doubly linked list** with `unique_ptr` for `next` and raw pointer for `prev`, ensuring proper cleanup.
2. Add **reverse iteration** support for doubly linked lists.
3. Implement a **merge operation** with iterator compatibility.
4. Test all operations using **unit tests** and **sanitizers** to ensure memory safety.
5. Implement **copy and move constructors** for linked list classes while maintaining proper ownership semantics.

5.3.5 Summary

- **Memory-leak free implementations**: Use smart pointers (`unique_ptr`) to automate resource management.
- **Iterator support**: Provides standard-compliant traversal and improves integration with STL algorithms.

- **Testing**: Always verify correctness and memory safety using unit tests and runtime tools.
- Mastering these practices ensures that linked lists in modern C++ are **robust, efficient, and maintainable**.

Chapter 6

Stacks, Queues, Deques, and Priority Queues

6.1 STL Wrappers vs Custom Implementations: `std::stack`, `std::queue`, `std::deque`, `std::priority_queue`

Linear data structures such as **stacks, queues, deques, and priority queues** are essential building blocks in algorithms. Modern C++ provides both **custom implementation options** and **STL wrappers**, offering flexibility depending on performance, safety, and development requirements. This section examines **STL containers**, their behavior, and trade-offs compared to **hand-crafted implementations**.

6.1.1 STL Wrappers Overview

The C++ Standard Template Library (STL) provides **container adapters** for linear structures:

Adapter	Underlying Container	Description
`std::stack`	`std::deque` (default)	LIFO (Last-In-First-Out) access
`std::queue`	`std::deque` (default)	FIFO (First-In-First-Out) access
`std::priority`	`std::vector` (default)	Max-heap / Min-heap priority ordering

Key Characteristics:

- Designed for **ease of use** and **exception safety**.
- Provide **restricted interfaces**: no random access for `stack` or `queue`.
- Underlying containers can be replaced (e.g., `std::list` or `std::vector`).
- Highly optimized for common operations.

6.1.2 Example: `std::stack`

```
#include <stack>
#include <iostream>

int main() {
    std::stack<int> stk;
    stk.push(10);
    stk.push(20);
    stk.push(30);
```

```
    while (!stk.empty()) {
        std::cout << stk.top() << " ";  // Output: 30 20 10
        stk.pop();
    }
}
```

Properties:

- LIFO ordering: top element is always the last pushed.
- `push`, `pop`, `top` — O(1) operations.
- Safe and concise compared to manual linked-list stack implementation.

6.1.3 Example: `std::queue`

```
#include <queue>
#include <iostream>

int main() {
    std::queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);

    while (!q.empty()) {
        std::cout << q.front() << " ";  // Output: 1 2 3
        q.pop();
    }
}
```

Properties:

- FIFO ordering.
- Supports `push`, `pop`, `front`, `back`.
- Internally uses `deque` by default for efficient insertion/removal.

6.1.4 Example: `std::deque`

A **double-ended queue** allows insertion and deletion at both ends.

```
#include <deque>
#include <iostream>

int main() {
    std::deque<int> dq;
    dq.push_back(1);
    dq.push_front(2);
    dq.push_back(3);

    for (int val : dq) std::cout << val << " ";  // Output: 2 1 3
}
```

Properties:

- Random access supported: `dq[i]` — O(1).
- Insertions/removals at both ends — O(1) amortized.
- Ideal as an underlying container for `stack` or `queue`.

6.1.5 Example: `std::priority_queue`

Max-heap priority queue implementation.

```cpp
#include <queue>
#include <vector>
#include <iostream>

int main() {
    std::priority_queue<int> pq;  // max-heap
    pq.push(10);
    pq.push(5);
    pq.push(20);

    while (!pq.empty()) {
        std::cout << pq.top() << " ";  // Output: 20 10 5
        pq.pop();
    }
}
```

Notes:

- For min-heap, use: `std::priority_queue<int, std::vector<int>, std::greater<int>>`.
- Internally implemented using `std::vector` and heap operations (`push_heap` / `pop_heap`).
- `push` and `pop` — O(log n), `top` — O(1).

6.1.6 Custom Implementations

While STL containers are robust, **custom implementations** provide:

- **Fine-grained control over memory** (e.g., node-based stack or queue).
- Custom features (e.g., size-limited stack, lock-free queue).
- Optimizations for **real-time or embedded systems**.

Custom Stack Example (Linked List Based):

```
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

class Stack {
    Node* topNode = nullptr;
public:
    void push(int val) {
        Node* newNode = new Node(val);
        newNode->next = topNode;
        topNode = newNode;
    }
    void pop() {
        if (!topNode) return;
        Node* temp = topNode;
        topNode = topNode->next;
        delete temp;
    }
    int top() { return topNode->data; }
    bool empty() { return topNode == nullptr; }
};
```

Trade-offs vs STL:

- **Manual memory management** is error-prone.
- Can be optimized for **memory layout or cache locality**.
- Provides deeper understanding of underlying algorithms.

6.1.7 When to Use STL vs Custom

Criterion	STL Adapter	Custom Implementation
Development speed	Very fast	Slower
Safety	Exception-safe, leak-free	Requires careful handling
Flexibility	Limited to interface	Fully customizable
Performance tuning	Slight overhead possible	Fine-grained control
Use in teaching	Can hide complexity	Good for learning internals
Real-time constraints	Less predictable	Optimizable for deterministic behavior

Rule of Thumb:

- Use **STL adapters** for most applications to save time and ensure safety.
- Use **custom structures** when performance profiling indicates bottlenecks, memory layout control is critical, or teaching/learning purposes.

6.1.8 Summary

- **STL wrappers** (`stack`, `queue`, `deque`, `priority_queue`) provide **safe, fast, and standard-compliant** implementations.
- **Custom implementations** offer **educational value** and **fine-grained control**, but require **careful memory management**.
- Understanding both approaches equips the C++ programmer with **flexibility to choose the right solution** for algorithmic challenges.

6.2 Use-Cases and Algorithmic Patterns (Expression Parsing, BFS, Sliding Window Optimums)

Linear data structures like **stacks, queues, deques, and priority queues** are not only foundational for basic storage but also **critical in algorithmic patterns**. Mastering their **use-cases** helps leverage these structures for real-world problem solving in modern C++. This section explores **classic applications** and patterns: **expression parsing, breadth-first search (BFS), and sliding window optimizations**.

6.2.1 Expression Parsing with Stacks

Stacks are widely used for **parsing mathematical expressions**, converting infix to postfix (Reverse Polish Notation), and evaluating expressions efficiently.
Example: Evaluating a postfix expression

```
#include <stack>
#include <string>
#include <sstream>
#include <iostream>

int evaluatePostfix(const std::string& expr) {
    std::stack<int> stk;
    std::istringstream iss(expr);
    std::string token;

    while (iss >> token) {
        if (isdigit(token[0])) {
            stk.push(std::stoi(token));
        } else {
            int b = stk.top(); stk.pop();
```

```
            int a = stk.top(); stk.pop();
            if (token == "+") stk.push(a + b);
            else if (token == "-") stk.push(a - b);
            else if (token == "*") stk.push(a * b);
            else if (token == "/") stk.push(a / b);
        }
    }
    return stk.top();
}

int main() {
    std::string expr = "3 4 + 2 * 7 /";  // ((3+4)*2)/7
    std::cout << evaluatePostfix(expr);   // Output: 2
}
```

Key Points:

- Stack maintains operands temporarily.
- Operators apply to top elements.
- Time complexity: O(n), Space complexity: O(n).
- Extends naturally to **parentheses matching** and **syntax validation**.

6.2.2 Breadth-First Search (BFS) with Queues

Queues are essential in **graph traversal algorithms**, particularly BFS, which explores nodes level by level.

Example: BFS traversal of an adjacency list graph

```cpp
#include <iostream>
#include <vector>
#include <queue>

void BFS(int start, const std::vector<std::vector<int>>& adj) {
    std::vector<bool> visited(adj.size(), false);
    std::queue<int> q;

    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
        int node = q.front(); q.pop();
        std::cout << node << " ";

        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}

int main() {
    std::vector<std::vector<int>> adj = {
        {1, 2}, {0, 3}, {0, 3}, {1, 2}  // graph with 4 nodes
    };
    BFS(0, adj);  // Output: 0 1 2 3
}
```

Key Points:

- Queue maintains nodes to explore.
- Ensures **level-order traversal**.
- Time complexity: O(V + E), Space complexity: O(V).
- Variants include **shortest path in unweighted graphs**.

6.2.3 Sliding Window Optimizations with Deques

Deque (double-ended queue) enables **efficient sliding window computations**, commonly used in problems like **maximum/minimum in a window**, **subarray sums**, and **monotonic queues**.

Example: Maximum in sliding window of size k

```
#include <iostream>
#include <deque>
#include <vector>

std::vector<int> maxSlidingWindow(const std::vector<int>& nums, int k) {
    std::deque<int> dq;
    std::vector<int> result;

    for (int i = 0; i < nums.size(); ++i) {
        while (!dq.empty() && dq.front() <= i - k)
        {
            dq.pop_front();
        }
        while (!dq.empty() && nums[i] >= nums[dq.back()])
        {
            dq.pop_back();
        }
        dq.push_back(i);
```

```
        if (i >= k - 1) result.push_back(nums[dq.front()]);
    }
    return result;
}

int main() {
    std::vector<int> nums = {1, 3, -1, -3, 5, 3, 6, 7};
    int k = 3;
    auto res = maxSlidingWindow(nums, k);
    for (int v : res) std::cout << v << " ";  // Output: 3 3 5 5 6 7
}
```

Key Points:

- Deque maintains **candidate indices** for max values.
- Each element enters and exits deque **at most once**, giving O(n) time.
- Widely applicable in **streaming data, online algorithms, and moving averages**.

6.2.4 Priority Queues in Algorithmic Patterns

Priority queues (heaps) solve problems requiring **dynamic retrieval of maximum or minimum elements**:

- **Dijkstra's algorithm**: min-priority queue to select node with smallest distance.
- **Task scheduling**: retrieve highest priority task efficiently.
- **Median maintenance**: two heaps (max-heap, min-heap).

Example: Top-k elements in a stream

```cpp
#include <queue>
#include <vector>
#include <iostream>

std::vector<int> topK(const std::vector<int>& nums, int k) {
    std::priority_queue<int, std::vector<int>, std::greater<>> minHeap;

    for (int n : nums) {
        minHeap.push(n);
        if (minHeap.size() > k) minHeap.pop();
    }

    std::vector<int> result;
    while (!minHeap.empty()) {
        result.push_back(minHeap.top());
        minHeap.pop();
    }
    return result;
}

int main() {
    std::vector<int> nums = {5, 1, 9, 3, 14, 7};
    auto res = topK(nums, 3);  // Output: 5 9 14
    for (int v : res) std::cout << v << " ";
}
```

Time Complexity: O(n log k), Space Complexity: O(k)

6.2.5 Summary of Patterns and Use-Cases

Data Structure	Common Patterns	Use-Cases
Stack	Expression evaluation, parenthesis matching, DFS (recursive emulation)	Parsing, compilers, undo/redo stacks
Queue	BFS, level-order traversal	Graph traversal, scheduling, streaming
Deque	Sliding window maximum/minimum, two-pointer algorithms	Moving window problems, online algorithms
Priority Queue	Dijkstra, top-k elements, task scheduling	Heaps, greedy algorithms, event-driven simulation

Takeaways:

- Selecting the right linear data structure is **critical to algorithm efficiency**.
- Stacks and queues **control flow**, deques **enable sliding window optimizations**, and priority queues **manage dynamic ordering**.
- Modern C++ STL adapters provide **safe, efficient implementations**, while custom structures allow **fine-tuned control** for specialized use-cases.

6.3 Exercises: Monotonic Queue, K-Largest Using Heaps

This section provides **hands-on exercises** designed to reinforce key algorithmic patterns using **deques** and **priority queues**. These exercises emphasize practical problem-solving while maintaining **efficiency** and **memory safety** in modern C++.

6.3.1 Monotonic Queue Exercise

Problem: Maintain a **sliding window maximum** or minimum efficiently using a **monotonic deque**.
Concept:

- A **monotonic queue** keeps elements in strictly increasing or decreasing order.
- Enables **O(n)** processing of windowed maximum or minimum by avoiding redundant comparisons.

Exercise Implementation:

```
#include <iostream>
#include <deque>
#include <vector>

std::vector<int> slidingWindowMax(const std::vector<int>& nums, int k) {
    std::deque<int> dq;  // store indices of elements
    std::vector<int> result;

    for (int i = 0; i < nums.size(); ++i) {
        // Remove indices that are out of the current window
        while (!dq.empty() && dq.front() <= i - k)
```

```
        {
            dq.pop_front();
        }

        // Remove smaller elements to maintain decreasing order
        while (!dq.empty() && nums[i] >= nums[dq.back()]) {
            dq.pop_back();
        }

        dq.push_back(i);

        if (i >= k - 1) result.push_back(nums[dq.front()]);
    }
    return result;
}

int main() {
    std::vector<int> nums = {1, 3, -1, -3, 5, 3, 6, 7};
    int k = 3;
    auto res = slidingWindowMax(nums, k);
    for (int v : res) std::cout << v << " ";  // Output: 3 3 5 5 6 7
}
```

Learning Objectives:

- Understand **deque-based sliding window optimizations**.
- Learn to maintain **monotonic properties** efficiently.
- Analyze time complexity: $\mathbf{O(n)}$, space complexity: $\mathbf{O(k)}$.

6.3.2 K-Largest Elements Using Heaps

Problem: Find the **k-largest elements** in a stream or array efficiently.
Concept:

- Use a **min-heap of size k** to track the k largest elements.
- Any new element smaller than the heap top is ignored; larger elements replace the minimum.

Exercise Implementation:

```
#include <queue>
#include <vector>
#include <iostream>

std::vector<int> kLargest(const std::vector<int>& nums, int k) {
    std::priority_queue<int, std::vector<int>, std::greater<>> minHeap;

    for (int n : nums) {
        minHeap.push(n);
        if (minHeap.size() > k) minHeap.pop();
    }

    std::vector<int> result;
    while (!minHeap.empty()) {
        result.push_back(minHeap.top());
        minHeap.pop();
    }
    return result;  // smallest to largest of k-largest
}

int main() {
```

```
    std::vector<int> nums = {5, 1, 9, 3, 14, 7};
    int k = 3;
    auto res = kLargest(nums, k);
    for (int v : res) std::cout << v << " ";  // Output: 5 9 14
}
```

Learning Objectives:

- Understand **heap-based selection algorithms**.
- Learn **priority queue operations**: `push`, `pop`, `top`.
- Analyze complexity: O(n log k), space complexity: O(k).
- Applicable to **stream processing, top-k queries, and event prioritization**.

6.3.3 Suggested Exercises

1. Implement **sliding window minimum** using a monotonic increasing deque.
2. Extend monotonic queue to **dynamic window sizes** (e.g., variable-length intervals).
3. Find **k-smallest elements** using a **max-heap**.
4. Combine **monotonic queue and heap** for complex streaming queries (e.g., top-k in sliding window).
5. Implement **unit tests** using **Catch2** or **GoogleTest** to verify correctness and edge cases (empty input, k > n, duplicate values).

6.3.4 Key Takeaways

- **Monotonic queues** efficiently compute windowed extremes in linear time.
- **Heaps / priority queues** provide a dynamic way to track k-largest or k-smallest elements in streaming data.
- Modern C++ STL containers (`std::deque`, `std::priority_queue`) allow safe, high-performance implementations.
- Combining these structures with **algorithmic patterns** strengthens problem-solving skills in **competitive programming**, **data streams**, and **real-time systems**.

Chapter 7

Hashing and Unordered Containers

7.1 `std::unordered_map/set` Internals, Collision Behavior, Custom Hashers

Modern C++ provides **unordered associative containers**—`std::unordered_map` and `std::unordered_set`—which allow **average constant-time access** using **hashing techniques**. This section explores the **internal mechanisms**, **collision handling**, and the use of **custom hash functions** to optimize performance and adapt to specialized data types.

7.1.1 Internals of `std::unordered_map` and `std::unordered_set`

- Both containers are implemented using **hash tables**.
- Key operations—**insert, find, erase**—typically have **O(1) average complexity**, but worst-case can degrade to **O(n)** if many collisions occur.
- Core components:

1. **Buckets**: An array where hashed keys are stored.
2. **Hash function**: Maps a key to a bucket index. Default: `std::hash<Key>`.
3. **Collision resolution**: Usually implemented with **chaining** (linked lists per bucket) or **open addressing** in some STL implementations.
4. **Load factor**: `size() / bucket_count()`. Exceeding a threshold triggers **rehashing** to maintain performance.

Example: Basic `std::unordered_map` usage

```
#include <unordered_map>
#include <string>
#include <iostream>

int main() {
    std::unordered_map<std::string, int> map;
    map["apple"] = 5;
    map["banana"] = 10;

    std::cout << map["apple"] << "\n";  // Output: 5
    std::cout << map["banana"] << "\n"; // Output: 10
}
```

7.1.2 Collision Behavior

Collisions occur when **two distinct keys hash to the same bucket**.
Common strategies in STL:

- **Separate chaining (linked lists)**: Multiple elements share a bucket using a linked list. Average lookup: O(1 +), where = load factor.

- **Rehashing**: On insertion, if load factor exceeds a threshold (typically 1.0), the table **resizes** to reduce collisions.

Observations:

- Poor hash functions can degrade performance to O(n).
- `std::unordered_map` and `std::unordered_set` rely on **quality of `std::hash`** for primitive types.
- Iteration order is **unspecified** and can change after rehashing.

7.1.3 Custom Hash Functions

For complex or user-defined types, **custom hashers** are necessary. Modern C++ allows defining hash functions via **function objects or lambdas**.
Example: Custom hash for a `Point` struct

```
#include <unordered_map>
#include <iostream>

struct Point {
    int x, y;
    bool operator==(const Point& other) const {
        return x == other.x && y == other.y;
    }
};

// Custom hash function
struct PointHasher {
    std::size_t operator()(const Point& p) const {
        std::size_t h1 = std::hash<int>{}(p.x);
```

```
        std::size_t h2 = std::hash<int>{}(p.y);
        return h1 ^ (h2 << 1); // Combine hashes
    }
};

int main() {
    std::unordered_map<Point, std::string, PointHasher> map;
    map[{1,2}] = "A";
    map[{3,4}] = "B";

    std::cout << map[{1,2}] << "\n";  // Output: A
    std::cout << map[{3,4}] << "\n";  // Output: B
}
```

Key Points:

- Always implement `operator==` for keys.
- Combine hashes carefully to minimize collisions.
- Custom hashers allow **efficient storage of tuples, structs, or complex objects**.

7.1.4 Load Factor and Rehashing

- **Load factor** controls **space-time trade-offs**.
- Default **max load factor**: 1.0.
- `rehash()` and `reserve()` can **preallocate buckets** to reduce runtime collisions.

```
std::unordered_map<int, int> map;
map.max_load_factor(0.75f); // Lower threshold triggers rehash sooner
map.reserve(100);           // Allocate buckets in advance
```

Benefits:

- Reduces the cost of repeated insertions.
- Improves cache performance and lookup speed.

7.1.5 Performance Considerations

Factor	Impact on Performance
Hash quality	Low-quality hashes → collisions → $O(n)$ lookups
Load factor	High load factor → more collisions; low load factor → memory overhead
Bucket count	More buckets → fewer collisions, but higher memory usage
Key type	Complex keys require efficient custom hashers

Best Practices:

- Use `std::hash` for primitive types.
- For custom types, define **strong hashers** and `operator==`.
- Preallocate with `reserve()` when size is known.
- Avoid excessive copying; store pointers or references if possible.

7.1.6 Summary

- `std::unordered_map` and `std::unordered_set` provide **hash table–based O(1) average access**.
- Collisions are handled with **chaining** and **rehashing**.
- **Custom hash functions** enable efficient use of complex types as keys.
- Proper tuning of **load factor** and **bucket count** ensures optimal performance.
- Mastery of STL unordered containers is essential for **algorithmic efficiency in modern C++**.

7.2 Hash-Based Algorithms: Frequency Counting, Two-Sum, Caching Strategies

Hash-based algorithms leverage **constant-time average lookups** to solve a variety of problems efficiently. Modern C++ provides **unordered associative containers** (`std::unordered_map`, `std::unordered_set`) which allow **fast insertion, search, and deletion**, making them ideal for **frequency analysis, pair sum problems, and caching**. This section explores common patterns and practical C++ implementations.

7.2.1 Frequency Counting

Problem: Count the occurrences of elements in an array or stream.
C++ Implementation:

```cpp
#include <unordered_map>
#include <vector>
#include <iostream>

int main() {
    std::vector<int> nums = {1, 3, 2, 1, 3, 1, 4};
    std::unordered_map<int, int> freq;

    for (int n : nums) {
        freq[n]++;  // Increment count for each element
    }

    for (auto &[key, value] : freq) {
        std::cout << key << ": " << value << "\n";
    }
    // Possible Output: 1:3 3:2 2:1 4:1
```

```
}
```

Key Points:

- **Time complexity:** O(n) average.
- **Space complexity:** O(n) for the map.
- Supports **stream processing**, e.g., counting events in real-time.
- Can be extended to **frequency of strings, tuples, or custom objects** using custom hashers.

7.2.2 Two-Sum Problem

Problem: Given an array and a target sum, find two numbers that add up to the target.

C++ Implementation:

```
#include <unordered_map>
#include <vector>
#include <iostream>

std::pair<int, int> twoSum(const std::vector<int>& nums, int target) {
    std::unordered_map<int, int> seen; // number -> index

    for (int i = 0; i < nums.size(); ++i) {
        int complement = target - nums[i];
        if (seen.find(complement) != seen.end()) {
            return {seen[complement], i};
        }
        seen[nums[i]] = i;
```

```
    }
    return {-1, -1};  // not found
}

int main() {
    std::vector<int> nums = {2, 7, 11, 15};
    int target = 9;
    auto result = twoSum(nums, target);
    std::cout << result.first << ", " << result.second;  // Output: 0, 1
}
```

Key Points:

- Hash map stores previously seen numbers for O(1) lookup.
- **Time complexity:** O(n) average.
- **Space complexity:** O(n).
- Can be extended to **k-sum problems** with more sophisticated hash-based strategies.

7.2.3 Caching Strategies (Memoization & LRU Cache)

Hash-based containers are central to **caching** and **memoization**, enabling **fast retrieval of computed results**.

Example: Memoization for Fibonacci Sequence

```
#include <unordered_map>
#include <iostream>

std::unordered_map<int, long long> memo;
```

```
long long fib(int n) {
    if (n <= 1) return n;
    if (memo.find(n) != memo.end()) return memo[n];
    memo[n] = fib(n-1) + fib(n-2);
    return memo[n];
}

int main() {
    std::cout << fib(50) << "\n";  // Efficiently computes large Fibonacci
}
```

LRU Cache Pattern:

- Combine `unordered_map` with a **doubly linked list or deque**.
- Map stores key → value; list maintains **recency order**.
- On access, move key to front; on insertion when full, remove least recently used.

Key Points:

- Hashing enables **average O(1) lookup**.
- Essential in **dynamic programming, online algorithms, and performance-sensitive systems**.
- Custom hashers can optimize cache behavior for complex keys.

7.2.4 Best Practices

- Use `unordered_map`/`unordered_set` for **frequent insertions and lookups**.

- For custom types, define **robust hash functions** and equality operators.
- Monitor **load factor** and **rehashing** to maintain consistent performance.
- Combine hash-based containers with **other linear structures** (stack, deque) for advanced patterns.
- For large-scale systems, consider **memory usage** and potential **hash collisions**.

7.2.5 Summary

- **Frequency counting**: Simple, efficient, and applicable to data streams and analytics.
- **Two-sum and general pair-sum problems**: Hash maps provide O(n) solutions vs $O(n^2)$ brute force.
- **Caching and memoization**: Hash tables are foundational for storing and retrieving intermediate results.
- Hash-based algorithms are a **cornerstone in modern C++**, offering **high-performance solutions** for real-world and competitive programming problems.

7.3 Exercises: Implement LRU Cache, Robin-Hood/Linear-Probing Sketch

This section provides **practical exercises** to solidify understanding of **hashing strategies and cache design**. Exercises focus on **implementing an LRU cache** and exploring **hash table collision handling techniques** such as **Robin-Hood hashing** and **linear probing**. These exercises emphasize **modern C++ implementations**, memory safety, and efficiency.

7.3.1 Exercise: Implement LRU Cache

Problem: Design a **Least Recently Used (LRU) cache** with **O(1) access and update time**.
Key Components:

- `std::unordered_map<Key, Iterator>` to store key → iterator mapping for fast access.
- `std::list<std::pair<Key, Value>>` to maintain **recency order** (front = most recent, back = least recent).
- On access, move the element to the front.
- On insertion, if the cache exceeds capacity, remove the **least recently used** item from the back.

C++ Implementation Sketch:

```
#include <unordered_map>
#include <list>
#include <iostream>

template <typename Key, typename Value>
class LRUCache {
private:
    size_t capacity;
    std::list<std::pair<Key, Value>> items; // recency order
    std::unordered_map<Key, typename std::list<std::pair<Key, Value>>::iterator> map;

public:
    LRUCache(size_t cap) : capacity(cap) {}

    Value get(const Key& key) {
        auto it = map.find(key);
        if (it == map.end()) throw std::runtime_error("Key not found");

        // Move accessed item to front
        items.splice(items.begin(), items, it->second);
        return it->second->second;
    }

    void put(const Key& key, const Value& value) {
        auto it = map.find(key);
        if (it != map.end()) {
            // Update value and move to front
            it->second->second = value;
            items.splice(items.begin(), items, it->second);
        } else {
            if (items.size() >= capacity) {
                // Remove least recently used
```

```
                auto last = items.back();
                map.erase(last.first);
                items.pop_back();
            }
            items.emplace_front(key, value);
            map[key] = items.begin();
        }
    }
};

int main() {
    LRUCache<int, std::string> cache(2);
    cache.put(1, "A");
    cache.put(2, "B");
    std::cout << cache.get(1) << "\n";  // Output: A
    cache.put(3, "C");                  // Evicts key 2
}
```

Learning Objectives:

- Combine **unordered_map** and **list** for efficient cache operations.
- Understand **recency order maintenance**.
- Apply modern C++ **templates** and **iterators** for generic solutions.

7.3.2 Exercise: Robin-Hood and Linear-Probing Sketch

Problem: Explore **open-addressing hash table techniques**, focusing on **linear probing** and **Robin-Hood hashing**.
Concepts:

- **Linear probing:** On collision, sequentially scan next available bucket.

- **Robin-Hood hashing:** On collision, evict element with **shorter probe distance** to ensure more uniform access times.

C++ Implementation Sketch:

```
#include <vector>
#include <optional>
#include <iostream>

template <typename Key, typename Value>
class LinearProbingHashTable {
private:
    struct Entry { Key key; Value value; bool occupied = false; };
    std::vector<Entry> table;
    size_t capacity;

    size_t hash(const Key& key) const { return std::hash<Key>{}(key) % capacity; }

public:
    LinearProbingHashTable(size_t cap) : capacity(cap), table(cap) {}

    void insert(const Key& key, const Value& value) {
        size_t idx = hash(key);
        size_t start = idx;
        while (table[idx].occupied) {
            if (table[idx].key == key) {
                table[idx].value = value;  // update
                return;
            }
            idx = (idx + 1) % capacity;
            if (idx == start) throw std::runtime_error("Hash table full");
        }
        table[idx] = {key, value, true};
```

```
    }

    std::optional<Value> find(const Key& key) const {
        size_t idx = hash(key);
        size_t start = idx;
        while (table[idx].occupied) {
            if (table[idx].key == key) return table[idx].value;
            idx = (idx + 1) % capacity;
            if (idx == start) break;
        }
        return std::nullopt;
    }
};

int main() {
    LinearProbingHashTable<int, std::string> ht(5);
    ht.insert(1, "A");
    ht.insert(6, "B");  // Collision handled via linear probing
    auto val = ht.find(6);
    if (val) std::cout << *val << "\n";  // Output: B
}
```

Key Points:

- Linear probing reduces **pointer overhead** compared to chaining.
- Robin-Hood improves **variance of probe lengths**, balancing access times.
- Requires careful handling of **deleted elements** and **rehashing**.
- Illustrates **fundamental mechanics behind `std::unordered_map`**.

7.3.3 Suggested Exercises

1. Extend LRU cache to support **time-based expiration**.
2. Implement **Robin-Hood hashing with probe distance tracking**.
3. Compare **linear probing vs separate chaining** performance for high load factors.
4. Write **unit tests** for edge cases: full capacity, duplicate keys, deletion scenarios.
5. Integrate **custom hashers** for complex keys in both LRU and open-addressing implementations.

7.3.4 Summary

- **LRU Cache** demonstrates combining **hash maps and linear containers** for efficient caching.
- **Open-addressing hash tables** provide insight into collision resolution strategies like **linear probing and Robin-Hood hashing**.
- Exercises reinforce **memory-safe, high-performance C++ design patterns** relevant for **systems programming, real-time processing, and competitive algorithms**.
- Understanding these patterns equips readers to **tune hash-based containers** and implement **advanced caching strategies** effectively.

Part III

Trees & Balanced Trees

Chapter 8

Binary Trees & Tree Traversals

8.1 Node Representation, Recursive vs Iterative Traversal, Iterator Adapters

Binary trees are one of the **fundamental data structures** in computer science, providing hierarchical organization of data and forming the backbone of advanced structures such as **heaps, search trees, and expression trees**. This section examines **node representation**, contrasts **recursive and iterative traversal strategies**, and introduces **iterator adapters** in modern C++ for clean and efficient tree traversal.

8.1.1 Node Representation

In modern C++, binary tree nodes can be represented using **raw pointers**, **smart pointers**, or **container-based approaches**.

a) Raw pointer approach (classic)

```
struct Node {
    int value;
    Node* left = nullptr;
    Node* right = nullptr;

    Node(int v) : value(v) {}
};
```

- Simple and familiar.
- Requires **manual memory management** (risk of leaks).
- Suitable for educational examples and low-level implementations.

b) Smart pointer approach (modern C++ safe alternative)

```
#include <memory>

struct Node {
    int value;
    std::unique_ptr<Node> left;
    std::unique_ptr<Node> right;

    Node(int v) : value(v) {}
};
```

- `std::unique_ptr` ensures **automatic memory management**.
- Prevents **dangling pointers** and **memory leaks**.
- Preferred in production-grade C++ code.

c) Container-based representation (for compact storage)

```
#include <vector>
struct Node {
    int value;
    int left_index = -1;
    int right_index = -1;
};
```

- Useful for **static or memory-constrained environments**.
- Tree nodes are stored in a vector or array, and children are referenced via indices.
- Enables **cache-friendly traversal** and easier serialization.

8.1.2 Recursive Traversal

Recursive traversals are simple, expressive, and closely match the mathematical definition of trees.

Pre-order, In-order, Post-order Traversal (Recursive)

```
void preorder(const Node* node) {
    if (!node) return;
    std::cout << node->value << " ";
    preorder(node->left.get());
    preorder(node->right.get());
}

void inorder(const Node* node) {
    if (!node) return;
    inorder(node->left.get());
    std::cout << node->value << " ";
```

```
    inorder(node->right.get());
}

void postorder(const Node* node) {
    if (!node) return;
    postorder(node->left.get());
    postorder(node->right.get());
    std::cout << node->value << " ";
}
```

Advantages:

- Simple and readable.
- Easy to implement for most problems.

Limitations:

- Recursive depth limited by stack size (risk of **stack overflow** for deep trees).
- Less control over **memory footprint** during traversal.

8.1.3 Iterative Traversal

Iterative traversals avoid recursion, using **explicit stacks** or **queues**.
Example: Iterative In-order Traversal

```
#include <stack>

void inorderIterative(const Node* root) {
    std::stack<const Node*> st;
    const Node* current = root;
```

```
    while (current || !st.empty()) {
        while (current) {
            st.push(current);
            current = current->left.get();
        }
        current = st.top(); st.pop();
        std::cout << current->value << " ";
        current = current->right.get();
    }
}
```

Advantages:

- Avoids recursion and **stack overflow**.
- Can be combined with **early termination** or **custom traversal logic**.

Trade-offs:

- Slightly more verbose than recursion.
- Requires explicit **stack or queue management**.

8.1.4 Iterator Adapters for Trees

Modern C++ encourages **iterator-based access** to abstract data structures. An **iterator adapter** provides **range-based traversal** without exposing tree internals.
Example: Simple In-order Iterator Skeleton

```
#include <stack>

class InOrderIterator {
    std::stack<const Node*> st;
    const Node* current;

public:
    explicit InOrderIterator(const Node* root) : current(root) {
        while (current) { st.push(current); current = current->left.get(); }
        advance();
    }

    const Node* operator*() const { return current; }

    InOrderIterator& operator++() { advance(); return *this; }

    bool operator!=(const InOrderIterator& other) const { return current !=
    ↪  other.current; }

private:
    void advance() {
        if (st.empty()) { current = nullptr; return; }
        current = st.top(); st.pop();
        const Node* node = current->right.get();
        while (node) { st.push(node); node = node->left.get(); }
    }
};

% <-- blank line here is required
```

Benefits:

- Enables **range-based for loops** over trees:

```
for (auto node : InOrderRange(root)) { /* use node->value */ }
```

- Encapsulates traversal logic, improving **code clarity and reusability**.
- Compatible with **algorithms in `<algorithm>`**.

8.1.5 Summary

- **Node representation** can be raw pointers, smart pointers, or index-based containers.
- **Recursive traversal** is simple and expressive but limited by stack depth.
- **Iterative traversal** avoids recursion, providing better control for deep or unbalanced trees.
- **Iterator adapters** allow modern C++ idioms like **range-based loops** and **algorithm compatibility**, encouraging safer and more maintainable tree processing.
- Understanding these patterns is essential for implementing **efficient tree algorithms**, balancing **performance, safety, and code clarity**.

8.2 Algorithms — Preorder/Inorder/Postorder, Level-Order, Tree Serialization/Deserialization

Binary tree traversal algorithms form the **foundation of tree processing**. They provide structured ways to visit all nodes and are prerequisites for higher-level algorithms like balancing, searching, or serialization. This section covers the **classical traversals** (preorder, inorder, postorder, and level-order) and explains how to implement **serialization and deserialization**, enabling binary trees to be stored or transmitted efficiently.

8.2.1 Depth-First Traversals

Depth-first traversals visit nodes by following branches deeply before backtracking. They can be implemented either **recursively** or **iteratively** with an explicit stack.

a) Preorder Traversal (Root → Left → Right)

- Processes the **root first**, then left subtree, then right subtree.
- Useful for **tree cloning, expression construction**, and serialization.

```
void preorder(const Node* node) {
    if (!node) return;
    std::cout << node->value << " ";
    preorder(node->left.get());
    preorder(node->right.get());
}
```

Iterative version with stack:

```
#include <stack>

void preorderIterative(const Node* root) {
    if (!root) return;
    std::stack<const Node*> st;
    st.push(root);

    while (!st.empty()) {
        auto node = st.top(); st.pop();
        std::cout << node->value << " ";
        if (node->right) st.push(node->right.get());
        if (node->left) st.push(node->left.get());
    }
}
```

b) Inorder Traversal (Left → Root → Right)

- Visits nodes in **sorted order** for Binary Search Trees.
- Central to algorithms like **BST validation** and **range queries**.

```
void inorder(const Node* node) {
    if (!node) return;
    inorder(node->left.get());
    std::cout << node->value << " ";
    inorder(node->right.get());
}
```

Iterative version uses a **stack**:

```
void inorderIterative(const Node* root) {
    std::stack<const Node*> st;
    const Node* current = root;

    while (current || !st.empty()) {
        while (current) {
            st.push(current);
            current = current->left.get();
        }
        current = st.top(); st.pop();
        std::cout << current->value << " ";
        current = current->right.get();
    }
}
```

c) Postorder Traversal (Left → Right → Root)

- Processes children **before the root**.
- Used in **tree deletion**, **expression evaluation**, and freeing resources.

```
void postorder(const Node* node) {
    if (!node) return;
    postorder(node->left.get());
    postorder(node->right.get());
    std::cout << node->value << " ";
}
```

Iterative version (using two stacks):

```
void postorderIterative(const Node* root) {
    if (!root) return;
    std::stack<const Node*> st1, st2;
    st1.push(root);

    while (!st1.empty()) {
        auto node = st1.top(); st1.pop();
        st2.push(node);
        if (node->left) st1.push(node->left.get());
        if (node->right) st1.push(node->right.get());
    }

    while (!st2.empty()) {
        std::cout << st2.top()->value << " ";
        st2.pop();
    }
}
```

8.2.2 Breadth-First Traversal (Level-Order)

Level-order traversal processes nodes **level by level**, using a queue. This is the basis of **BFS algorithms** on trees.

```
#include <queue>

void levelOrder(const Node* root) {
    if (!root) return;
    std::queue<const Node*> q;
    q.push(root);

    while (!q.empty()) {
```

```
        auto node = q.front(); q.pop();
        std::cout << node->value << " ";
        if (node->left) q.push(node->left.get());
        if (node->right) q.push(node->right.get());
    }
}
```

Applications:

- Shortest path in unweighted trees.
- Layered computations (e.g., **minimum depth**).
- Serialization format aligned with **breadth-first storage**.

8.2.3 Tree Serialization & Deserialization

Serialization converts a tree into a **linear representation** (string or array).
Deserialization reconstructs the tree from this format.

a) Preorder Serialization (with null markers)

```
#include <sstream>

void serialize(const Node* node, std::ostringstream& out) {
    if (!node) {
        out << "# ";
        return;
    }
    out << node->value << " ";
    serialize(node->left.get(), out);
    serialize(node->right.get(), out);
}
```

```
std::string serialize(const Node* root) {
    std::ostringstream out;
    serialize(root, out);
    return out.str();
}
```

b) Preorder Deserialization

```
#include <sstream>
#include <memory>

std::unique_ptr<Node> deserialize(std::istringstream& in) {
    std::string val;
    in >> val;
    if (val == "#") return nullptr;

    auto node = std::make_unique<Node>(std::stoi(val));
    node->left = deserialize(in);
    node->right = deserialize(in);
    return node;
}

std::unique_ptr<Node> deserialize(const std::string& data) {
    std::istringstream in(data);
    return deserialize(in);
}
```

c) Level-order Serialization/Deserialization

- Stores nodes level by level with # placeholders.
- More **compact for complete/balanced trees**, and often used in competitive

programming.

```
std::string serializeLevelOrder(const Node* root) {
    if (!root) return "";
    std::ostringstream out;
    std::queue<const Node*> q;
    q.push(root);

    while (!q.empty()) {
        auto node = q.front(); q.pop();
        if (!node) {
            out << "# ";
            continue;
        }
        out << node->value << " ";
        q.push(node->left.get());
        q.push(node->right.get());
    }
    return out.str();
}
```

Deserialization follows the same logic, reconstructing child links **level by level**.

8.2.4 Summary

- **Preorder, Inorder, and Postorder** traversals enable depth-first exploration, each serving different algorithmic roles.
- **Level-order** traversal is essential for BFS and layered processing.
- **Serialization and deserialization** allow trees to be stored, transmitted, and reconstructed reliably.

- Both **recursive** and **iterative** implementations are important: recursion for clarity, iteration for stack safety in large or skewed trees.

8.3 Exercises — Reconstruct Tree from Traversals, Subtree Checks

Exercises in binary trees often move beyond traversal mechanics and focus on **reconstruction** and **structural comparisons**. Both tasks are fundamental for algorithm developers: reconstruction exercises deepen understanding of traversal uniqueness, while subtree checks model real-world problems such as **pattern matching in hierarchical data** or **subgraph detection**.

8.3.1 Reconstructing a Tree from Traversals

Different traversal orders provide complementary information. The most common problems ask to rebuild a binary tree given two traversals.

- **a) Reconstruct from Preorder + Inorder**
  - **Preorder** reveals the **root** immediately (first element).
  - **Inorder** gives the **left and right subtree boundaries** relative to the root.

  Algorithm:

  1. Use the preorder sequence to pick the root.
  2. Split the inorder sequence into left and right subtrees around the root.
  3. Recurse on left and right parts.

```
#include <vector>
#include <unordered_map>
#include <memory>
```

```cpp
struct Node {
    int value;
    std::unique_ptr<Node> left, right;
    Node(int v) : value(v) {}
};

std::unique_ptr<Node> buildPreIn(
    const std::vector<int>& preorder, int preL, int preR,
    const std::vector<int>& inorder, int inL, int inR,
    const std::unordered_map<int,int>& inIndex)
{
    if (preL > preR || inL > inR) return nullptr;

    int rootVal = preorder[preL];
    auto root = std::make_unique<Node>(rootVal);

    int mid = inIndex.at(rootVal);
    int leftSize = mid - inL;

    root->left = buildPreIn(preorder, preL + 1, preL + leftSize,
                            inorder, inL, mid - 1, inIndex);
    root->right = buildPreIn(preorder, preL + leftSize + 1, preR,
                             inorder, mid + 1, inR, inIndex);
    return root;
}

std::unique_ptr<Node> buildTreePreIn(
    const std::vector<int>& preorder, const std::vector<int>& inorder)
{
    std::unordered_map<int,int> inIndex;
    for (int i = 0; i < inorder.size(); ++i) {
```

```
        inIndex[inorder[i]] = i;
    }
    return buildPreIn(preorder, 0, preorder.size() - 1,
                      inorder, 0, inorder.size() - 1, inIndex);
}
```

Complexity:

- Time: `O(n)` using hash map for index lookups.
- Space: `O(n)` recursion depth worst case (skewed tree).

- **b) Reconstruct from Postorder + Inorder**
  - **Postorder** reveals the **root last**.
  - Inorder provides subtree boundaries as before.

Algorithm is symmetric: choose the last postorder element as root, split inorder, recurse left and right.

```
std::unique_ptr<Node> buildPostIn(
    const std::vector<int>& postorder, int postL, int postR,
    const std::vector<int>& inorder, int inL, int inR,
    const std::unordered_map<int,int>& inIndex)
{
    if (postL > postR || inL > inR) return nullptr;

    int rootVal = postorder[postR];
    auto root = std::make_unique<Node>(rootVal);

    int mid = inIndex.at(rootVal);
```

```
    int leftSize = mid - inL;

    root->left = buildPostIn(postorder, postL, postL + leftSize - 1,
                             inorder, inL, mid - 1, inIndex);
    root->right = buildPostIn(postorder, postL + leftSize, postR - 1,
                              inorder, mid + 1, inR, inIndex);
    return root;
}
```

- **c) Reconstruct from Preorder + Postorder**
  - Without inorder, reconstruction is **ambiguous** unless the tree is **full** (every node has 0 or 2 children).
  - Common in competitive programming exercises.

  Key idea:
  - Root is first in preorder, last in postorder.
  - Divide subtrees by next preorder element's position in postorder.

8.3.2 Subtree Checks

Subtree problems check whether one tree (`T2`) is a subtree of another (`T1`).

- **a) Direct Recursive Check**

  Definition: `T2` is a subtree of `T1` if either:
  - `T1` matches `T2` entirely, or
  - `T2` is a subtree of `T1->left` or `T1->right`.

```
bool isSame(const Node* a, const Node* b) {
    if (!a && !b) return true;
    if (!a || !b) return false;
    return (a->value == b->value &&
            isSame(a->left.get(), b->left.get()) &&
            isSame(a->right.get(), b->right.get()));
}

bool isSubtree(const Node* root, const Node* sub) {
    if (!sub) return true;
    if (!root) return false;
    if (isSame(root, sub)) return true;
    return isSubtree(root->left.get(), sub) || isSubtree(root->right.get(),
    ↪  sub);
}
```

Complexity:

- Worst-case time: `O(|T1| * |T2|)` (every node of `T1` checked against root of `T2`).

- **b) Optimized Subtree Check with Serialization**

  Another approach serializes both trees (e.g., preorder with null markers) and checks whether the string of `T2` is a substring of `T1`.

```
void serializePreorder(const Node* node, std::ostringstream& out) {
    if (!node) { out << "# "; return; }
    out << node->value << " ";
    serializePreorder(node->left.get(), out);
    serializePreorder(node->right.get(), out);
}
```

```
bool isSubtreeSerialized(const Node* root, const Node* sub) {
    std::ostringstream s1, s2;
    serializePreorder(root, s1);
    serializePreorder(sub, s2);
    return s1.str().find(s2.str()) != std::string::npos;
}
```

Pros:

- Converts structural matching to **string matching**.
- Can be optimized further with KMP or rolling hash.

8.3.3 Suggested Exercises

1. **Reconstruction Tasks**
   - Build tree from preorder + inorder.
   - Build tree from postorder + inorder.
   - Handle reconstruction when input traversals represent a skewed tree.
2. **Subtree Problems**
   - Implement `isSubtree` using recursion.
   - Optimize using serialization and substring search.
   - Extend to **substructure checks** (allow partial matches, like prefix trees).
3. **Validation Tasks**

- Write unit tests to validate reconstruction correctness by traversing the built tree.
- Verify `isSubtree` with both balanced and skewed examples.
- Measure complexity by testing large random trees.

8.3.4 Summary

- Reconstruction exercises deepen mastery of **traversal relationships**.
- Subtree checks illustrate the power of **recursive structure comparison** and serialization tricks.
- Together, they provide strong foundations for **tree-based problem-solving**, with applications in compilers, databases, XML/JSON parsing, and more.

Chapter 9

Binary Search Trees & Augmented Trees

9.1 BST Operations, Invariants, Performance Edge Cases

Binary Search Trees (BSTs) form the **backbone of many higher-level data structures** such as balanced trees, order-statistic trees, and interval trees. A BST organizes elements so that each node satisfies a simple yet powerful invariant, enabling efficient search, insertion, and deletion. However, performance depends heavily on maintaining balance; otherwise, operations degrade to linear time.

9.1.1 The BST Invariant

For each node `N` in the tree:

- All keys in the **left subtree** are **strictly less** than `N.key`.
- All keys in the **right subtree** are **strictly greater** than `N.key`.

This invariant ensures that an **inorder traversal yields a sorted sequence** of all elements.

Modern C++ Implementation (Node):

```
#include <memory>

struct Node {
    int key;
    std::unique_ptr<Node> left, right;
    Node(int k) : key(k) {}
};
```

We use `std::unique_ptr` to manage memory automatically, avoiding leaks common in raw-pointer implementations.

9.1.2 Core BST Operations

- **a) Search**

  Searching proceeds by recursively or iteratively comparing the target value to the current node's key:

```
const Node* search(const Node* root, int key) {
    if (!root || root->key == key) {
        return root;
    }
    if (key < root->key) {
        return search(root->left.get(), key);
    }
    return search(root->right.get(), key);
}
```

  - **Best/Average case:** `O(log n)` in balanced trees.
  - **Worst case:** `O(n)` in degenerate (skewed) trees.

- **b) Insertion**

  Insert by descending the tree until reaching the correct null position:

```
void insert(std::unique_ptr<Node>& root, int key) {
    if (!root) {
        root = std::make_unique<Node>(key);
        return;
    }
    if (key < root->key) insert(root->left, key);
    else if (key > root->key) insert(root->right, key);
```

```
        // duplicate keys ignored by this version
    }
```

Invariant preserved:

- The new key is placed in a position consistent with BST ordering.
- No reordering of existing nodes is needed.

- **c) Deletion**

  Deletion is the trickiest operation, requiring different cases:

  1. **Node is a leaf:** Simply remove it.
  2. **Node has one child:** Replace the node with its child.
  3. **Node has two children:** Replace the node with its **inorder successor** (smallest in right subtree) or **inorder predecessor** (largest in left subtree), then delete that node recursively.

```
        Node* findMin(Node* root) {
            while (root && root->left) {
                root = root->left.get();
                }
            return root;
        }

        void remove(std::unique_ptr<Node>& root, int key) {
            if (!root) return;
            if (key < root->key) {
                    remove(root->left, key);
                }
```

```
        else if (key > root->key) {
            remove(root->right, key);
            }
            else {
                if (!root->left) {
                    root = std::move(root->right);
                } else if (!root->right) {
                    root = std::move(root->left);
                } else {
                    Node* successor = findMin(root->right.get());
                    root->key = successor->key;
                    remove(root->right, successor->key);
                }
        }
    }
```

9.1.3 Performance Considerations

- **a) Balanced vs Unbalanced Trees**
  - In a **balanced BST**, height `O(log n)`, making search/insert/delete efficient.
  - In a **skewed BST** (e.g., inserting sorted input into an unbalanced tree), height = `O(n)`, degenerating to linked-list performance.

Example of edge case:

```
// inserting sorted input into a naive BST
for (int i = 1; i <= n; ++i) insert(root, i);
// height = n, all operations degrade to linear
```

- **b) Randomization and Balancing**
  - To avoid skewness, randomized insertions or **self-balancing trees** (AVL, Red-Black) are used in practice.
  - Standard Library associative containers like `std::set` and `std::map` use **Red-Black Trees**, guaranteeing logarithmic performance.
- **c) Duplicate Handling**
  - Simple BSTs often **ignore duplicates**.
  - Variants store a **count field** or allow duplicates in one subtree (commonly the right).
  - Handling duplicates consistently is crucial for correctness in algorithms such as frequency counting.
- **d) Memory and Cache Behavior**
  - Each node requires dynamic allocation (`new`), which can fragment memory.
  - Linked structure results in **poor cache locality** compared to flat arrays or B-trees.
  - Optimized structures (like **van Emde Boas layouts** or **B-trees**) mitigate these issues in high-performance contexts.

9.1.4 Edge Cases to Address in Implementations

1. **Empty tree operations:** Ensure insert, search, and delete handle `nullptr` gracefully.

2. **Deletion of root node:** Requires special care when root has two children.

3. **Skewed inputs:** Repeated sorted input should trigger awareness of imbalance.

4. **Large trees:** Recursion depth can exceed stack limits—iterative implementations are safer for robustness.

5. **Duplicate keys:** Must define a consistent policy to avoid breaking invariants.

9.1.5 Summary

- BSTs rely on the **ordering invariant** that left $<$ root $<$ right.

- Operations (search, insert, delete) are conceptually simple but must carefully maintain the invariant.

- Performance hinges on tree height: balanced trees achieve `O(log n)`, while skewed trees degrade to `O(n)`.

- Edge cases such as duplicate handling, memory fragmentation, and recursion depth must be considered in robust implementations.

- Modern C++ encourages safe memory management (`unique_ptr`) and clean interfaces for correctness.

9.2 Augmented Trees for Range Queries and Order Statistics (`order_of_key`)

While basic Binary Search Trees (BSTs) support search, insertion, and deletion, many practical applications require **augmented capabilities**: efficiently computing order statistics, supporting range queries, and enabling rank-based lookups. Augmented trees enrich nodes with additional metadata—such as subtree sizes, counts, or ranges—while preserving the BST invariant.

9.2.1 Motivation for Augmented Trees

In many domains, data structures must answer queries like:

- **Order statistics:** "What is the 5th smallest element?"
- **Rank queries:** "How many elements are less than key = 42?"
- **Range queries:** "How many elements lie between `L` and `R`?"

A plain BST cannot answer these queries efficiently because it only stores keys and relies on traversal for order-related information. Augmented trees embed auxiliary data in each node to provide these answers in logarithmic time.

9.2.2 Core Augmentation: Subtree Size

Each node is annotated with the **size of its subtree** (i.e., total number of nodes in the subtree rooted at that node).

Augmented Node Definition:

```cpp
#include <memory>

struct Node {
    int key;
    int size;  // number of nodes in this subtree
    std::unique_ptr<Node> left, right;

    Node(int k) : key(k), size(1) {}
};

// utility to get size safely
int getSize(const std::unique_ptr<Node>& n) {
    return n ? n->size : 0;
}

// update size after insertion or deletion
void updateSize(Node* n) {
    if (n) {
        n->size = 1 + getSize(n->left) + getSize(n->right);
    }
}
```

Every insertion or deletion must **recompute the size** when unwinding the recursion.

9.2.3 Order Statistics

- **a) `kth_element` (find k-th smallest element)**

  We can navigate using subtree sizes:

  - Let `leftSize = size(left)`.
  - If `k == leftSize + 1`, the root is the k-th smallest.

  - If `k <= leftSize`, recurse left.
  - Otherwise, recurse right with adjusted `k`.

```
const Node* kthElement(const Node* root, int k) {
    if (!root) return nullptr;
    int leftSize = getSize(root->left);
    if (k == leftSize + 1) return root;
    if (k <= leftSize) return kthElement(root->left.get(), k);
    return kthElement(root->right.get(), k - leftSize - 1);
}
```

Time complexity: `O(log n)` in balanced trees, `O(n)` in skewed trees.

- **b) `order_of_key` (rank of a key)**

  This function computes **how many keys are strictly smaller than `x`**.

  - If `x <= root->key`, recurse left.
  - If `x > root->key`, the rank includes `1 + size(left)` plus rank in the right subtree.

```
int orderOfKey(const Node* root, int x) {
    if (!root) return 0;
    if (x <= root->key) {
        return orderOfKey(root->left.get(), x);
    } else {
        return 1 + getSize(root->left) + orderOfKey(root->right.get(), x);
    }
}
```

This allows queries like:

- ”How many elements < 50?” $\rightarrow$ `orderOfKey(root, 50)`
- Works seamlessly with duplicates if the tree stores counts per key.

9.2.4 Range Queries

Range queries extend `order_of_key`:
Count elements in range [L, R]:

```
int countInRange(const Node* root, int L, int R) {
    return orderOfKey(root, R + 1) - orderOfKey(root, L);
}
```

This works because:

- `order_of_key(R+1)` counts all elements R.
- `order_of_key(L)` counts all elements $<$ L.
- Difference yields the number in [L, R].

9.2.5 Handling Duplicates

Two strategies:

1. **Store frequency count in each node**
   - `count` field keeps track of duplicates.
   - Subtree size includes `count`.
2. **Insert duplicates consistently in right subtree**
   - Simpler but makes rank calculations less direct.

Example with frequency augmentation:

```
struct Node {
    int key, count, size;
    std::unique_ptr<Node> left, right;
    Node(int k) : key(k), count(1), size(1) {}
};

void updateSize(Node* n) {
    if (n) {
        n->size = n->count + getSize(n->left) + getSize(n->right);
    }
}
```

This ensures `order_of_key` and `kth_element` remain correct in presence of duplicates.

9.2.6 Performance Considerations

- **Time Complexity:** `O(log n)` for search, insertion, deletion, `kth_element`, and `order_of_key` in balanced augmented trees.
- **Space Overhead:** Storing `size` or `count` per node adds minimal overhead (usually one or two integers).
- **Balancing:** Without self-balancing, performance may degrade to `O(n)` due to skewness. In practice, augmentations are usually applied on **AVL trees, Red-Black trees, or Treaps**.
- **Standard Library:** `std::set` and `std::map` provide balanced trees, but not order statistics. Specialized structures (like GNU PBDS in GCC) offer `order_of_key` and `find_by_order`.

9.2.7 Practical Applications

- **Databases and Indexing:** Efficiently retrieve k-th record or count records within ranges.
- **Statistics:** Compute percentiles and quantiles in real time.
- **Scheduling:** Track tasks by deadlines and query positions.
- **Competitive Programming:** Many problems rely on order-statistics trees for rank queries.

9.2.8 Summary

Augmented BSTs extend ordinary trees by storing **subtree sizes and counts**, enabling powerful operations:

- `kth_element(k)`: retrieve the k-th smallest in `O(log n)`.
- `order_of_key(x)`: count elements smaller than `x` in `O(log n)`.
- Range queries ([L, R]) follow naturally from rank queries.

These augmentations make BSTs practical for **ranked datasets, real-time statistics, and range-based queries** while keeping the same asymptotic complexity. In modern C++, memory safety can be combined with these classic augmentations by using `unique_ptr` and well-structured updates.

9.3 Exercises — *kth Smallest, Interval Trees*

Exercises provide the critical step from theoretical understanding to hands-on mastery. This section emphasizes two widely useful augmentations: **finding the k-th smallest element in a BST** and **building interval trees for overlap queries**. Both exercises deepen the reader's grasp of augmented data structures, order statistics, and range problems.

9.3.1 Exercise: K-th Smallest Element in a BST

- **Problem Statement**

  Given a Binary Search Tree, implement an efficient function to return the k-th smallest element (1-indexed). The algorithm must run in logarithmic time on a balanced BST and correctly handle duplicates.

- **Hints**

  - Augment each node with `size` = total number of nodes in its subtree.
  - On traversing, compare `k` against the size of the left subtree.
  - Adjust `k` as you recurse into the right subtree.

- **Example Walkthrough**

  Tree:

```
      8
    /   \
   4     12
  / \   /  \
 2   6 10  14
```

  - `k = 4`:
    - Left size of root (8) = 3 (nodes 2,4,6).
    - Since `k = 4 = left_size + 1`, answer = 8.

- **Implementation Sketch in C++**

```cpp
#include <memory>
#include <iostream>

struct Node {
    int key;
    int size;  // subtree size
    std::unique_ptr<Node> left, right;

    Node(int k) : key(k), size(1) {}
};

int getSize(const std::unique_ptr<Node>& node) {
    return node ? node->size : 0;
}

void updateSize(Node* node) {
    if (node) {
        node->size = 1 + getSize(node->left) + getSize(node->right);
    }
}

const Node* kthSmallest(const Node* root, int k) {
    if (!root) return nullptr;
    int leftSize = getSize(root->left);
    if (k == leftSize + 1) return root;
    if (k <= leftSize) return kthSmallest(root->left.get(), k);
```

```
    return kthSmallest(root->right.get(), k - leftSize - 1);
}
```

- **Challenge Extensions**
  - Modify the code to handle duplicate values by introducing a `count` field per node.
  - Extend the function to return all k smallest elements up to index `k`.
  - Compare performance with an in-order traversal approach.

9.3.2 Exercise: Interval Trees

- **Problem Statement**

  Design an **Interval Tree** based on a BST that stores intervals `[l, r]`. Support queries of the form:
  - "Does any interval overlap with `[L, R]`?"
  - "Return all intervals overlapping with `[L, R]`."

- **Conceptual Overview**
  - Each node stores an interval `[l, r]`.
  - The node is augmented with `maxEnd` = maximum `r` among all intervals in its subtree.
  - Overlap detection relies on checking whether the current interval intersects `[L, R]` and pruning subtrees based on `maxEnd`.

- **Example Walkthrough**

  Intervals: `[15,20]`, `[10,30]`, `[17,19]`, `[5,20]`, `[12,15]`, `[30,40]`
  Query: `[14,16]`

  – The algorithm traverses nodes and finds `[10,30]` and `[12,15]` as overlaps.

- **Implementation Sketch in C++**

```
struct Interval {
    int low, high;
};

struct IntervalNode {
    Interval interval;
    int maxEnd;  // maximum high endpoint in subtree
    std::unique_ptr<IntervalNode> left, right;

    IntervalNode(int l, int h) : interval{l, h}, maxEnd(h) {}
};

int getMaxEnd(const std::unique_ptr<IntervalNode>& node) {
    return node ? node->maxEnd : INT_MIN;
}

void updateMaxEnd(IntervalNode* node) {
    if (node) {
        node->maxEnd = std::max({node->interval.high,
                                 getMaxEnd(node->left),
                                 getMaxEnd(node->right)});
    }
}
```

```
bool doOverlap(const Interval& a, const Interval& b) {
    return (a.low <= b.high && b.low <= a.high);
}

// Query if any overlap exists
const IntervalNode* searchOverlap(const IntervalNode* root, const Interval&
↪  query) {
    if (!root) return nullptr;

    if (doOverlap(root->interval, query)) return root;

    if (root->left && root->left->maxEnd >= query.low)
        return searchOverlap(root->left.get(), query);

    return searchOverlap(root->right.get(), query);
}
```

- **Challenge Extensions**
  - Modify the query to return **all overlapping intervals** instead of just one.
  - Implement **deletion** of intervals and ensure `maxEnd` is correctly maintained.
  - Compare performance with brute-force scanning of all intervals.

9.3.3 Testing and Benchmarking

Students should verify correctness and efficiency by:

- Creating trees with 1e5 random keys or intervals.
- Validating `kthSmallest` against sorted arrays.

- Comparing interval queries against naive linear scans.

Tools like **GoogleTest** for correctness and **Google Benchmark** for timing will help ensure implementations are robust.

9.3.4 Summary

These exercises illustrate the power of **augmenting trees**:

- `kthSmallest` demonstrates order-statistics queries using subtree sizes.
- Interval trees highlight range overlap detection using maximum endpoints.

Both are classic examples of how BSTs can be extended into **problem-specific data structures** with only modest metadata augmentation, while retaining logarithmic efficiency.

Chapter 10

Self-Balancing Trees (AVL, Red-Black)

10.1 AVL Rotations in C++ — Code Walkthrough

AVL trees are the earliest form of **self-balancing binary search trees**, ensuring that the height difference (balance factor) between the left and right subtrees of any node never exceeds one. This invariant guarantees logarithmic time for searches, insertions, and deletions. The heart of maintaining balance lies in **rotations** — structured pointer manipulations that rebalance the tree while preserving the Binary Search Tree (BST) ordering property.

10.1.1 Balance Factor and Rotation Trigger

- **Balance Factor (BF):**
  Defined as `height(left) - height(right)` for a given node.

  - If `BF ∈ {-1, 0, 1}`, the node is balanced.
  - If `BF < -1` → right-heavy imbalance.
  - If `BF > 1` → left-heavy imbalance.

- **Imbalance Cases:**
  - **Left-Left (LL):** Single right rotation.
  - **Right-Right (RR):** Single left rotation.
  - **Left-Right (LR):** Left rotation on left child, then right rotation.
  - **Right-Left (RL):** Right rotation on right child, then left rotation.

10.1.2 Node Structure in Modern C++

We use `std::unique_ptr` for automatic memory management to prevent leaks. Heights are cached for efficiency.

```
#include <memory>
#include <algorithm>

struct Node {
    int key;
    int height;
    std::unique_ptr<Node> left, right;

    Node(int k) : key(k), height(1) {}
};

int getHeight(const std::unique_ptr<Node>& n) {
    return n ? n->height : 0;
}
```

```
int getBalance(const std::unique_ptr<Node>& n) {
    return n ? getHeight(n->left) - getHeight(n->right) : 0;
}

void updateHeight(Node* n) {
    if (n) {
        n->height = 1 + std::max(getHeight(n->left), getHeight(n->right));
    }
}
```

10.1.3 Single Rotations

- **Right Rotation (for LL imbalance)**

```
std::unique_ptr<Node> rightRotate(std::unique_ptr<Node> y) {
    auto x = std::move(y->left);          // x becomes new root
    auto T2 = std::move(x->right);        // T2 will be moved

    x->right = std::move(y);              // old root y becomes right child of
    ↪ x
    x->right->left = std::move(T2);       // reattach T2

    updateHeight(x->right.get());
    updateHeight(x.get());
    return x;                             // return new root
}
```

- **Left Rotation (for RR imbalance)**

```
std::unique_ptr<Node> leftRotate(std::unique_ptr<Node> x) {
    auto y = std::move(x->right);          // y becomes new root
    auto T2 = std::move(y->left);          // T2 will be moved

    y->left = std::move(x);                // old root x becomes left child of y
    y->left->right = std::move(T2);        // reattach T2

    updateHeight(y->left.get());
    updateHeight(y.get());
    return y;                              // return new root
}
```

10.1.4 Double Rotations

- **Left-Right (LR) Rotation**
  - First rotate left on the left child.
  - Then rotate right on the root.

```
std::unique_ptr<Node> leftRightRotate(std::unique_ptr<Node> node) {
    node->left = leftRotate(std::move(node->left));
    return rightRotate(std::move(node));
}
```

- **Right-Left (RL) Rotation**
  - First rotate right on the right child.
  - Then rotate left on the root.

```
std::unique_ptr<Node> rightLeftRotate(std::unique_ptr<Node> node) {
    node->right = rightRotate(std::move(node->right));
    return leftRotate(std::move(node));
}
```

10.1.5 Rotation Integration in Insertions

During insertions, we check the balance factor at each step and apply the appropriate rotation.

```
std::unique_ptr<Node> insert(std::unique_ptr<Node> node, int key) {
    if (!node) return std::make_unique<Node>(key);

    if (key < node->key)
        node->left = insert(std::move(node->left), key);
    else if (key > node->key)
        node->right = insert(std::move(node->right), key);
    else
        return node; // duplicates not allowed

    updateHeight(node.get());

    int balance = getBalance(node);

    // Left Left
    if (balance > 1 && key < node->left->key)
        return rightRotate(std::move(node));

    // Right Right
    if (balance < -1 && key > node->right->key)
        return leftRotate(std::move(node));
```

```
    // Left Right
    if (balance > 1 && key > node->left->key)
        return leftRightRotate(std::move(node));

    // Right Left
    if (balance < -1 && key < node->right->key)
        return rightLeftRotate(std::move(node));

    return node; // unchanged if balanced
}
```

10.1.6 Walkthrough Example

Insert sequence: `10, 20, 30`

- Insert 10 → root = 10. Balanced.
- Insert 20 → root = 10, right child = 20. Balanced.
- Insert 30 → root = 10, right-heavy (BF = -2).
  → Apply **Left Rotation** at root.
  → New root = 20, left child = 10, right child = 30.

Insert sequence: `30, 10, 20`

- Insert 30 → root = 30.
- Insert 10 → root = 30, left child = 10.
- Insert 20 → root = 30, left child = 10 (with right child 20).
  → root BF = +2 (left-heavy), left child BF = -1.

→ Case = **Left-Right (LR)**.
→ Rotate left at 10, then rotate right at 30.
→ Balanced root = 20, left = 10, right = 30.

10.1.7 Key Insights

- **Rotations are purely pointer manipulations**; no keys are reordered beyond local restructuring.
- **Double rotations** can always be reduced to two single rotations.
- **Smart pointers** in modern C++ make ownership clear but require careful use of `std::move`.
- **Heights must be updated bottom-up** after every rotation.

10.1.8 Exercises

1. Implement deletion in an AVL tree and test rebalancing.
2. Instrument the code to count the number of rotations during insertion of `n` random keys.
3. Compare AVL insertions against unbalanced BSTs in terms of search path length.

10.2 Red-Black Tree Principles and Relation to `std::map` / `std::set`

Red-Black Trees (RBTs) are a type of **self-balancing binary search tree** widely used in practice because they provide **near-optimal logarithmic performance** while being simpler to maintain than AVL trees in terms of insertion and deletion balancing. They form the underlying implementation for standard C++ associative containers like `std::map` and `std::set`.

10.2.1 Red-Black Tree Properties

A Red-Black Tree enforces the following **five properties**:

1. **Node Color:** Each node is either **red** or **black**.
2. **Root Property:** The root is always black.
3. **Leaf Property:** All leaves (nullptr or sentinel nodes) are black.
4. **Red Property:** A red node cannot have a red child (no consecutive reds).
5. **Black-Height Property:** Every path from a node to its descendant leaves contains the same number of black nodes.

These properties ensure that the longest path from root to leaf is **at most twice as long** as the shortest path, guaranteeing **O(log n)** height.

10.2.2 Core Operations and Rebalancing

Red-Black trees maintain balance through **color flips and rotations** during insertion and deletion.

- **a) Insertion**
  - Insert new nodes as **red** to avoid violating the black-height property.
  - If the parent is black → done.
  - If the parent is red → rebalance using:
    - **Recoloring** (if uncle is red).
    - **Rotation(s)** (single or double) if uncle is black.
- **b) Deletion**
  - More complex than insertion due to potential **double-black violations**.
  - Requires a combination of:
    - Recoloring.
    - Rotations to move black height up the tree.
  - Guarantees that tree height remains logarithmic.

10.2.3 Relation to `std::map` and `std::set`

The C++ Standard Library uses **Red-Black Trees** for all ordered associative containers:

Container	Behavior
`std::set`	Stores unique keys in order.
`std::multiset`	Stores keys in order, allows duplicates.
`std::map`	Stores key-value pairs in order by key, unique keys.
`std::multimap`	Stores key-value pairs in order, allows duplicate keys.

Key points:

- Insertion, lookup, and deletion are all **O(log n)** due to RBT properties.
- Self-balancing ensures consistent performance regardless of input order.
- Internal nodes maintain **pointers, key, value (for map), color**, and sometimes a parent pointer to facilitate rotations.

10.2.4 C++ Implementation Highlights

While the full RBT implementation is complex, the essential structure includes:

```
enum class Color { RED, BLACK };

template <typename Key, typename Value>
struct RBNode {
    Key key;
    Value value;
    Color color;
    RBNode* parent;
    std::unique_ptr<RBNode> left, right;

    RBNode(const Key& k, const Value& v, Color c)
        : key(k), value(v), color(c), parent(nullptr) {}
};
```

- `parent` pointer facilitates upward traversal during rotations.
- `unique_ptr` ensures proper memory management for left and right subtrees.
- Rotations and recoloring preserve BST ordering and Red-Black properties.

10.2.5 Rotations in Red-Black Trees

- **Left rotation** and **right rotation** are identical to AVL rotations, except they may also adjust **colors**.
- Rotations combined with recoloring resolve violations of Red-Black properties.

Example Scenario:

- Inserting a red node under a red parent with a red uncle triggers **recoloring**.
- If the uncle is black, a **rotation** (single or double) is performed, followed by recoloring.

10.2.6 Comparison with AVL Trees

Feature	AVL Tree	Red-Black Tree
Height	Strictly log(n)	$2 \times \log(n)$
Insert/Delete Complexity	Slightly slower due to stricter balancing	Slightly faster due to relaxed balancing
Rotations per insertion	Can require multiple rotations	Usually fewer rotations
Use Case	Lookup-heavy scenarios	Insertion-heavy / standard libraries

Summary:

- AVL trees provide **faster lookups** because they are more strictly balanced.

- Red-Black trees offer **faster insertions and deletions** on average, which is why they underpin `std::map` and `std::set`.

10.2.7 Practical Takeaways

1. **Always expect O(log n) operations** with ordered associative containers.

2. `std::map` and `std::set` abstract away RBT complexity, but understanding the underlying structure helps in:

   - Predicting performance in edge cases.
   - Debugging insertion/deletion anomalies in large datasets.

3. Rotations and recoloring are the **key operations** ensuring the tree remains balanced without violating BST properties.

4. Modern C++ encourages safe memory management via `unique_ptr` and references, but the **parent pointer** is often retained for efficient upward traversal.

10.3 Exercises — Implement an AVL with Unit Tests; Compare Against `std::set` Performance

Hands-on exercises are essential to solidify understanding of **self-balancing trees**. This section guides the reader through implementing an **AVL tree from scratch**, writing **unit tests** for correctness, and **benchmarking** against `std::set` to observe real-world performance differences.

10.3.1 Exercise 1: Implement an AVL Tree

- **Objectives**
  - Create a C++ class representing an **AVL tree** with:
    - Insertion
    - Deletion
    - Search (`contains`)
    - Height balancing via rotations (LL, RR, LR, RL)
  - Ensure proper **memory safety** using `std::unique_ptr` for child nodes.
- **Suggested Interface**

```cpp
#include <memory>
#include <iostream>

class AVLTree {
    struct Node {
        int key;
        int height;
        std::unique_ptr<Node> left, right;
        Node(int k) : key(k), height(1) {}
    };
    std::unique_ptr<Node> root;

public:
    void insert(int key);
    void remove(int key);
    bool contains(int key) const;
    void inorder() const;  // optional, for testing
};
```

- **Implementation Hints**
  - Update **subtree height** after insertion/deletion.
  - Compute **balance factor** to detect imbalance.
  - Apply the appropriate **rotation** for each imbalance case.
  - Use **helper functions** for rotations and height updates.

10.3.2 Exercise 2: Unit Testing

Unit testing ensures the AVL tree maintains **correct BST ordering** and **balance properties**.

- **Recommended Frameworks**
  - **GoogleTest (gtest)**
  - **Catch2**
- **Example Tests**

```
#include <gtest/gtest.h>
#include "avl_tree.h"

TEST(AVLTreeTest, InsertAndContains) {
    AVLTree tree;
    tree.insert(10);
    tree.insert(20);
    tree.insert(5);

    EXPECT_TRUE(tree.contains(10));
    EXPECT_TRUE(tree.contains(20));
```

```
    EXPECT_TRUE(tree.contains(5));
    EXPECT_FALSE(tree.contains(15));
}

TEST(AVLTreeTest, BalanceAfterInsertions) {
    AVLTree tree;
    for (int k : {30, 20, 40, 10, 25, 35, 50}) {
        tree.insert(k);
    }
    // Check in-order traversal
    std::vector<int> result;
    tree.inorder(result);
    EXPECT_EQ(result, std::vector<int>({10, 20, 25, 30, 35, 40, 50}));
}
```

- **Exercise Goals**
  - Verify that all rotations maintain **BST properties**.
  - Check that **balance factor** of each node is in `[-1, 1]`.
  - Validate **deletion scenarios**, including nodes with 0, 1, or 2 children.

10.3.3 Exercise 3: Performance Comparison Against `std::set`

After implementing a correct AVL tree, measure **real-world performance**:

- **Setup**
  - Generate a large dataset of `n` unique integers (e.g., 1e5 to 1e6).
  - Insert the same keys into your **AVL tree** and a **`std::set<int>`**.
  - Measure the time for:

    * Bulk insertion
    * Search operations
    * Deletion operations

- **Example Using `std::chrono`**

```
#include <chrono>
#include <set>
#include "avl_tree.h"

auto start = std::chrono::high_resolution_clock::now();
for (int key : keys) avl.insert(key);
auto end = std::chrono::high_resolution_clock::now();
std::cout << "AVL insertion time: "
          << std::chrono::duration_cast<std::chrono::milliseconds>(end -
          ↪  start).count()
          << " ms\n";

start = std::chrono::high_resolution_clock::now();
std::set<int> s;
for (int key : keys) s.insert(key);
end = std::chrono::high_resolution_clock::now();
std::cout << "std::set insertion time: "
          << std::chrono::duration_cast<std::chrono::milliseconds>(end -
          ↪  start).count()
          << " ms\n";
```

- **Expected Observations**
    - Both AVL and `std::set` have **O(log n)** insertion, deletion, and search.
    - `std::set` is highly optimized in standard libraries and often performs slightly faster due to:

    * Compiler optimizations
    * Efficient memory allocation
    * Low-level pointer operations
  - Custom AVL tree is valuable for:
    * Learning rotations and balance maintenance
    * Extending nodes with additional metadata (augmented trees)

10.3.4 Optional Extensions

1. **Augment AVL nodes** to store **subtree sizes** → implement `kth_smallest`.
2. Compare **memory usage** between `std::set` and AVL tree using large datasets.
3. Test **edge cases**: inserting sorted data, random data, and duplicates.

10.3.5 Summary

This exercise section reinforces:

- The **mechanics of AVL rotations** and balance maintenance.
- **Unit testing** as an essential tool for validating correctness.
- **Practical benchmarking** to understand performance characteristics relative to `std::set`.

By completing these exercises, readers gain both **theoretical understanding** and **hands-on experience**, preparing them for more advanced self-balancing structures like **Red-Black Trees** or **Augmented AVL Trees**.

Chapter 11

B-Trees and External-Memory Structures

11.1 B-Tree Node Layout, Block I/O Considerations (C++ Structures for Disk-Backed Nodes)

B-Trees are **multi-way search trees** specifically designed for **external memory storage**, such as disks or SSDs. They optimize **I/O efficiency** by minimizing the number of block reads/writes required for search, insertion, and deletion. Understanding node layout and how it maps to disk blocks is crucial for high-performance implementations.

11.1.1 B-Tree Node Structure

A **B-Tree of order `t`** (minimum degree) satisfies:

- Each node contains at most `2t-1` keys.

- Each node (except root) contains at least `t-1` keys.
- Internal nodes have **children pointers** equal to **number of keys + 1**.
- All leaf nodes appear at the same depth.
- **In-Memory Representation**

```
template <typename Key, typename Value, size_t t>
struct BTreeNode {
    size_t nKeys;                                  // current number of keys
    Key keys[2*t - 1];                             // keys array
    Value values[2*t - 1];                         // corresponding values
    std::unique_ptr<BTreeNode> children[2*t];    // child pointers
    bool leaf;                                     // true if node is leaf

    BTreeNode(bool isLeaf) : nKeys(0), leaf(isLeaf) {}
};
```

  - Using **fixed-size arrays** aligns well with disk-block layout.
  - `leaf` flag distinguishes between internal and leaf nodes.
  - `nKeys` tracks the current number of stored keys.
  - `children` array allows up to `2t` subtrees.

11.1.2 Disk Block Considerations

When designing **disk-backed B-Trees**, the node structure must align with **block I/O**:

- **Block Size Matching:**
  - Each node should fit into a single disk block (commonly 4KB or 8KB).

- Choosing `t` (minimum degree) is based on block size:

$$2t - 1 \approx \frac{\text{block\_size}}{\text{sizeof(Key + Value + pointer)}}$$

- **Contiguous Memory Layout:**
  - Keys and values are stored in contiguous arrays for fast block transfer.
  - Avoid pointers for on-disk children; instead, store **disk offsets** or **page IDs**.

```
struct DiskBTreeNode {
    bool leaf;
    uint16_t nKeys;
    Key keys[MAX_KEYS];
    Value values[MAX_KEYS];
    uint64_t childOffsets[MAX_KEYS + 1]; // disk positions instead of in-memory
    ↪  pointers
};
```

- On read/write, **serialize/deserialize** the node to/from disk.
- **Little-endian vs big-endian** consistency is important if disk data may move across platforms.

11.1.3 Advantages of This Layout

1. **Efficient I/O:**
   - Large fan-out reduces tree height → fewer disk accesses per operation.
2. **Cache-Friendly:**

- Contiguous arrays allow prefetching and reduced cache misses.

3. **Predictable Size:**

   - Each node occupies a fixed block, simplifying memory mapping and file offsets.

4. **Support for Large Datasets:**

   - B-Trees can scale to millions of keys because disk I/O dominates runtime, not in-memory traversal.

11.1.4 C++ Considerations

- **Smart Pointers vs Disk Offsets:**
  - In-memory nodes can use `std::unique_ptr` for safety.
  - Disk-backed nodes must use **offsets** or **IDs**, with a node cache mapping offsets to loaded objects.
- **Serialization Example:**

```
void writeNodeToDisk(const DiskBTreeNode& node, std::ofstream& file, uint64_t offset)
↪  {
    file.seekp(offset);
    file.write(reinterpret_cast<const char*>(&node), sizeof(node));
}

DiskBTreeNode readNodeFromDisk(std::ifstream& file, uint64_t offset) {
    DiskBTreeNode node;
    file.seekg(offset);
```

```
    file.read(reinterpret_cast<char*>(&node), sizeof(node));
    return node;
}
```

- **Alignment and Padding:**
  - Ensure `sizeof(DiskBTreeNode)` matches block size.
  - Use `#pragma pack` or alignas if necessary.

11.1.5 Summary

- **Node layout** determines **I/O efficiency**, crucial for large-scale storage.
- **Disk-backed B-Trees** store child references as offsets rather than pointers.
- **Fixed-size arrays** and careful memory alignment reduce unnecessary reads / writes.
- B-Trees remain the standard for database indexes, filesystem directories, and any external-memory structure requiring **predictable O(log n) access**.

11.2 Practical Uses — Simple On-Disk Key-Value Store Prototype

B-Trees are particularly well-suited for **disk-backed storage systems**, where minimizing disk I/O is essential. In this section, we illustrate a **prototype for a simple on-disk key-value store** using a B-Tree as the underlying index. This demonstrates how theoretical B-Tree concepts translate to practical, high-performance storage applications.

11.2.1 Design Overview

- **Goals**
  - Store key-value pairs on disk in **sorted order**.
  - Support efficient **insertions, lookups, and range queries**.
  - Minimize **disk reads and writes** per operation.
  - Prototype a **single-file store** without complex transaction handling.
- **Core Components**
  1. **Disk Storage Layer**
     - Stores serialized nodes in fixed-size blocks.
     - Provides `readNode(offset)` and `writeNode(node, offset)` operations.
  2. **B-Tree Index Layer**
     - Maintains nodes in memory while navigating or modifying the tree.
     - Uses **node offsets** instead of pointers for disk persistence.
  3. **Cache Layer (Optional)**

  - Maintains frequently accessed nodes in memory.
  - Reduces repeated disk accesses for hot keys.

11.2.2 Disk Node Structure

```
struct DiskNode {
    bool leaf;
    uint16_t nKeys;
    int keys[MAX_KEYS];
    int values[MAX_KEYS];              // simple integer values for prototype
    uint64_t childOffsets[MAX_KEYS+1]; // disk offsets of children
};
```

- Each node occupies a **single fixed-size block** (e.g., 4KB).
- `leaf` indicates if the node has children.
- `childOffsets` replace in-memory pointers to enable persistence.

11.2.3 Basic Operations

- **a) Lookup**
  - Start at the root node (offset stored in file header).
  - Load node from disk using `readNode(offset)`.
  - Perform **binary search** among node's keys to find target or child index.
  - Recursively follow **child offsets** until key is found or leaf is reached.

```
std::optional<int> findKey(uint64_t nodeOffset, int key) {
    DiskNode node = readNode(nodeOffset);
    int i = 0;
    while (i < node.nKeys && key > node.keys[i]) i++;
    if (i < node.nKeys && key == node.keys[i]) return node.values[i];
    if (node.leaf) return std::nullopt;
    return findKey(node.childOffsets[i], key);
}
```

- **b) Insertion**
  - Insert key into leaf node.
  - If node is full (`nKeys == MAX_KEYS`), **split the node**:
    1. Allocate a new disk block.
    2. Move half of the keys and values into the new node.
    3. Push the middle key up to the parent node.
  - Recursively handle parent splits up to the root.

```
void splitChild(DiskNode& parent, int idx, DiskNode& child, std::ofstream&
↪  file) {
    DiskNode newNode;
    newNode.leaf = child.leaf;
    newNode.nKeys = t - 1;
    // Copy second half of keys/values
    for (int j = 0; j < t-1; ++j) {
        newNode.keys[j] = child.keys[j+t];
        newNode.values[j] = child.values[j+t];
    }
    // Copy child pointers if not leaf
```

```
    if (!child.leaf) {
        for (int j = 0; j < t; ++j)
            newNode.childOffsets[j] = child.childOffsets[j+t];
    }
    child.nKeys = t - 1;
    // Write newNode to disk, update parent pointers, etc.
}
```

  - Ensure all **offsets are updated** in parent and disk.
  - Root may grow by creating a **new root node**.

- **c) Range Queries**
  - Load nodes recursively in-order.
  - Only load blocks containing keys in the requested range.
  - Minimizes I/O compared to scanning the entire file.

```
void rangeQuery(uint64_t nodeOffset, int low, int high, std::vector<int>&
↪  result) {
    DiskNode node = readNode(nodeOffset);
    int i = 0;
    while (i < node.nKeys && node.keys[i] < low) i++;
    for (; i < node.nKeys && node.keys[i] <= high; i++) {
        if (!node.leaf) rangeQuery(node.childOffsets[i], low, high, result);
        result.push_back(node.values[i]);
    }
    if (!node.leaf) rangeQuery(node.childOffsets[i], low, high, result);
}
```

11.2.4 Performance Considerations

- **Block Size Selection:**
  - Match node size to disk block size (commonly 4KB–8KB).
- **Caching:**
  - Keep frequently accessed nodes in memory to avoid repeated reads.
- **Serialization:**
  - Fixed-size arrays reduce serialization/deserialization overhead.
- **Minimized Disk Writes:**
  - Batch writes when splitting nodes or updating multiple nodes.

11.2.5 Extensions

1. Support **larger key and value types** (strings, structs).
2. Implement **persistence metadata** (file headers, free block lists).
3. Add **transaction support** or **journaling** for crash safety.
4. Benchmark performance against `std::map` or **SQLite in-memory tables**.

11.2.6 Summary

This exercise demonstrates the **practical application of B-Trees** for external-memory systems:

- Nodes map directly to **disk blocks** to minimize I/O.
- Child offsets replace pointers, enabling persistence.
- Insertion and lookup algorithms mirror **in-memory B-Trees** but account for **block I/O costs**.
- Even a simple prototype provides insight into **databases, key-value stores, and file systems**.

11.3 Exercise — Small B-Tree Library Sketch with Tests

This exercise guides readers through **creating a minimal C++ B-Tree library**, emphasizing **disk-backed storage, correctness testing, and modular design**. The goal is not production-level completeness but to give a **hands-on experience with B-Tree mechanics**, persistence, and test-driven development.

11.3.1 Library Structure

A minimal B-Tree library should include the following components:

1. **Node Representation**
    - In-memory nodes using `std::unique_ptr` or offsets for disk-backed storage.
    - Fixed-size key and value arrays for predictable layout.

```
template <typename Key, typename Value, size_t t>
struct BTreeNode {
    bool leaf;
    size_t nKeys;
    Key keys[2*t - 1];
    Value values[2*t - 1];
    std::unique_ptr<BTreeNode> children[2*t];

    BTreeNode(bool isLeaf) : leaf(isLeaf), nKeys(0) {}
};
```

1. **B-Tree Class Interface**

```
template <typename Key, typename Value, size_t t>
class BTree {
    std::unique_ptr<BTreeNode<Key, Value, t>> root;

public:
    BTree() : root(std::make_unique<BTreeNode<Key, Value, t>>(true)) {}
    void insert(const Key& key, const Value& value);
    std::optional<Value> search(const Key& key) const;
    void traverse() const; // For in-order verification
};
```

11.3.2 Key Operations

- **a) Search**
  - Recursively search within node keys.
  - Descend to child if key is not found and node is not a leaf.

```
std::optional<Value> searchNode(const BTreeNode<Key, Value, t>* node, const
↪  Key& key) const {
    size_t i = 0;
    while (i < node->nKeys && key > node->keys[i]) ++i;
    if (i < node->nKeys && key == node->keys[i]) return node->values[i];
    if (node->leaf) return std::nullopt;
    return searchNode(node->children[i].get(), key);
}
```

- **b) Insertion**

- Insert into leaf if space available.
- Split full child nodes during insertion.
- Promote middle key to parent when splitting.

```
void splitChild(BTreeNode<Key, Value, t>* parent, int idx) {
    auto& child = parent->children[idx];
    auto newNode = std::make_unique<BTreeNode<Key, Value, t>>(child->leaf);
    newNode->nKeys = t - 1;

    // Copy second half of keys and values
    for (size_t j = 0; j < t - 1; ++j) {
        newNode->keys[j] = child->keys[j + t];
        newNode->values[j] = child->values[j + t];
    }
    if (!child->leaf) {
        for (size_t j = 0; j < t; ++j)
            newNode->children[j] = std::move(child->children[j + t]);
    }
    child->nKeys = t - 1;

    // Insert newNode into parent's children
    for (size_t j = parent->nKeys; j > idx; --j)
        parent->children[j + 1] = std::move(parent->children[j]);
    parent->children[idx + 1] = std::move(newNode);

    // Move middle key to parent
    for (size_t j = parent->nKeys; j > idx; --j) {
        parent->keys[j] = parent->keys[j - 1];
        parent->values[j] = parent->values[j - 1];
    }
    parent->keys[idx] = child->keys[t - 1];
    parent->values[idx] = child->values[t - 1];
```

```
    parent->nKeys++;
}
```

11.3.3 Unit Testing

Unit tests ensure **correctness and balance** of the tree.

- **Recommended Frameworks**
  - **GoogleTest (gtest)** or **Catch2**
- **Example Tests**

```
#include <gtest/gtest.h>
#include "btree.h"

TEST(BTreeTest, InsertAndSearch) {
    BTree<int, int, 3> tree;
    tree.insert(10, 100);
    tree.insert(20, 200);
    tree.insert(5, 50);

    EXPECT_EQ(tree.search(10).value(), 100);
    EXPECT_EQ(tree.search(20).value(), 200);
    EXPECT_EQ(tree.search(5).value(), 50);
    EXPECT_FALSE(tree.search(15).has_value());
}

TEST(BTreeTest, TraversalCheck) {
    BTree<int, int, 3> tree;
    for (int k : {10, 20, 5, 6, 12}) tree.insert(k, k*10);
```

```
    std::vector<int> result;
    tree.traverse(result);
    EXPECT_EQ(result, std::vector<int>({5, 6, 10, 12, 20}));
}
```

11.3.4 Optional Extensions

1. **Disk-backed prototype**: replace `std::unique_ptr` with offsets and serialize nodes to file.
2. **Range queries**: implement `findRange(low, high)` using in-order traversal of relevant nodes.
3. **Bulk insert**: optimize for large datasets with fewer splits.
4. **Performance benchmark**: compare in-memory vs disk-backed B-Tree.

11.3.5 Learning Outcomes

By completing this exercise, readers will:

- Understand **B-Tree node structure and insertion logic**.
- Implement **splitting and promotion** of keys.
- Use **unit tests** to verify correctness and tree invariants.
- Prepare to extend the library into **disk-backed key-value stores** or **database indexing structures**.

Part IV

Graphs (Implemented in C++)

Chapter 12

Graph Representations in C++

12.1 Adjacency list/matrix, edge lists, compressed sparse row (CSR) for performance

12.1.1 Adjacency List

An adjacency list is a space-efficient way to represent a graph, especially when dealing with sparse graphs. In this representation:

- **Structure**: Each vertex has a list (or vector) containing its adjacent vertices.
- **Implementation**: In C++, this is typically implemented using an array of vectors or lists, where each index corresponds to a vertex, and the associated vector/list contains its neighbors.

Advantages:

- **Space Efficiency**: Requires O(V + E) space, where V is the number of vertices and E is the number of edges.

- **Efficient Iteration**: Allows quick iteration over the neighbors of a vertex.

Disadvantages:

- **Edge Lookup**: Checking if an edge exists between two vertices can take O(d) time, where d is the degree of the vertex.
- **Not Cache-Friendly**: Due to non-contiguous memory allocation, it may suffer from poor cache performance.

Example in C++:

```
#include <vector>
#include <iostream>

class Graph {
    int V;
    std::vector<int>* adjList;
public:
    Graph(int V) : V(V) {
        adjList = new std::vector<int>[V];
    }
    void addEdge(int v, int w) {
        adjList[v].push_back(w);
    }
    void printGraph() {
        for (int v = 0; v < V; ++v) {
            std::cout << "\n Vertex " << v << ":";
            for (int x : adjList[v])
                std::cout << " -> " << x;
            std::cout << std::endl;
        }
    }
};
```

12.1.2 Edge List

An edge list is a simple representation where all edges are stored in a list of pairs (or tuples):

- **Structure**: A list of pairs, each representing an edge between two vertices.
- **Implementation**: In C++, this is implemented using a `std::vector<std::pair<int, int>>` for an undirected graph.

Advantages:

- **Simple Structure**: Easy to implement and understand.
- **Space Efficient**: Requires O(E) space.

Disadvantages:

- **Inefficient Edge Lookup**: Checking if an edge exists takes O(E) time.
- **Inefficient Neighbor Lookup**: Finding all neighbors of a vertex requires O(E) time.

Example in C++:

```
#include <vector>
#include <iostream>

class Graph {
    std::vector<std::pair<int, int>> edges;
public:
    void addEdge(int u, int v) {
```

```
        edges.push_back(std::make_pair(u, v));
    }
    void printEdges() {
        for (auto& edge : edges)
            std::cout << "(" << edge.first << ", " << edge.second << ")\n";
    }
};
```

12.1.3 Compressed Sparse Row (CSR)

The CSR format is a memory-efficient representation, particularly suited for sparse graphs:

- **Structure**:
  - `values[]`: Stores the destination vertices of all edges.
  - `columns[]`: Stores the indices of the destination vertices in `values[]`.
  - `row_ptr[]`: Stores the index in `values[]` where each row starts.
- **Implementation**: In C++, this can be implemented using three `std::vector<int>` arrays: `values`, `columns`, and `row_ptr`.

Advantages:

- **Space Efficiency**: Requires O(V + E) space.
- **Cache-Friendly**: Stores data contiguously in memory, leading to better cache performance.
- **Efficient Traversal**: Allows efficient traversal of neighbors.

Disadvantages:

- **Complex Implementation**: More complex to implement and manage.
- **Fixed Structure**: Difficult to modify (e.g., adding/removing edges or vertices).

Example in C++:

```
#include <vector>
#include <iostream>

class Graph {
    std::vector<int> values;
    std::vector<int> columns;
    std::vector<int> row_ptr;
public:
    Graph(int V, int E) {
        values.reserve(E);
        columns.reserve(E);
        row_ptr.reserve(V + 1);
        row_ptr.push_back(0);
    }
    void addEdge(int u, int v) {
        values.push_back(v);
        columns.push_back(v);
        row_ptr[u + 1]++;
    }
    void finalize() {
        for (int i = 1; i < row_ptr.size(); ++i)
            row_ptr[i] += row_ptr[i - 1];
    }
    void printGraph() {
        for (int i = 0; i < row_ptr.size() - 1; ++i) {
```

```
            std::cout << "\n Vertex " << i << ":";
            for (int j = row_ptr[i]; j < row_ptr[i + 1]; ++j)
                std::cout << " -> " << values[j];
            std::cout << std::endl;
        }
    }
};
```

12.1.4 Comparison Table

Representation	Space Complexity	Edge Lookup	Neighbor Lookup	Cache Efficiency	Use Case
Adjacency List	O(V + E)	O(d)	O(d)	Moderate	Sparse graphs with frequent neighbor queries
Edge List	O(E)	O(E)	O(E)	Low	Simple graphs with few operations
CSR	O(V + E)	O(log E)	O(d)	High	Large sparse graphs with infrequent modifications

12.1.5 Choosing the Right Representation

- **Adjacency List**: Best suited for sparse graphs where you need to frequently access the neighbors of a vertex.
- **Edge List**: Ideal for simple graphs or when you need to perform operations like edge enumeration.
- **CSR**: Optimal for large, sparse graphs where memory efficiency and fast traversal are critical, and the graph structure remains static.

When implementing graph algorithms in C++, it's essential to choose the representation that aligns with your specific requirements, balancing between memory usage, access speed, and complexity.

12.2 Weighted graphs, directed/undirected, memory-oriented designs

12.2.1 Weighted Graphs

Weighted graphs assign a value (weight) to each edge, representing cost, distance, capacity, or other metrics. They are crucial for algorithms like **Dijkstra's shortest path**, **Prim's MST**, and **Bellman-Ford**.

12.2.1.1 Representation in C++

Weighted graphs are often represented using:

1. **Adjacency List of Pairs**

```cpp
#include <vector>
#include <utility> // for std::pair

class WeightedGraph {
    int V;
    std::vector<std::pair<int, double>>* adjList; // pair<destination, weight>
public:
    WeightedGraph(int V) : V(V) {
        adjList = new std::vector<std::pair<int, double>>[V];
    }

    void addEdge(int u, int v, double w) {
        adjList[u].push_back({v, w});
    }
```

```
    void printGraph() {
        for (int i = 0; i < V; ++i) {
            std::cout << "Vertex " << i << ":";
            for (auto &p : adjList[i])
                std::cout << " -> (" << p.first << ", " << p.second << ")";
            std::cout << "\n";
        }
    }
};
```

1. **Weighted Adjacency Matrix**

- Each cell `[i][j]` stores the weight of the edge from vertex `i` to vertex `j`, or a special value (e.g., `INF`) if no edge exists.

```
#include <vector>
#include <limits>

const double INF = std::numeric_limits<double>::infinity();

class WeightedMatrixGraph {
    int V;
    std::vector<std::vector<double>> adjMatrix;
public:
    WeightedMatrixGraph(int V) : V(V) {
        adjMatrix.assign(V, std::vector<double>(V, INF));
    }

    void addEdge(int u, int v, double w) {
        adjMatrix[u][v] = w;
    }
};
```

12.2.1.2 Advantages and Use Cases

- **Adjacency List**: Efficient for sparse graphs; O(V + E) memory; efficient neighbor traversal.
- **Adjacency Matrix**: Efficient edge lookup (O(1)); $O(V^2)$ memory; better for dense graphs.

12.2.2 Directed vs. Undirected Graphs

12.2.2.1 Directed Graphs (Digraphs)

- Edges have a direction; an edge `(u, v)` goes from `u` to `v` only.
- Represented in adjacency lists or matrices without duplicating edges.
- Example: Social network follow relationships.

C++ Representation:

```
adjList[u].push_back(v); // Only u -> v
```

12.2.2.2 Undirected Graphs

- Edges are bidirectional; edge `(u, v)` implies `(v, u)`.
- In adjacency list: add both directions.

```
adjList[u].push_back(v);
adjList[v].push_back(u);
```

12.2.2.3 Weighted Directed/Undirected Graphs

- Both types can carry weights by storing pairs `(destination, weight)` in adjacency lists or weight values in matrices.

12.2.3 Memory-Oriented Graph Designs

When designing large-scale graphs in C++, memory layout and efficiency are critical, especially for:

- Sparse graphs with millions of vertices.
- High-performance algorithms where cache locality matters.

12.2.3.1 Key Techniques

1. **Contiguous Storage**

- Use `std::vector` or CSR-style arrays for adjacency lists to improve cache performance.
- Avoid `std::list` due to pointer overhead and scattered memory.

1. **Compressed Sparse Row (CSR) for Weighted Graphs**

- Store all weights in a contiguous `weights[]` array aligned with `values[]` and `row_ptr[]`.

- Efficient for read-heavy algorithms like BFS/DFS, shortest path, or matrix-vector operations.

1. **Memory-Packed Edge Structures**

```
struct Edge {
    int u, v;
    double weight;
};
std::vector<Edge> edges; // Compact and cache-friendly
```

- Best for edge-centric algorithms like Kruskal's MST or Bellman-Ford.

1. **Custom Allocators**

- For extremely large graphs, using custom memory pools can reduce fragmentation and improve allocation/deallocation speed.

12.2.4 Performance and Trade-Offs

Feature	Adjacency List	Adjacency Matrix	CSR / Packed Memory
Space Complexity	O(V + E)	$O(V^2)$	O(V + E)
Edge Lookup	O(d)	O(1)	O(d) or O(log d)
Cache Efficiency	Moderate	High	High

Feature	Adjacency List	Adjacency Matrix	CSR / Packed Memory
Flexibility (Dynamic Graphs)	Easy	Hard	Moderate
Best Use Case	Sparse, dynamic	Dense graphs	Sparse, static, high-performance

12.2.4.1 Best Practices in Modern C++

- Prefer `std::vector` over raw arrays or `std::list` for adjacency lists.
- For weighted graphs, always use `std::pair` or structured types to store edges with weights.
- For extremely large graphs, CSR or edge arrays with contiguous storage maximize memory locality.
- Separate the graph structure from algorithms to improve modularity, maintainability, and cache efficiency.

Chapter 13

Traversal & Search

13.1 Depth-First Search (DFS) & Breadth-First Search (BFS) with Iterator-Based C++ APIs

Graph traversal is a fundamental technique for exploring the vertices and edges of a graph. Depth-First Search (DFS) and Breadth-First Search (BFS) are the most widely used algorithms, and modern C++ allows flexible implementations using iterators, STL containers, and generic programming.

13.1.1 Depth-First Search (DFS)

DFS explores a graph by starting at a source vertex and visiting as far as possible along each branch before backtracking. It can be implemented recursively or iteratively.

1. **Recursive DFS**

    - Uses the **call stack** to maintain the path.

- Elegant and concise but limited by stack depth in large graphs.

C++ Implementation with Iterator-Based API:

```
#include <vector>
#include <iostream>

class Graph {
    int V;
    std::vector<std::vector<int>> adjList;
public:
    Graph(int V) : V(V), adjList(V) {}

    void addEdge(int u, int v) {
        adjList[u].push_back(v);
    }

    void dfsRecursive(int start, std::vector<bool>& visited) const {
        visited[start] = true;
        std::cout << start << " ";

        for (auto it = adjList[start].cbegin(); it != adjList[start].cend();
        ↪  ++it)
            if (!visited[*it])
                dfsRecursive(*it, visited);
    }

    void dfs(int start) const {
        std::vector<bool> visited(V, false);
        dfsRecursive(start, visited);
    }
};
```

```
int main() {
    Graph g(5);
    g.addEdge(0, 1); g.addEdge(0, 2);
    g.addEdge(1, 3); g.addEdge(2, 4);

    g.dfs(0); // Output: 0 1 3 2 4
}
```

Notes on Iterator Usage:

- `cbegin()` / `cend()` provide const iterators over the adjacency list.
- This approach ensures **read-only access** to the graph during traversal.

2. **Iterative DFS**

- Uses an explicit **stack** instead of recursion.
- Useful for large graphs where recursion depth may be exceeded.

```
#include <stack>

void dfsIterative(int start) const {
    std::vector<bool> visited(V, false);
    std::stack<int> s;
    s.push(start);

    while (!s.empty()) {
        int v = s.top();
        s.pop();

        if (!visited[v]) {
```

```
            visited[v] = true;
            std::cout << v << " ";

            for (auto it = adjList[v].crbegin(); it != adjList[v].crend();
            ↪  ++it)
                if (!visited[*it])
                    s.push(*it);
        }
    }
}
```

Iterator Note:

- `crbegin()` / `crend()` traverse in reverse, ensuring the order of recursive DFS is mimicked.

13.1.2 Breadth-First Search (BFS)

BFS explores the graph level by level using a queue. It is commonly used for **shortest path in unweighted graphs**, connected component detection, and bipartiteness checking.

```
#include <queue>

void bfs(int start) const {
    std::vector<bool> visited(V, false);
    std::queue<int> q;

    visited[start] = true;
    q.push(start);
```

```
    while (!q.empty()) {
        int v = q.front(); q.pop();
        std::cout << v << " ";

        for (auto it = adjList[v].cbegin(); it != adjList[v].cend(); ++it)
            if (!visited[*it]) {
                visited[*it] = true;
                q.push(*it);
            }
    }
}
```

Iterator Notes:

- `cbegin()` / `cend()` provide safe traversal over neighbors.
- Using iterators allows generic traversal over different container types (`vector`, `list`, or `set`).

13.1.3 Iterator-Based API Design for Traversal

Modern C++ encourages **generic and iterator-friendly APIs**:

- **Template-Based Traversal**: Allow the adjacency container type to be generic.
- **Const-Correctness**: Use `cbegin()` / `cend()` for read-only operations.
- **Range-Based Loops**: Can simplify traversal while keeping iterators behind the scenes.

Example Template DFS Traversal:

```
template <typename AdjList>
void dfsTemplate(int v, std::vector<bool>& visited, const AdjList& adj) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto it = adj[v].cbegin(); it != adj[v].cend(); ++it)
        if (!visited[*it])
            dfsTemplate(*it, visited, adj);
}
```

- Works with any container type (`vector`, `list`, `deque`) for adjacency lists.
- Encourages reusability and abstraction in algorithm design.

13.1.4 Comparison: Recursive vs Iterative DFS vs BFS

Algorithm	Data Structure	Memory Usage	Use Case
DFS Recursive	Call stack	O(V)	Simple traversal, pathfinding
DFS Iterative	Explicit stack	O(V)	Large graphs, stack depth concerns
BFS	Queue	O(V)	Shortest path in unweighted graphs, level-order exploration

Performance Notes:

- Both DFS and BFS run in $\mathbf{O(V + E)}$ for adjacency list representation.
- BFS requires more memory if the branching factor is high (queue can grow large).

- Iterator-based implementations improve **readability, genericity, and const-correctness**.

13.1.5 Best Practices in Modern C++

1. Prefer **`std::vector`** for adjacency lists for cache locality and performance.
2. Use **iterators and range-based loops** for generic traversal.
3. Separate traversal algorithms from the graph structure to improve modularity.
4. For weighted graphs or other metadata, pass **additional structures** alongside DFS/BFS iterators.
5. Consider **iterative implementations** for very large graphs to avoid stack overflow.

13.2 Applications of Graph Traversal

Graph traversal algorithms like DFS and BFS are not just theoretical; they serve as building blocks for many practical applications in graph analysis. This section focuses on three fundamental applications:

13.2.1 Connected Components

A **connected component** in an undirected graph is a maximal set of vertices such that each pair of vertices is reachable from each other. For directed graphs, **strongly connected components (SCCs)** are the analogous concept where every vertex is reachable from every other vertex in the component.

- **Algorithm for Undirected Graphs**
  - Use DFS or BFS to explore from each unvisited vertex.
  - Each traversal marks all vertices in that component.
  - Count each traversal as a new connected component.

C++ Implementation Using DFS:

```
#include <vector>
#include <iostream>

class Graph {
    int V;
    std::vector<std::vector<int>> adjList;
public:
    Graph(int V) : V(V), adjList(V) {}
```

```
    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u); // undirected
    }

    void dfs(int v, std::vector<bool>& visited) const {
        visited[v] = true;
        for (int u : adjList[v])
            if (!visited[u])
                dfs(u, visited);
    }

    int connectedComponents() const {
        std::vector<bool> visited(V, false);
        int count = 0;
        for (int v = 0; v < V; ++v)
            if (!visited[v]) {
                dfs(v, visited);
                ++count;
            }
        return count;
    }
};
```

Notes:

- Complexity: O(V + E)
- Useful for analyzing **network connectivity**, clusters in social networks, and isolated subgraphs.

- **Strongly Connected Components (Directed Graphs)**

- **Kosaraju's Algorithm**:
  1. Perform DFS and push vertices to a stack in the order of finishing times.
  2. Transpose the graph.
  3. Pop vertices from the stack and perform DFS on the transposed graph; each DFS gives an SCC.
- Complexity: O(V + E)

13.2.2 Cycle Detection

Detecting cycles is crucial in graphs to prevent deadlocks, ensure valid dependencies, or check for invalid configurations.

1. **Graphs**
   - Use DFS while keeping track of the **parent vertex**.
   - If a visited vertex is encountered that is not the parent, a cycle exists.

C++ Implementation:

```
bool dfsCycle(int v, int parent, std::vector<bool>& visited) const {
    visited[v] = true;
    for (int u : adjList[v]) {
        if (!visited[u]) {
            if (dfsCycle(u, v, visited)) return true;
        } else if (u != parent) {
            return true; // Cycle detected
        }
    }
    return false;
}
```

```
bool hasCycle() const {
    std::vector<bool> visited(V, false);
    for (int v = 0; v < V; ++v)
        if (!visited[v])
            if (dfsCycle(v, -1, visited)) return true;
    return false;
}
```

2. **Directed Graphs**

- Use DFS with a **recursion stack** or **coloring**:
  - White: unvisited
  - Gray: currently in recursion stack
  - Black: fully visited
- A back-edge to a gray vertex indicates a cycle.

C++ Implementation (Coloring Method):

```
bool dfsCycleDirected(int v, std::vector<int>& color) const {
    color[v] = 1; // Gray
    for (int u : adjList[v]) {
        if (color[u] == 1) return true;   // Back-edge found
        if (color[u] == 0 && dfsCycleDirected(u, color)) return true;
    }
    color[v] = 2; // Black
    return false;
}

bool hasCycleDirected() const {
```

```
    std::vector<int> color(V, 0);
    for (int v = 0; v < V; ++v)
        if (color[v] == 0)
            if (dfsCycleDirected(v, color)) return true;
    return false;
}
```

- Complexity: O(V + E)
- Applications: Deadlock detection, scheduling validation, and dependency checks.

13.2.3 Topological Sort

Topological sorting is an ordering of vertices in a **directed acyclic graph (DAG)** such that for every edge `(u, v)`, `u` appears before `v` in the ordering.

1. **DFS-Based Topological Sort**

   (a) Perform DFS and push each vertex to a stack after all its neighbors are visited.

   (b) Pop vertices from the stack to get the topological order.

C++ Implementation:

```
#include <stack>

void topologicalSortUtil(int v, std::vector<bool>& visited, std::stack<int>&
↪  Stack) const {
    visited[v] = true;
```

```
    for (int u : adjList[v])
        if (!visited[u])
            topologicalSortUtil(u, visited, Stack);
    Stack.push(v);
}

std::vector<int> topologicalSort() const {
    std::stack<int> Stack;
    std::vector<bool> visited(V, false);

    for (int v = 0; v < V; ++v)
        if (!visited[v])
            topologicalSortUtil(v, visited, Stack);

    std::vector<int> order;
    while (!Stack.empty()) {
        order.push_back(Stack.top());
        Stack.pop();
    }
    return order;
}
```

2. **Kahn's Algorithm (BFS-Based)**

   - Compute **in-degree** of each vertex.
   - Repeatedly remove vertices with in-degree 0 and update in-degrees of neighbors.
   - Produces topological order using BFS.

Applications of Topological Sort

- Task scheduling and build systems (e.g., `make` or project compilation).
- Dependency resolution in package managers.
- Prerequisite ordering in courses or workflows.

13.2.4 Summary of Applications

Application	Graph Type	Algorithm / Approach	Complexity	Use Case
Connected Components	Undirected	DFS/BFS	O(V + E)	Network clustering, isolated subgraphs
Strongly Connected Components	Directed	Kosaraju / Tarjan	O(V + E)	Module detection, SCC analysis
Cycle Detection	Undirected	DFS with parent tracking	O(V + E)	Deadlock prevention, graph validation
Cycle Detection	Directed	DFS with recursion stack or coloring	O(V + E)	Dependency checking, scheduling validation

Application	Graph Type	Algorithm / Approach	Complexity	Use Case
Topological Sort	DAG	DFS + Stack / Kahn's BFS	O(V + E)	Task scheduling, dependency resolution

- **Best Practices in Modern C++**

  1. Use **iterators and range-based loops** for adjacency traversal.
  2. Keep graph algorithms **separate from the graph data structure**.
  3. Favor **generic templates** to allow different adjacency containers (`vector`, `list`, `set`).
  4. Use **explicit stacks or queues** for iterative solutions to avoid recursion limits in large graphs.
  5. Combine traversal algorithms with metadata arrays (`visited`, `color`, `parent`) for clarity and performance.

Chapter 14

Shortest Paths

14.1 Dijkstra's Algorithm

Dijkstra's algorithm is the canonical method for computing the **single-source shortest paths** in a weighted graph with **non-negative edge weights**. Modern C++ implementations can leverage **priority queues** to optimize performance, with subtle techniques that maximize efficiency.

14.1.1 Basic Dijkstra Algorithm

Algorithm Overview

1. Initialize distances from the source vertex `s` to all vertices as infinity (`INF`), except `s` itself, which is 0.

2. Use a **priority queue** (min-heap) to select the vertex with the smallest tentative distance.

3. Relax all outgoing edges of the selected vertex.

4. Repeat until all vertices are processed.

Naive Approach Complexity:

- Using a simple array for selection: $O(V^2)$
- Using a min-heap (priority queue): $O((V + E) \log V)$

14.1.2 Priority Queue Optimization

1. **Standard Approach Using `std::priority_queue`**

   C++'s `std::priority_queue` is a **max-heap by default**, but we can store pairs `(distance, vertex)` and reverse the comparison for a min-heap.

```
#include <vector>
#include <queue>
#include <iostream>
#include <limits>

const int INF = std::numeric_limits<int>::max();

class Graph {
    int V;
    std::vector<std::vector<std::pair<int,int>>> adjList; // pair<neighbor,
    ↪  weight>
public:
    Graph(int V) : V(V), adjList(V) {}

    void addEdge(int u, int v, int w) {
        adjList[u].push_back({v, w});
```

```
    }

    std::vector<int> dijkstra(int src) {
        std::vector<int> dist(V, INF);
        dist[src] = 0;

        using pii = std::pair<int,int>; // {distance, vertex}
        std::priority_queue<pii, std::vector<pii>, std::greater<pii>> pq;

        pq.push({0, src});

        while (!pq.empty()) {
            int u = pq.top().second;
            int d = pq.top().first;
            pq.pop();

            if (d > dist[u]) continue; // Outdated entry

            for (auto &[v, w] : adjList[u]) {
                if (dist[u] + w < dist[v]) {
                    dist[v] = dist[u] + w;
                    pq.push({dist[v], v});
                }
            }
        }

        return dist;
    }
};
```

Notes on Optimization:

(a) **Avoid unnecessary updates** by skipping outdated entries (`if (d > dist[u]) continue`).

(b) **Min-Heap via `std::greater`**: transforms the default max-heap into a min-heap.

2. **Using Custom Comparators**

   For more advanced cases, like **dynamic priority changes** or sorting by secondary criteria:

```
struct Node {
    int vertex;
    int distance;
    bool operator>(const Node& other) const {
        return distance > other.distance;
    }
};

std::priority_queue<Node, std::vector<Node>, std::greater<Node>> pq;
```

   - This approach allows attaching additional metadata (e.g., parent pointers, edge types) to nodes.

3. **Pair Optimization: Lazy vs. Indexed Heap**

   - **Lazy Dijkstra**: Insert multiple entries for the same vertex; skip outdated ones (as shown above). Simple, effective, O((V + E) log V).
   - **Indexed Heap / Decrease-Key**: Maintain a heap that supports `decrease-key` to avoid duplicates. More complex, but reduces total operations; requires custom heap or external library.

14.1.3 Iterators and Modern C++ Features

- Using **range-based loops** and **structured bindings** improves readability:

```
for (auto &[v, w] : adjList[u]) {
    if (dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
        pq.push({dist[v], v});
    }
}
```

- `auto &` avoids unnecessary copies.
- `structured bindings` (`[v, w]`) make edge access expressive.

14.1.4 Performance Considerations

Feature	Complexity	Notes
Basic Array Selection	$O(V^2)$	Suitable for dense graphs
Min-Heap via `std::priority_queue`	$O((V + E) \log V)$	Standard approach
Indexed Heap / Fibonacci Heap	$O(E + V \log V)$	More efficient for large sparse graphs; complex implementation
Lazy Insertions	$O((V + E) \log V)$	Simple and effective in practice

Common `std::priority_queue` Hacks

1. **Store negative weights**: Convert max-heap to min-heap without using `std::greater`.

2. **Avoid custom comparator overhead**: Use `std::pair` instead of a struct for simplicity.

3. **Skip outdated entries**: Use a check `if (d > dist[u]) continue` to prevent extra relaxations.

4. **Use vector reserve**: Preallocate adjacency vectors for performance in large graphs.

14.1.5 Example Usage

```
int main() {
    Graph g(5);
    g.addEdge(0, 1, 10);
    g.addEdge(0, 4, 5);
    g.addEdge(1, 2, 1);
    g.addEdge(4, 1, 3);
    g.addEdge(4, 3, 2);
    g.addEdge(2, 3, 4);

    std::vector<int> dist = g.dijkstra(0);
    for (int i = 0; i < dist.size(); ++i)
        std::cout << "Distance from 0 to " << i << ": " << dist[i] << "\n";
}
```

Output:

```
Distance from 0 to 0: 0
Distance from 0 to 1: 8
Distance from 0 to 2: 9
Distance from 0 to 3: 7
Distance from 0 to 4: 5
```

Best Practices in Modern C++

1. Use `std::priority_queue` with **structured bindings** for clean and readable code.
2. Prefer **lazy insertion** over implementing decrease-key for simplicity.
3. Reserve memory in adjacency lists and vectors for large graphs.
4. Always **guard against outdated entries** to maintain correctness.
5. Consider using **custom node structs** for more complex scenarios (e.g., path reconstruction or multi-criteria shortest paths).

14.2 Bellman-Ford, SPFA Notes, and C++ Pitfalls

While Dijkstra's algorithm is efficient for graphs with non-negative weights, many applications involve **negative edge weights**. Bellman-Ford and its optimization SPFA (Shortest Path Faster Algorithm) address this, but careful C++ implementation is essential to avoid common pitfalls.

14.2.1 Bellman-Ford Algorithm

The **Bellman-Ford algorithm** computes single-source shortest paths for graphs with **negative edge weights** and detects **negative cycles**.

1. **Algorithm Overview**

   (a) Initialize distances from the source `s` to all vertices as infinity (`INF`), except `s` itself which is 0.

   (b) Relax all edges **V-1 times**.

   (c) Check for negative cycles: if any edge can still be relaxed, the graph contains a negative cycle.

   Time Complexity: O(V × E)
   Space Complexity: O(V)

2. **C++ Implementation**

```cpp
#include <vector>
#include <iostream>
#include <limits>

struct Edge { int u, v, w; };
```

```cpp
std::vector<int> bellmanFord(int V, const std::vector<Edge>& edges, int src) {
    const int INF = std::numeric_limits<int>::max();
    std::vector<int> dist(V, INF);
    dist[src] = 0;

    for (int i = 0; i < V - 1; ++i) {
        for (auto& e : edges) {
            if (dist[e.u] != INF && dist[e.u] + e.w < dist[e.v]) {
                dist[e.v] = dist[e.u] + e.w;
            }
        }
    }

    // Negative cycle detection
    for (auto& e : edges) {
        if (dist[e.u] != INF && dist[e.u] + e.w < dist[e.v]) {
            throw std::runtime_error("Graph contains a negative-weight cycle");
        }
    }

    return dist;
}
```

Notes:

- `dist[e.u] != INF` is essential to avoid integer overflow.
- Throws an exception if a negative cycle exists.
- Uses a **struct Edge** for clarity and cache efficiency.

14.2.2 SPFA (Shortest Path Faster Algorithm)

SPFA is an optimization over Bellman-Ford:

- Uses a **queue** to relax only vertices whose distance was updated.
- Often faster in practice for sparse graphs, but worst-case complexity remains O(V × E).

C++ Implementation

```cpp
#include <queue>

std::vector<int> spfa(int V, const std::vector<Edge>& edges, int src) {
    const int INF = std::numeric_limits<int>::max();
    std::vector<int> dist(V, INF);
    std::vector<int> inQueue(V, false);
    dist[src] = 0;

    std::queue<int> q;
    q.push(src);
    inQueue[src] = true;

    while (!q.empty()) {
        int u = q.front(); q.pop();
        inQueue[u] = false;

        for (auto& e : edges) {
            if (e.u == u && dist[u] != INF && dist[u] + e.w < dist[e.v]) {
                dist[e.v] = dist[u] + e.w;
                if (!inQueue[e.v]) {
                    q.push(e.v);
```

```
                    inQueue[e.v] = true;
                }
            }
        }

    return dist;
}
```

Notes:

- SPFA may **dequeue a vertex multiple times**, but avoids unnecessary relaxations.
- Using a boolean array `inQueue` prevents multiple duplicate entries.

14.2.3 C++ Pitfalls to Avoid

1. **Integer Overflow**

```
dist[e.u] + e.w < dist[e.v] // If dist[e.u] == INF, can overflow
```

- Always check `dist[e.u] != INF` before performing additions.

1. **Negative Cycle Detection**

- Forgetting to detect negative cycles can result in infinite loops or invalid distances.

1. **Using `std::priority_queue` with negative edges**

- Dijkstra cannot handle negative edges, using a priority queue can produce incorrect results.

1. **Queue Management in SPFA**

- Forgetting `inQueue` boolean check can lead to **excessive queue growth**, slowing the algorithm or exceeding memory limits.

1. **Floating-Point Weights**

- When using `double` or `float`, beware of **rounding errors**; consider epsilon comparisons instead of strict equality.

1. **Edge Representation**

- Use **struct Edge** for clarity, avoid storing separate parallel arrays unless memory-critical.

1. **Large Graphs**

- Prefer `reserve()` for adjacency lists and edge vectors to reduce dynamic allocations.

14.2.4 Comparison of Bellman-Ford vs SPFA

Feature	**Bellman-Ford**	**SPFA**
Time Complexity	O(V × E)	O(V × E) worst-case, often faster in practice

Feature	Bellman-Ford	SPFA
Space Complexity	O(V + E)	O(V + E)
Negative Cycle Detection	Easy	Easy (with counter/queue cycle check)
Suitable for Dense Graphs	Yes	Often better for sparse graphs
Implementation Complexity	Simple	Slightly more complex due to queue management

14.2.5 Best Practices in Modern C++

1. **Use `struct Edge` or tuple for clarity and memory locality**.
2. **Always guard against integer overflow** when relaxing edges.
3. **Use `std::queue` with `inQueue` array** in SPFA to avoid duplicate entries.
4. **Separate graph data structure from algorithms** for modularity.
5. For extremely large graphs, **consider memory-efficient representations** (CSR, adjacency list with edge structs).
6. When using floating-point weights, implement comparisons with tolerance to avoid precision issues.

14.3 A* Algorithm with C++ Heuristics and Custom Comparators

The **A* algorithm** is a widely used **best-first search algorithm** for finding the shortest path in a graph, combining the benefits of **Dijkstra's algorithm** with a **heuristic function** that estimates the cost to reach the goal. Its efficiency depends heavily on the choice of heuristic and the correct use of priority queues.

14.3.1 Algorithm Overview

A* maintains the following:

- `g(n)`: the exact cost from the start node to node `n`.
- `h(n)`: a heuristic estimate of the cost from `n` to the goal.
- `f(n) = g(n) + h(n)`: total estimated cost of a solution through `n`.

The algorithm selects the node with the **lowest `f(n)`** from the priority queue at each step.

Requirements for Heuristics:

- **Admissible**: never overestimates the true cost (ensures optimality).
- **Consistent / Monotone**: `h(n)` `cost(n, n') + h(n')` for every edge `(n, n')`.

14.3.2 Basic C++ Implementation Using `std::priority_queue`

```
#include <vector>
#include <queue>
#include <unordered_map>
#include <iostream>
#include <cmath>
#include <limits>

struct Node {
    int vertex;
    double g; // cost from start
    double f; // estimated total cost
    bool operator>(const Node& other) const { return f > other.f; }
};

class Graph {
    int V;
    std::vector<std::vector<std::pair<int,double>>> adjList; // {neighbor, weight}
public:
    Graph(int V) : V(V), adjList(V) {}

    void addEdge(int u, int v, double w) {
        adjList[u].push_back({v, w});
    }

    // Example heuristic: Euclidean distance (requires positions)
    double heuristic(int u, int goal, const std::vector<std::pair<double,double>>&
    ↪  pos) {
        double dx = pos[u].first - pos[goal].first;
        double dy = pos[u].second - pos[goal].second;
        return std::sqrt(dx*dx + dy*dy);
    }
```

```
    std::vector<int> aStar(int start, int goal, const
    ↪  std::vector<std::pair<double,double>>& pos) {
        std::vector<double> gScore(V, std::numeric_limits<double>::infinity());
        std::vector<int> cameFrom(V, -1);
        gScore[start] = 0;

        std::priority_queue<Node, std::vector<Node>, std::greater<Node>> pq;
        pq.push({start, 0, heuristic(start, goal, pos)});

        while (!pq.empty()) {
            Node current = pq.top(); pq.pop();
            int u = current.vertex;
            if (u == goal) break;

            for (auto &[v, w] : adjList[u]) {
                double tentative_g = gScore[u] + w;
                if (tentative_g < gScore[v]) {
                    gScore[v] = tentative_g;
                    cameFrom[v] = u;
                    pq.push({v, tentative_g, tentative_g + heuristic(v, goal, pos)});
                }
            }
        }

        // Reconstruct path
        std::vector<int> path;
        for (int at = goal; at != -1; at = cameFrom[at])
            path.push_back(at);
        std::reverse(path.begin(), path.end());
        return path;
    }
};
```

Key Points:

1. `Node` struct with `operator>` allows `std::priority_queue` to function as a min-heap.
2. `gScore` stores the cost from start to each vertex.
3. `heuristic()` provides problem-specific guidance.
4. `cameFrom` is used for **path reconstruction**.

14.3.3 Custom Comparators in C++

Sometimes you need more complex comparison than `operator>`:

- Include **secondary criteria**, e.g., tie-breaking with depth or other metrics.
- Use **lambda functions** or functors.

Example with Lambda Comparator:

```
auto cmp = [](const Node &a, const Node &b) { return a.f > b.f; };
std::priority_queue<Node, std::vector<Node>, decltype(cmp)> pq(cmp);
```

- This allows flexibility for **dynamic criteria**, multi-objective heuristics, or prioritizing nodes with smaller indices in case of ties.

14.3.4 Heuristic Design in C++

1. **Euclidean / Manhattan Distance**: Standard for 2D or 3D grids.
2. **Domain-Specific**: Can incorporate terrain cost, traffic, or obstacle penalties.

3. **Dynamic / Learned Heuristics**: For AI applications or games, heuristic may change at runtime.

Tip: Heuristic evaluation should be **fast**, as it is called many times. Use `inline` or precomputed tables when possible.

14.3.5 Performance and Pitfalls in C++

Aspect	Notes / Pitfalls
Priority Queue Usage	Use `std::greater` or custom comparator; remember min-heap vs max-heap defaults
Heuristic Computation	Avoid expensive calculations inside loops; precompute if possible
Floating-Point Precision	Use `double` consistently; beware of small numerical errors
Path Reconstruction	Always maintain `cameFrom` array or equivalent to recover optimal path
Memory Allocation	Reserve vectors and adjacency lists in advance for large graphs
Outdated Queue Entries	Unlike Dijkstra, may tolerate multiple entries; `gScore` check ensures correctness

14.3.6 Example: A* on a 2D Grid

```
int main() {
    Graph g(4);
    g.addEdge(0, 1, 1); g.addEdge(0, 2, 1.5);
    g.addEdge(1, 3, 1); g.addEdge(2, 3, 1);

    std::vector<std::pair<double,double>> pos = {{0,0},{1,0},{0,1},{1,1}};
    std::vector<int> path = g.aStar(0, 3, pos);

    for (int v : path) std::cout << v << " "; // Output: 0 1 3 or 0 2 3 depending on
    ↪  tie-breaking
}
```

14.3.7 Best Practices in Modern C++

1. Use **structured bindings** and range-based loops for adjacency traversal.
2. Define **Node structs** or tuples with `operator>` for clarity.
3. For dynamic or multi-criteria heuristics, prefer **lambda comparators** with `std::priority_queue`.
4. Precompute heuristic values when possible to reduce repeated calculations.
5. Always check **`gScore` before pushing to the priority queue** to avoid processing suboptimal paths.

14.4 Exercises — Multi-Source SSSP and Path Reconstruction Templates

This section provides practical exercises for implementing **multi-source single-source shortest paths (SSSP)** and designing **template-based path reconstruction utilities** in modern C++. These exercises reinforce algorithmic understanding and demonstrate reusable C++ patterns.

14.4.1 Multi-Source Single-Source Shortest Paths (SSSP)

In a **multi-source SSSP**, shortest paths are computed **from multiple starting vertices** to all other vertices in the graph. This is common in applications such as network routing, resource allocation, and grid navigation with multiple entry points.

1. **Problem Statement**

   - Given a graph `G = (V, E)` and a set of source vertices `S  V`, compute the minimum distance from any vertex in `S` to every other vertex in `V`.
   - Supports weighted or unweighted graphs; can be implemented via BFS for unweighted or Dijkstra for weighted graphs.

2. **Multi-Source BFS for Unweighted Graphs**

```
#include <vector>
#include <queue>
#include <iostream>

std::vector<int> multiSourceBFS(int V, const std::vector<std::vector<int>>&
↪  adjList, const std::vector<int>& sources) {
    const int INF = std::numeric_limits<int>::max();
```

```
    std::vector<int> dist(V, INF);
    std::queue<int> q;

    for (int s : sources) {
        dist[s] = 0;
        q.push(s);
    }

    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adjList[u]) {
            if (dist[v] == INF) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }

    return dist;
}
```

Exercise:

- Extend this BFS to **track predecessor vertices** for path reconstruction.
- Apply this BFS to a grid with multiple starting points to find the nearest source to each cell.

3. **Multi-Source Dijkstra for Weighted Graphs**

```cpp
#include <queue>
#include <tuple>

std::vector<int> multiSourceDijkstra(int V, const
↪  std::vector<std::vector<std::pair<int,int>>>& adjList, const
↪  std::vector<int>& sources) {
    const int INF = std::numeric_limits<int>::max();
    std::vector<int> dist(V, INF);
    using pii = std::pair<int,int>;
    std::priority_queue<pii, std::vector<pii>, std::greater<pii>> pq;

    for (int s : sources) {
        dist[s] = 0;
        pq.push({0, s});
    }

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) continue;

        for (auto &[v, w] : adjList[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

Exercise:

- Modify this implementation to **store `cameFrom` arrays** for each source.
- Use **templates** to generalize the algorithm for different numeric types (`int`, `double`, `float`).

14.4.2 Path Reconstruction Templates

In many shortest-path problems, it is not enough to know the distance; reconstructing the **actual path** is crucial.

1. **Generic Path Reconstruction Template**

```cpp
#include <vector>
#include <algorithm>

template <typename Vertex>
std::vector<Vertex> reconstructPath(Vertex target, const std::vector<Vertex>&
↪  cameFrom) {
    std::vector<Vertex> path;
    for (Vertex at = target; at != Vertex(-1); at = cameFrom[at])
        path.push_back(at);
    std::reverse(path.begin(), path.end());
    return path;
}
```

Key Features:

(a) **Generic**: Works with any vertex type (`int`, `size_t`, or user-defined type with integer indexing).

(b) **Reversible**: Uses `std::reverse` for natural ordering from source to target.

(c) **Safe Default**: Requires `cameFrom[source] = -1`.

2. **Exercise: Multi-Source Path Reconstruction**

    - Implement a **multi-source `cameFrom` map**: store a predecessor for each vertex **corresponding to the closest source**.
    - Reconstruct shortest paths from **any vertex to its nearest source**:

```
template <typename Vertex>
std::vector<Vertex> reconstructMultiSourcePath(Vertex target, const
↪  std::vector<Vertex>& cameFrom) {
    std::vector<Vertex> path;
    for (Vertex at = target; cameFrom[at] != at; at = cameFrom[at])
        path.push_back(at);
    path.push_back(cameFrom[target]); // Add the source
    std::reverse(path.begin(), path.end());
    return path;
}
```

    - Exercise: Compare BFS vs Dijkstra versions for correctness and performance.

14.4.3 Additional Exercises

1. **Weighted Grid Multi-Source SSSP**: Implement Dijkstra with multiple sources on a 2D grid with terrain costs.
2. **SPFA Multi-Source Exercise**: Extend SPFA to accept multiple starting vertices and track predecessor arrays.
3. **A* with Multiple Goals**: Adapt the A* heuristic to terminate early upon reaching any of the multiple goal vertices.

4. **Template-Based Edge Types**: Generalize path reconstruction and multi-source SSSP for graphs with **custom edge structs**, supporting additional metadata like weights, labels, or capacity.

14.4.4 Best Practices in Modern C++

1. Use **template functions** for reusable path reconstruction across graph types and vertex types.
2. Preallocate `dist`, `cameFrom`, and adjacency structures to avoid dynamic allocation overhead.
3. Prefer **structured bindings** and **range-based loops** for clarity.
4. Always **initialize `cameFrom` arrays** for multi-source problems to avoid invalid accesses.
5. Use **`std::vector` indices** for predecessor arrays, or `std::unordered_map` for non-contiguous/custom vertex types.

Chapter 15

Minimum Spanning Trees & Union-Find

15.1 Kruskal's Algorithm with Efficient DSU

Kruskal's algorithm is one of the most widely used algorithms to compute a **Minimum Spanning Tree (MST)** of a connected, weighted graph. Its efficiency depends heavily on the **Disjoint Set Union (DSU)** or Union-Find data structure, especially when enhanced with **path compression** and **union by rank**.

15.1.1 Algorithm Overview

Kruskal's MST Algorithm:

1. Sort all edges by **non-decreasing weight**.
2. Initialize a **DSU** structure where each vertex is its own set.
3. Iterate over sorted edges:

- If the edge connects two **different sets**, include it in the MST and **union** the sets.
- Otherwise, skip the edge to avoid cycles.

4. Stop when MST contains exactly **V - 1 edges**.

Time Complexity:

- Sorting edges: O(E log E)
- DSU operations: O((V)), where is the inverse Ackermann function.
- Total: O(E log E + E (V)) O(E log E) for practical purposes.

15.1.2 Efficient DSU Implementation

1. **Path Compression**
   - During `find(x)`, recursively update each visited node's parent to the root.
   - Reduces the depth of the tree for future operations.
2. **Union by Rank**
   - Always attach the **smaller tree** under the root of the **larger tree** (rank indicates tree depth).
   - Prevents trees from becoming too deep.
3. **C++ Template DSU**

```cpp
#include <vector>
#include <algorithm>
#include <tuple>
#include <iostream>

template <typename T>
class DSU {
    std::vector<T> parent;
    std::vector<T> rank;
public:
    DSU(T n) : parent(n), rank(n, 0) {
        for (T i = 0; i < n; ++i)
            parent[i] = i;
    }

    T find(T x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]); // path compression
        return parent[x];
    }

    bool unite(T x, T y) {
        T px = find(x);
        T py = find(y);
        if (px == py) return false;

        if (rank[px] < rank[py]) {
            parent[px] = py;
        } else if (rank[px] > rank[py]) {
            parent[py] = px;
        } else {
            parent[py] = px;
```

```
            rank[px]++;
        }
        return true;
    }
};
```

Key Features:

- Fully generic over vertex type `T` (e.g., `int`, `size_t`).
- Combines **path compression** and **union by rank** for near-constant amortized time per operation.
- `unite` returns `false` if the two vertices were already connected.

15.1.3 Kruskal's Algorithm Using DSU

```
template <typename T>
struct Edge {
    T u, v;
    int weight;
    bool operator<(const Edge& other) const { return weight < other.weight; }
};

template <typename T>
std::vector<Edge<T>> kruskal(int V, std::vector<Edge<T>>& edges) {
    std::sort(edges.begin(), edges.end());
    DSU<T> dsu(V);
    std::vector<Edge<T>> mst;

    for (auto& e : edges) {
```

```
        if (dsu.unite(e.u, e.v)) {
            mst.push_back(e);
        }
        if (mst.size() == V - 1) break;
    }

    return mst;
}
```

Notes:

1. Sorting edges ensures **minimum weight edges are considered first**.
2. DSU prevents cycles by only adding edges connecting **different sets**.
3. Returns a vector of edges representing the MST.

15.1.4 Example Usage

```
int main() {
    int V = 5;
    std::vector<Edge<int>> edges = {
        {0,1,10}, {0,2,6}, {0,3,5}, {1,3,15}, {2,3,4}
    };

    auto mst = kruskal(V, edges);

    std::cout << "Edges in MST:\n";
    for (auto &e : mst)
        std::cout << e.u << " - " << e.v << " : " << e.weight << "\n";
}
```

Output:

```
Edges in MST:
2 - 3 : 4
0 - 3 : 5
0 - 1 : 10
```

15.1.5 Best Practices in Modern C++

1. **Templates**:
    - Use templates for DSU and Edge to support multiple vertex types and numeric types.
2. **Structured Bindings**:
    - Can use `[u,v,w]` with `std::tie` or C++20 structured bindings when iterating over edges.
3. **Preallocate Vectors**:
    - Avoid dynamic resizing for `parent` and `rank` arrays in DSU for large graphs.
4. **Early Termination**:
    - Stop iterating edges once MST has `V-1` edges to save unnecessary iterations.
5. **Use `std::sort`**:
    - Prefer stable, standard library sort for portability and efficiency.

15.1.6 Exercises

1. Extend DSU to support **rollback operations** for offline dynamic connectivity.
2. Modify Kruskal's algorithm to handle **multi-graphs** with parallel edges efficiently.
3. Implement **Kruskal with custom comparators** to compute **maximum spanning tree** instead of minimum.
4. Compare **union by size** vs **union by rank** in practice for large-scale graphs.

15.2 Prim's Algorithm — Binary Heap vs Fibonacci Heap

Prim's algorithm is a classic **greedy algorithm** for computing the **Minimum Spanning Tree (MST)** of a connected, weighted graph. Its efficiency depends heavily on the **priority queue structure** used to select the next minimum-weight edge.

15.2.1 Prim's Algorithm Overview

Steps:

1. Start with an arbitrary vertex and mark it as part of the MST.
2. Use a **priority queue** to maintain candidate edges connecting the MST to non-MST vertices.
3. Repeatedly select the **minimum-weight edge** that expands the MST.
4. Continue until all vertices are included in the MST.

Time Complexity:

- Using an **array**: $O(V^2)$ — simple but inefficient for sparse graphs.
- Using a **binary heap**: $O(E \log V)$ — efficient for most practical graphs.
- Using a **Fibonacci heap**: $O(E + V \log V)$ — asymptotically optimal for very large sparse graphs.

15.2.2 Prim Using Binary Heap (Standard Approach)

Binary heaps are implemented via **`std::priority_queue`** in C++. While `std::priority_queue` is a max-heap by default, it can be converted to a min-heap using `std::greater` and `std::pair` for distances.

C++ Implementation

```cpp
#include <vector>
#include <queue>
#include <limits>
#include <iostream>

struct Edge {
    int to;
    int weight;
};

std::vector<int> primBinaryHeap(int V, const std::vector<std::vector<Edge>>& adjList)
↪ {
    const int INF = std::numeric_limits<int>::max();
    std::vector<int> key(V, INF);       // Minimum edge weight to MST
    std::vector<int> parent(V, -1);     // Parent in MST
    std::vector<bool> inMST(V, false);

    key[0] = 0;
    using pii = std::pair<int,int>; // {key, vertex}
    std::priority_queue<pii, std::vector<pii>, std::greater<pii>> pq;
    pq.push({0, 0});

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
```

```
        if (inMST[u]) continue;
        inMST[u] = true;

        for (auto &e : adjList[u]) {
            int v = e.to;
            int w = e.weight;
            if (!inMST[v] && w < key[v]) {
                key[v] = w;
                parent[v] = u;
                pq.push({key[v], v}); // lazy update
            }
        }
    }

    return parent; // contains MST edges
}
```

Notes:

- Lazy updates may push multiple entries per vertex; correctness is maintained by `inMST` check.
- `O(E log V)` complexity makes this suitable for graphs with up to hundreds of thousands of edges.

15.2.3 Fibonacci Heap: Theoretical Advantage

Fibonacci heaps improve the **decrease-key operation** to **O(1) amortized**, making Prim's algorithm asymptotically faster:

- **Binary Heap**: `decrease-key` is O(log V), total O(E log V)
- **Fibonacci Heap**: `decrease-key` is O(1), total O(E + V log V)

When to use:

- Very **large, sparse graphs** (e.g., E O(V))
- Performance-critical applications like network optimization or computational geometry.

Practical Considerations:

- Complex to implement in C++; rarely used in production.
- Binary heap is simpler, cache-friendly, and often faster for small to medium-sized graphs.
- Libraries like Boost provide Fibonacci heap implementations if needed.

15.2.4 Comparison: Binary Heap vs Fibonacci Heap

Feature	Binary Heap	Fibonacci Heap
`insert`	O(log V)	O(1)
`extract-min`	O(log V)	O(log V)
`decrease-key`	O(log V)	O(1)
Space Complexity	O(V)	O(V)
Implementation Complexity	Simple	Complex
Practical Performance	Fast for small/medium graphs	Only better for very large sparse graphs

Recommendation: Use **binary heap** in C++ for most real-world MST problems; reserve Fibonacci heap for theoretical studies or extreme-scale graphs.

15.2.5 C++ Implementation Notes and Best Practices

1. **Structured Bindings**: Use `[v,w]` or `{to,weight}` for adjacency traversal clarity.
2. **Lazy Updates**: Avoid explicit decrease-key if using `std::priority_queue`; simply push new values and check `inMST`.
3. **Memory Preallocation**: Reserve adjacency vectors for large graphs to reduce allocation overhead.
4. **Parent Array**: Always maintain a `parent` array for easy MST edge reconstruction.
5. **Edge Cases**: Ensure the graph is connected; otherwise, MST is undefined.

15.2.6 Exercises

1. Implement **Prim's algorithm using an indexed binary heap** with explicit decrease-key.
2. Compare runtime for dense vs sparse graphs using **array**, **binary heap**, and **Fibonacci heap** implementations.
3. Extend Prim to handle **multi-graph** scenarios and **negative weights**.
4. Implement MST edge reconstruction and compute **total weight** of the MST.

15.3 Exercises — MST Variants and Dynamic Connectivity

This section provides exercises designed to deepen understanding of **minimum spanning trees (MST)** and the use of **Union-Find (DSU)** for **dynamic connectivity** problems. These exercises challenge readers to explore MST variants and advanced applications of union-find in modern C++.

15.3.1 MST Variants

MST problems have several interesting **variants**, each with subtle algorithmic differences:

1. **Maximum Spanning Tree**

   - Instead of minimizing the sum of edge weights, find a spanning tree that **maximizes the total weight**.
   - Modification of Kruskal's algorithm: sort edges in **descending order** and apply DSU as usual.

   Exercise:

   - Implement a **template-based Kruskal** to handle both minimum and maximum spanning trees.
   - Compare runtime with the standard MST algorithm on graphs with the same topology but different weight distributions.

2. **MST with Constraints**

- **Edge constraints**: e.g., include or exclude certain edges.
- **Vertex constraints**: e.g., only connect a subset of vertices or respect partitioning.
- Can be handled via **preprocessing edges** or **modifying union rules**.

Exercise:

- Implement **Kruskal variant** that **forces specific edges** to be included in the MST while maintaining correctness.
- Handle cases where constraints create **disconnected components** and output an error if MST is impossible.

3. **Multi-Graph and Parallel Edges**

- MST algorithms need to account for **parallel edges** between vertices.
- Always select **the edge with minimal weight** in Kruskal or Prim iterations.

Exercise:

- Implement Kruskal on a **multi-graph**, handling duplicate edges efficiently.
- Compare MST weight with that of the simple graph to verify correctness.

15.3.2 Dynamic Connectivity

Dynamic connectivity deals with graphs where edges are **added or removed over time**, and queries ask whether two vertices are connected.

1. **Union-Find for Dynamic Connectivity**

- DSU is naturally suited for **incremental connectivity** (edge additions).
- Each union operation merges two components efficiently.
- Path compression and union by rank ensure **near-constant time operations**.

Exercise:

- Implement a **dynamic connectivity tracker** using DSU with path compression.
- Support operations:
  - `union(u,v)`: add an edge between `u` and `v`
  - `connected(u,v)`: check if `u` and `v` belong to the same component

```
template <typename T>
class DynamicConnectivity {
    DSU<T> dsu;
public:
    DynamicConnectivity(T n) : dsu(n) {}
    void addEdge(T u, T v) { dsu.unite(u, v); }
    bool connected(T u, T v) { return dsu.find(u) == dsu.find(v); }
};
```

2. **Offline Connectivity Queries**

- When **edge removals** are allowed, simple DSU is insufficient.
- Use **offline techniques**:
  - **Reverse Delete**: Remove edges in reverse and answer queries.

  - **Euler Tour Trees** or **Link-Cut Trees**: advanced structures for fully dynamic connectivity.

Exercise:

- Implement a basic **offline connectivity query** algorithm:
  - Given a sequence of edge additions/removals and connectivity queries, determine which queries are satisfied at each step.
- Optional: Explore **Link-Cut Trees** for online dynamic connectivity.

3. **MST Under Dynamic Updates**

- **Dynamic MST problem**: maintain MST while edges are **added or removed**.
- Techniques:
  - **Recompute MST** from scratch (simple but inefficient)
  - **Use DSU with edge priority structures** (incremental updates)
  - **Advanced structures**: Euler Tour Trees, Dynamic Trees

Exercise:

- Implement a **DSU-based incremental MST**: edges are added one by one, and MST weight is updated efficiently.
- Optional: measure performance for sparse vs dense graphs.

15.3.3 Best Practices and C++ Tips

1. **Use templates** for DSU and Edge structures to generalize algorithms for multiple numeric types and vertex types.

2. **Preallocate vectors** (`parent`, `rank`, `adjacency lists`) to avoid dynamic memory overhead.
3. **Structured bindings**: use `[u,v,w]` or `{to,weight}` for edge traversal clarity.
4. For dynamic graphs:
   - Maintain **explicit parent arrays**.
   - Always check **component sizes** if implementing union by size.
5. Consider **lazy updates** for binary heaps or MST updates in dynamic scenarios.

15.3.4 Additional Exercises for Mastery

1. Implement **maximum spanning tree** using Kruskal and compare with the standard MST.
2. Implement **MST with forced edges**, verifying correctness when constraints are impossible.
3. Develop a **dynamic connectivity tracker** supporting millions of `union` and `connected` queries.
4. Extend MST updates to handle **edge deletions**, exploring offline techniques for correctness.
5. Compare performance of **binary heap vs Fibonacci heap** in incremental MST updates.

Chapter 16

Network Flow & Advanced Graphs

16.1 Ford-Fulkerson, Edmonds-Karp, Dinic — C++ Implementations and Performance Tradeoffs

Network flow algorithms are a cornerstone of graph theory with applications in **transportation, communication networks, bipartite matching, and scheduling**. This section covers **maximum flow algorithms** in modern C++ with emphasis on **performance, trade-offs, and practical implementation**.

16.1.1 Ford-Fulkerson Method

1. **Algorithm Overview**

   Ford-Fulkerson computes the **maximum flow** in a flow network using the **augmenting path approach**:

   (a) Initialize flow to 0.

   (b) While a path exists from source `s` to sink `t` in the **residual graph**, augment flow along the path.

   (c) Update the residual capacities and reverse edges.

   (d) Repeat until no augmenting path exists.

2. **Complexity**

   - Depends on **maximum flow value `F`** and number of edges `E`.
   - **Time Complexity**: O(E × F) with naive DFS for augmenting paths.
   - Not guaranteed to terminate with irrational capacities.

3. **C++ Implementation (DFS-based)**

```cpp
#include <vector>
#include <algorithm>

struct Edge {
    int to;
    int rev;
    int capacity;
};

class FlowNetwork {
    int V;
    std::vector<std::vector<Edge>> adj;
public:
    FlowNetwork(int V) : V(V), adj(V) {}

    void addEdge(int u, int v, int cap) {
        adj[u].push_back({v, (int)adj[v].size(), cap});
        adj[v].push_back({u, (int)adj[u].size()-1, 0}); // reverse edge
    }

    int dfs(int u, int t, int f, std::vector<bool>& visited) {
        if (u == t) return f;
        visited[u] = true;
        for (auto &e : adj[u]) {
            if (!visited[e.to] && e.capacity > 0) {
                int d = dfs(e.to, t, std::min(f, e.capacity), visited);
                if (d > 0) {
                    e.capacity -= d;
                    adj[e.to][e.rev].capacity += d;
                    return d;
                }
            }
```

```
        }
        return 0;
    }

    int maxFlow(int s, int t) {
        int flow = 0, f;
        std::vector<bool> visited(V);
        while (true) {
            std::fill(visited.begin(), visited.end(), false);
            f = dfs(s, t, INT32_MAX, visited);
            if (f == 0) break;
            flow += f;
        }
        return flow;
    }
};
```

Notes:

- Simple, intuitive implementation.
- Can be inefficient for large capacities or dense networks.
- DFS can lead to non-optimal path selection, increasing iterations.

16.1.2 Edmonds-Karp Algorithm

1. **Algorithm Overview**
    - A specialization of Ford-Fulkerson.
    - Uses **BFS** to find the shortest augmenting path in terms of **number of edges**.

- Ensures **polynomial time**: $O(V \times E^2)$.

2. **C++ BFS-based Implementation**

```cpp
#include <queue>

int bfs(int s, int t, std::vector<int>& parent, const
↪  std::vector<std::vector<Edge>>& adj) {
    std::fill(parent.begin(), parent.end(), -1);
    parent[s] = s;
    std::queue<std::pair<int,int>> q;
    q.push({s, INT32_MAX});

    while (!q.empty()) {
        auto [u, flow] = q.front(); q.pop();
        for (auto &e : adj[u]) {
            if (parent[e.to] == -1 && e.capacity > 0) {
                parent[e.to] = u;
                int new_flow = std::min(flow, e.capacity);
                if (e.to == t) return new_flow;
                q.push({e.to, new_flow});
            }
        }
    }
    return 0;
}

int edmondsKarp(int s, int t, std::vector<std::vector<Edge>>& adj) {
    int flow = 0;
    std::vector<int> parent(adj.size());
    int new_flow;

    while ((new_flow = bfs(s, t, parent, adj)) > 0) {
```

```
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            for (auto &e : adj[prev]) {
                if (e.to == cur) {
                    e.capacity -= new_flow;
                    adj[cur][e.rev].capacity += new_flow;
                    break;
                }
            }
            cur = prev;
        }
    }
    return flow;
}
```

Advantages:

- Guarantees O(V × E²) complexity.
- Avoids path selection issues from naive DFS.

Disadvantages:

- Still slower on **very large sparse networks**.
- BFS adds memory overhead for `queue` and `parent` arrays.

16.1.3 Dinic's Algorithm

1. **Algorithm Overview**

   Dinic's algorithm improves performance using:

(a) **Level Graph**: Constructed using BFS to represent shortest path layers from source.

(b) **Blocking Flow**: Use DFS to push maximum flow along paths in the level graph.

(c) Repeat until no more blocking flows exist.

2. **Complexity**

- $O(E \times V^2)$ for general graphs.
- $O(E \times \sqrt{V})$ for **unit capacity graphs**.
- Much faster in practice for **dense networks**.

3. **C++ Implementation**

```
class Dinic {
    struct Edge {
        int to, rev;
        int cap;
    };
    int V;
    std::vector<std::vector<Edge>> adj;
    std::vector<int> level, ptr;

    bool bfs(int s, int t) {
        level.assign(V, -1);
        level[s] = 0;
        std::queue<int> q;
        q.push(s);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (auto &e : adj[u]) {
```

```
                if (e.cap > 0 && level[e.to] == -1) {
                    level[e.to] = level[u] + 1;
                    q.push(e.to);
                }
            }
        }
        return level[t] != -1;
    }

    int dfs(int u, int t, int pushed) {
        if (u == t || pushed == 0) return pushed;
        for (int &i = ptr[u]; i < adj[u].size(); ++i) {
            Edge &e = adj[u][i];
            if (level[e.to] == level[u] + 1 && e.cap > 0) {
                int tr = dfs(e.to, t, std::min(pushed, e.cap));
                if (tr > 0) {
                    e.cap -= tr;
                    adj[e.to][e.rev].cap += tr;
                    return tr;
                }
            }
        }
        return 0;
    }

public:
    Dinic(int V) : V(V), adj(V), level(V), ptr(V) {}

    void addEdge(int u, int v, int cap) {
        adj[u].push_back({v, (int)adj[v].size(), cap});
        adj[v].push_back({u, (int)adj[u].size()-1, 0});
    }
```

```
    int maxFlow(int s, int t) {
        int flow = 0, pushed;
        while (bfs(s, t)) {
            ptr.assign(V, 0);
            while ((pushed = dfs(s, t, INT32_MAX)) > 0)
                flow += pushed;
        }
        return flow;
    }
};
```

Advantages:

- Much faster than Edmonds-Karp for dense or large-capacity graphs.
- Layered BFS ensures **efficient blocking flows**.
- Works well with **unit capacity graphs** or **highly connected networks**.

Disadvantages:

- Slightly more complex to implement.
- Requires additional memory for `level` and `ptr` arrays.

16.1.4 Performance Trade-offs

Algorithm	Time Complexity	Memory Usage	Practical Notes
Ford-Fulkerson DFS	$O(E \times F)$	Low	Simple; inefficient with large capacities
Edmonds-Karp BFS	$O(V \times E^2)$	Medium	Guaranteed polynomial; simple BFS implementation
Dinic	$O(E \times V^2)$ general / $O(E \sqrt{V})$ unit	Higher	Fastest for dense graphs; slightly complex

Guidelines:

1. Use **Ford-Fulkerson** for educational purposes or small integer flows.
2. Use **Edmonds-Karp** for guaranteed correctness on small to medium graphs.
3. Use **Dinic** for large, dense, or high-capacity graphs in practice.

16.1.5 Exercises

1. Implement **Ford-Fulkerson** and test with both **DFS** and **BFS** augmenting paths; compare performance.
2. Implement **Dinic's algorithm** and analyze runtime for unit-capacity vs general graphs.
3. Modify algorithms to handle **undirected graphs** and **multi-edges** correctly.
4. Apply Dinic to **maximum bipartite matching** problem using flow transformation.

5. Benchmark all three algorithms on a **grid network** and **dense random graph**, measuring runtime and memory usage.

16.2 Matching Algorithms and Min-Cost Max-Flow

This section delves into **advanced network flow techniques**, covering **bipartite matching** using the Hopcroft–Karp algorithm and **flows with capacities and costs**, including the **Min-Cost Max-Flow problem**. We emphasize **modern C++ implementations**, performance trade-offs, and practical considerations.

16.2.1 Bipartite Matching — Hopcroft–Karp Algorithm

1. **Problem Overview**

   Given a bipartite graph $G = (U \cup V, E)$:

   - Find the **maximum matching**: the largest set of edges with no two edges sharing a vertex.
   - Hopcroft–Karp improves over naive DFS-based matching by finding **multiple augmenting paths simultaneously**.

2. **Algorithm Steps**

   (a) Initialize **matching M =** .

   (b) While an **augmenting path** exists:

   - Construct a **layered BFS** from unmatched vertices in U.
   - Use **DFS** on layers to find **vertex-disjoint augmenting paths**.
   - Augment all paths simultaneously.

   (c) Repeat until no augmenting path exists.

   Time Complexity:

- $O(\sqrt{V} \times E)$, significantly faster than $O(V \times E)$ for naive approaches.

3. **C++ Implementation**

```cpp
#include <vector>
#include <queue>
#include <algorithm>

class BipartiteMatcher {
    int n, m; // sizes of left and right sets
    std::vector<std::vector<int>> adj;
    std::vector<int> pairU, pairV, dist;

public:
    BipartiteMatcher(int n, int m) : n(n), m(m), adj(n), pairU(n, -1), pairV(m,
    ↪  -1), dist(n) {}

    void addEdge(int u, int v) { adj[u].push_back(v); }

    bool bfs() {
        std::queue<int> q;
        for (int u = 0; u < n; u++) {
            if (pairU[u] == -1) {
                dist[u] = 0;
                q.push(u);
            } else {
                dist[u] = INT32_MAX;
            }
        }
        bool found = false;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int v : adj[u]) {
```

```
            if (pairV[v] == -1) {
                found = true;
            } else if (dist[pairV[v]] == INT32_MAX) {
                dist[pairV[v]] = dist[u] + 1;
                q.push(pairV[v]);
            }
        }
    }
    return found;
}

bool dfs(int u) {
    for (int v : adj[u]) {
        if (pairV[v] == -1 || (dist[pairV[v]] == dist[u] + 1 &&
        ↪  dfs(pairV[v]))) {
            pairU[u] = v;
            pairV[v] = u;
            return true;
        }
    }
    dist[u] = INT32_MAX;
    return false;
}

int maxMatching() {
    int matching = 0;
    while (bfs()) {
        for (int u = 0; u < n; u++)
            if (pairU[u] == -1 && dfs(u))
                matching++;
    }
    return matching;
```

```
    }
};{        while (!q.empty()) \{}
\NormalTok{            int u = q.front(); q.pop();}
\NormalTok{            for (int v : adj[u]) \{}
\NormalTok#include <vector>
#include <queue>
#include <algorithm>

class BipartiteMatcher {
    int n, m; // sizes of left and right sets
    std::vector<std::vector<int>> adj;
    std::vector<int> pairU, pairV, dist;

public:
    BipartiteMatcher(int n, int m) : n(n), m(m), adj(n), pairU(n, -1), pairV(m,
    ↪  -1), dist(n) {}

    void addEdge(int u, int v) { adj[u].push_back(v); }

    bool bfs() {
        std::queue<int> q;
        for (int u = 0; u < n; u++) {
            if (pairU[u] == -1) {
                dist[u] = 0;
                q.push(u);
            } else {
                dist[u] = INT32_MAX;
            }
        }
        bool found = false;
        while (!q.empty()) {
            int u = q.front(); q.pop();
```

```
            for (int v : adj[u]) {
                if (pairV[v] == -1) {
                    found = true;
                } else if (dist[pairV[v]] == INT32_MAX) {
                    dist[pairV[v]] = dist[u] + 1;
                    q.push(pairV[v]);
                }
            }
        }
        return found;
    }

    bool dfs(int u) {
        for (int v : adj[u]) {
            if (pairV[v] == -1 || (dist[pairV[v]] == dist[u] + 1 &&
            ↪  dfs(pairV[v]))) {
                pairU[u] = v;
                pairV[v] = u;
                return true;
            }
        }
        dist[u] = INT32_MAX;
        return false;
    }

    int maxMatching() {
        int matching = 0;
        while (bfs()) {
            for (int u = 0; u < n; u++)
                if (pairU[u] == -1 && dfs(u))
                    matching++;
        }
```

```
        return matching;
    }
};
```

Notes:

- Efficient for large bipartite graphs (e.g., assignment problems).
- BFS layers reduce redundant searches and improve amortized performance.

16.2.2 Flows with Capacities and Costs — Min-Cost Max-Flow (MCMF)

1. **Problem Overview**
   - Extend the **maximum flow problem** to minimize the **total cost** of sending flow.
   - Each edge (u,v) has:
     - **Capacity** $c(u,v)$
     - **Cost per unit flow** $w(u,v)$
   - Goal: Maximize flow from source `s` to sink `t` while minimizing total cost.
2. **Key Approaches**
   (a) **Successive Shortest Path Algorithm**:
   - Repeatedly find **shortest paths** (using edge costs) in residual graph.
   - Push flow along these paths until no more augmenting paths exist.

   (b) **Cycle-Canceling Algorithm**:

- Start with any feasible flow.
- Repeatedly find **negative-cost cycles** in residual graph and cancel them.

(c) **Practical Implementation**:

- Use **Bellman-Ford** or **Dijkstra with potentials** to find shortest paths efficiently.

3. **C++ Implementation Sketch (Successive Shortest Path)**

```cpp
#include <vector>
#include <queue>
#include <limits>
#include <tuple>

struct Edge {
    int to, rev;
    int capacity;
    int cost;
};

class MinCostMaxFlow {
    int V;
    std::vector<std::vector<Edge>> adj;
public:
    MinCostMaxFlow(int V) : V(V), adj(V) {}

    void addEdge(int u, int v, int cap, int cost) {
        adj[u].push_back({v, (int)adj[v].size(), cap, cost});
        adj[v].push_back({u, (int)adj[u].size()-1, 0, -cost});
    }
```

```
std::pair<int,int> minCostMaxFlow(int s, int t) {
    int flow = 0, cost = 0;
    const int INF = std::numeric_limits<int>::max();
    std::vector<int> dist(V), parent(V), parentEdge(V);
    while (true) {
        std::fill(dist.begin(), dist.end(), INF);
        dist[s] = 0;
        bool updated = true;
        // Bellman-Ford
        for (int i = 0; i < V && updated; i++) {
            updated = false;
            for (int u = 0; u < V; u++) {
                if (dist[u] == INF) continue;
                for (int k = 0; k < adj[u].size(); k++) {
                    Edge &e = adj[u][k];
                    if (e.capacity > 0 && dist[u] + e.cost < dist[e.to]) {
                        dist[e.to] = dist[u] + e.cost;
                        parent[e.to] = u;
                        parentEdge[e.to] = k;
                        updated = true;
                    }
                }
            }
        }
        if (dist[t] == INF) break;

        // Find bottleneck
        int pushFlow = INF;
        for (int v = t; v != s; v = parent[v])
            pushFlow = std::min(pushFlow,
            ↪  adj[parent[v]][parentEdge[v]].capacity);
```

```
            // Apply flow
            for (int v = t; v != s; v = parent[v]) {
                Edge &e = adj[parent[v]][parentEdge[v]];
                e.capacity -= pushFlow;
                adj[v][e.rev].capacity += pushFlow;
            }

            flow += pushFlow;
            cost += pushFlow * dist[t];
        }
        return {flow, cost};
    }
};
```

Notes:

- Works for moderate graph sizes.
- Can be optimized with **Dijkstra + potentials** for graphs with non-negative costs.
- Useful in **assignment problems, network routing, and logistics**.

16.2.3 Performance Trade-offs

Algorithm	**Time Complexity**	**Memory Usage**	**Practical Notes**
Hopcroft–Karp (Bipartite)	$O(\sqrt{V} \times E)$	Medium	Efficient for large bipartite graphs

Min-Cost Max-Flow (Bellman–Ford)	$O(F \times V \times E)$	Medium	Simple but slow for large flows
Min-Cost Max-Flow (Dijkstra with potentials)	$O(F \times E \log V)$	Higher	Fast for sparse graphs with non-negative costs

Guidelines:

1. Use **Hopcroft–Karp** for maximum bipartite matching; avoids slow DFS iterations.
2. Use **successive shortest path** for MCMF on small/medium graphs.
3. Use **Dijkstra + potentials** or **cycle-canceling optimizations** for large graphs with non-negative edge costs.

16.2.4 Exercises

1. Implement Hopcroft–Karp and test on large random bipartite graphs.
2. Implement MCMF for **assignment problem**: minimize cost of assigning workers to jobs.
3. Modify MCMF to handle **edges with zero capacity or negative costs**; analyze correctness.
4. Benchmark **Bellman-Ford vs Dijkstra potentials** for MCMF on sparse vs dense graphs.
5. Extend Hopcroft–Karp to support **weighted bipartite matching** using MCMF transformation.

16.3 Exercises — Bipartite Matching and Project Allocation Simulation

This section provides hands-on exercises to deepen understanding of **bipartite matching**, **project allocation problems**, and the practical application of **network flow algorithms** in C++. These exercises are designed to reinforce concepts introduced in previous sections and challenge readers to implement solutions for real-world scenarios.

16.3.1 Bipartite Matching Exercises

1. **Maximum Matching in Bipartite Graphs**

   Objective: Implement the **Hopcroft–Karp algorithm** to find the maximum matching in a bipartite graph.

   Exercise Tasks:

   (a) Generate a bipartite graph with sets U and V, and a random set of edges.

   (b) Implement **Hopcroft–Karp** to compute the maximum matching.

   (c) Verify correctness by ensuring:

   - No two edges share a vertex.
   - The total number of matched edges equals the algorithm output.

   (d) Benchmark performance for:

   - Sparse graphs (|E| |V|)
   - Dense graphs (|E| |U| × |V|)

Advanced Task: Extend the algorithm to **weighted bipartite graphs**, transforming the problem into **Min-Cost Max-Flow**.

2. **Edge Case Handling**

   - Test cases with:
     - Empty sets (no vertices or edges)
     - Fully connected bipartite graphs
     - Graphs with isolated vertices
   - Verify algorithm handles **all scenarios without runtime errors**.

   C++ Tips:

   - Use `std::vector<int>` for adjacency lists.
   - Use **structured bindings** `[u,v]` for clarity in loops.
   - Preallocate vectors for large graphs to reduce memory allocation overhead.

16.3.2 Project Allocation Simulation

1. **Problem Overview**

   The **project allocation problem** is a common application of **bipartite matching and network flows**:

   - **Objective**: Assign `n` students to `m` projects based on preferences.
   - Each student can have multiple preferred projects.
   - Each project may have a **capacity constraint** (number of students it can accept).
   - Goal: Maximize satisfaction (total number of assigned students).

This is a **classic application of bipartite matching and Min-Cost Max-Flow**.

2. **Modeling the Problem**

   (a) Represent **students** as set U and **projects** as set V.

   (b) Each preference forms an **edge** between a student and a project.

   (c) Projects have **capacities**, represented by edge capacities in a **flow network**.

   (d) Optionally, assign **costs** to preferences to maximize satisfaction using **MCMF**.

3. **C++ Implementation Sketch**

```cpp
struct Edge { int to, rev, cap, cost; };

class ProjectAllocation {
    int V;
    std::vector<std::vector<Edge>> adj;
public:
    ProjectAllocation(int V) : V(V), adj(V) {}

    void addEdge(int u, int v, int cap, int cost = 0) {
        adj[u].push_back({v, (int)adj[v].size(), cap, cost});
        adj[v].push_back({u, (int)adj[u].size()-1, 0, -cost});
    }

    // Min-Cost Max-Flow implementation
    std::pair<int,int> allocate(int s, int t) {
        // Successive Shortest Path or Dinic-based MCMF
        // Returns {totalFlow, totalCost}
    }
};
```

Exercise Tasks:

(a) Implement **flow network** for student-project assignment.

(b) Use **Min-Cost Max-Flow** to maximize the number of satisfied student preferences.

(c) Output:

- Assigned students per project
- Total number of assignments
- Total satisfaction score (optional if using costs)

(d) Test with:

- Uniform capacities (all projects accept same number of students)
- Varied capacities
- Random preference distributions

4. **Performance Considerations**

- **Sparse preference networks** → simple Edmonds-Karp or DFS-based MCMF suffices.
- **Dense preference networks** → Dinic or Dijkstra with potentials improves performance.
- Preallocate **adjacency lists** and **residual capacities** for large-scale simulations.
- Avoid dynamic memory allocations inside loops; reuse vectors where possible.

16.3.3 Advanced Extensions

1. **Weighted Preferences**: Assign numerical scores to student preferences and maximize total satisfaction.
2. **Multiple Assignment Constraints**: Students can be assigned to more than one project; extend flow network with multiple edges per student.
3. **Dynamic Updates**: Add or remove students/projects and update allocation dynamically using **incremental flow updates**.

16.3.4 Suggested Exercise Sequence

1. Implement **basic bipartite matching** on small graphs.
2. Extend to **weighted bipartite matching** using MCMF.
3. Simulate **project allocation** with fixed capacities.
4. Benchmark allocation algorithms on **random large-scale student-project networks**.
5. Explore **satisfaction optimization** with variable edge costs.

Part V

Design Paradigms & Algorithmic Techniques

Chapter 17

Divide and Conquer

17.1 Merge Sort, Quicksort, and Recursion Patterns in C++

Divide and Conquer is a foundational design paradigm where a large problem is broken into smaller subproblems, solved recursively, and then combined into the final result. Among the most illustrative algorithms in this category are **Merge Sort** and **Quicksort**. Their implementations demonstrate key recursion patterns in C++, including **tail recursion** and **iterative transformations** for efficiency.

17.1.1 Merge Sort

1. **Algorithm Overview**

   - **Divide**: Split the array into two halves.
   - **Conquer**: Recursively sort both halves.
   - **Combine**: Merge two sorted halves into a single sorted array.

Time Complexity:

- Best, Average, Worst: O(n log n).

Space Complexity:

- O(n) for auxiliary array.

2. **C++ Implementation**

```cpp
#include <vector>
#include <algorithm>

void merge(std::vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    std::vector<int> L(arr.begin() + left, arr.begin() + mid + 1);
    std::vector<int> R(arr.begin() + mid + 1, arr.begin() + right + 1);

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
```

```
    merge(arr, left, mid, right);
}
```

3. **Notes on Merge Sort in C++**

   - **Stable Sorting**: Merge Sort preserves the relative order of equal elements.
   - **Cache Behavior**: Merging involves additional memory, affecting locality.
   - **Practical Usage**: STL's `std::stable_sort` is based on optimized Merge Sort variants.

17.1.2 Quicksort

1. **Algorithm Overview**

   - **Divide**: Select a pivot and partition elements into two groups:
     - Left of pivot: smaller or equal elements.
     - Right of pivot: larger elements.
   - **Conquer**: Recursively sort both partitions.
   - **Combine**: Concatenation is implicit—no explicit merging.

   Time Complexity:

   - Best, Average: O(n log n).
   - Worst: $O(n^2)$ (poor pivot selection).
     Space Complexity:
   - O(log n) recursion stack (in-place sorting).

2. **C++ Implementation**

```cpp
#include <vector>
#include <algorithm>

int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high]; // Last element as pivot
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            std::swap(arr[++i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(std::vector<int>& arr, int low, int high) {
    if (low >= high) return;
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
}
```

3. **Pivot Strategies**

   - **Naive (last element)**: Simple but prone to worst-case $O(n^2)$.
   - **Randomized pivot**: Reduce likelihood of worst-case.
   - **Median-of-three**: Choose median of first, middle, last elements—improves balance.

4. **Notes on Quicksort in C++**

- **In-Place Sorting**: Requires no additional arrays.
- **Unstable**: Equal elements may swap order.
- **Practical Usage**: STL's `std::sort` uses **introsort** (Quicksort + Heapsort fallback).

17.1.3 Recursion Patterns in C++

1. **Standard Recursion**

   Both Merge Sort and Quicksort rely on recursive function calls. Each recursive call operates on smaller subproblems until base cases are reached.

2. **Tail Recursion**

   A function is **tail recursive** if the recursive call is the last operation before returning.
   Example:

```
void tailRecQuickSort(std::vector<int>& arr, int low, int high) {
    while (low < high) {
        int pi = partition(arr, low, high);
        // Recurse only on the smaller partition (tail recursion elimination)
        if (pi - low < high - pi) {
            tailRecQuickSort(arr, low, pi - 1);
            low = pi + 1; // Tail recursion eliminated
        } else {
            tailRecQuickSort(arr, pi + 1, high);
            high = pi - 1;
        }
    }
}
```

Advantages:

- Eliminates deep recursion.
- Reduces stack overflow risk on large inputs.
- Modern compilers may optimize tail recursion automatically.

3. **Iterative Transformations**

   Recursive algorithms can be transformed into **explicit iterative forms** using stacks:

```
void iterativeQuickSort(std::vector<int>& arr) {
    std::vector<std::pair<int,int>> stack;
    stack.push_back({0, (int)arr.size() - 1});

    while (!stack.empty()) {
        auto [low, high] = stack.back();
        stack.pop_back();
        if (low < high) {
            int pi = partition(arr, low, high);
            stack.push_back({low, pi - 1});
            stack.push_back({pi + 1, high});
        }
    }
}
```

Notes:

- Explicit stack simulates recursion.
- Useful for systems with limited recursion depth.

17.1.4 Comparative Summary

Algorithm	Time Complexity	Space Complexity	Stability	Practical Usage
Merge Sort	$O(n \log n)$ worst/avg/best	$O(n)$	Stable	`std::stable_sort`
Quicksort	$O(n \log n)$ avg, $O(n^2)$ worst	$O(\log n)$ stack	Unstable	`std::sort` (with introsort hybrid)

17.1.5 Exercises

1. Implement Merge Sort with **iterative bottom-up merging** instead of recursion.
2. Implement Quicksort with:
   - Randomized pivot
   - Median-of-three pivot
     Compare performance on sorted vs random arrays.
3. Convert recursive Merge Sort into an **iterative version** using explicit stacks.
4. Demonstrate **tail recursion elimination** in Quicksort and analyze stack depth reduction.
5. Compare **Merge Sort vs Quicksort** on large datasets with different distributions (random, sorted, reverse-sorted).

17.2 Parallel Divide-and-Conquer with `std::execution` and Thread Pools

The **divide-and-conquer paradigm** is inherently parallelizable because subproblems can often be solved independently. Modern C++ (since C++17 and further improved in C++20/23) provides **standardized tools for parallel execution** through the `<execution>` library, while **thread pools** allow fine-grained control over concurrency. Combining these techniques enables scalable implementations of classic divide-and-conquer algorithms like Merge Sort, Quicksort, and matrix multiplication.

17.2.1 Parallel Divide-and-Conquer: Conceptual Overview

- **Divide**: Split the input problem into smaller independent subproblems.
- **Conquer**: Solve each subproblem in parallel.
- **Combine**: Merge results.

Why it fits parallelism:

- Independent subproblems → parallel tasks.
- Balanced recursive tree → near logarithmic depth with high parallelism.
- Tail recursion + cutoff thresholds → avoid excessive thread creation.

17.2.2 `std::execution` in Parallel Divide-and-Conquer

C++17 introduced **parallel execution policies** in `<execution>`:

- `std::execution::seq`: Sequential execution (default).

- `std::execution::par`: Parallel execution (tasks may run on multiple threads).
- `std::execution::par_unseq`: Parallel + vectorized execution.

1. **Example: Parallel Sort**

   The standard library's `std::sort` can be parallelized via execution policies:

```
#include <algorithm>
#include <execution>
#include <vector>

int main() {
    std::vector<int> data = {9, 2, 5, 1, 7, 3};
    std::sort(std::execution::par, data.begin(), data.end());
}
```

   Internally, this uses **divide-and-conquer parallelism** (often introsort with thread pool support) when `std::execution::par` is selected.

2. **Limitations**

   - Current `<execution>` implementations vary across compilers and standard libraries.
   - Fine-grained task control (e.g., custom cutoff thresholds, recursion strategies) is not directly exposed.
   - For advanced workloads, custom **thread pools** are required.

17.2.3 Thread Pools in Divide-and-Conquer

A **thread pool** manages a fixed set of worker threads, avoiding the overhead of creating and destroying threads repeatedly. Divide-and-conquer recursion can submit

tasks to the pool, allowing **bounded parallelism**.

1. **Minimal Thread Pool Implementation**

```
#include <vector>
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <future>

class ThreadPool {
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> tasks;
    std::mutex mtx;
    std::condition_variable cv;
    bool stop = false;

public:
    ThreadPool(size_t threads) {
        for (size_t i = 0; i < threads; ++i) {
            workers.emplace_back([this] {
                while (true) {
                    std::function<void()> task;
                    {
                        std::unique_lock<std::mutex> lock(mtx);
                        cv.wait(lock, [this]{ return stop || !tasks.empty();
                        ↪  });
                        if (stop && tasks.empty()) return;
                        task = std::move(tasks.front());
                        tasks.pop();
                    }
```

```
                    task();
                }
            });
        }
    }

    template<class F, class... Args>
    auto enqueue(F&& f, Args&&... args) {
        using Ret = std::invoke_result_t<F, Args...>;
        auto task = std::make_shared<std::packaged_task<Ret()>>(
            std::bind(std::forward<F>(f), std::forward<Args>(args)...)
        );
        std::future<Ret> res = task->get_future();
        {
            std::lock_guard<std::mutex> lock(mtx);
            tasks.emplace([task]{ (*task)(); });
        }
        cv.notify_one();
        return res;
    }

    ~ThreadPool() {
        {
            std::lock_guard<std::mutex> lock(mtx);
            stop = true;
        }
        cv.notify_all();
        for (auto& worker : workers) worker.join();
    }
};
```

2. **Divide-and-Conquer Example: Parallel Merge Sort**

```
void parallelMergeSort(std::vector<int>& arr, int left, int right, ThreadPool&
↪  pool, int depth = 0) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;

    if (depth < 3) { // control parallel recursion depth
        auto leftFuture = pool.enqueue([&]{ parallelMergeSort(arr, left, mid,
        ↪  pool, depth+1); });
        parallelMergeSort(arr, mid+1, right, pool, depth+1);
        leftFuture.get();
    } else {
        parallelMergeSort(arr, left, mid, pool, depth+1);
        parallelMergeSort(arr, mid+1, right, pool, depth+1);
    }

    merge(arr, left, mid, right);
}
```

Features:

- Parallelism is limited by `depth` to avoid task explosion.
- Workloads naturally balance at higher recursion levels.
- Thread pool amortizes thread management costs.

17.2.4 Hybrid Approach: Execution Policies + Custom Thread Pools

1. Use **`std::execution`** for **coarse-grained parallel algorithms** (sorting, transformations).
2. Use **thread pools** for **fine-grained recursive control**.

3. Combine both to scale across different workloads.

Example:

- Parallel merge sort for large datasets.
- Sequential sort for small subarrays (base case).
- Thread pool used internally for recursive task scheduling.

17.2.5 Performance Considerations

- **Granularity Control**: Avoid spawning threads for small tasks—introduce cutoff thresholds.
- **Load Balancing**: Thread pools distribute tasks evenly; uneven partitions (Quicksort on skewed data) can reduce efficiency.
- **Memory Locality**: Excessive parallelism may degrade cache performance.
- **Synchronization Overhead**: Locks, atomics, and futures add overhead—use them only when necessary.

17.2.6 Exercises

1. Implement **parallel quicksort** using a custom thread pool. Compare with sequential quicksort.
2. Use `std::execution::par_unseq` with `std::transform` to apply a divide-and-conquer vector transformation in parallel.
3. Modify **parallel merge sort** to use `std::stable_sort` with `std::execution::par` for the base case.

4. Extend the thread pool to support **work stealing**, reducing idle threads when partitions are unbalanced.

5. Benchmark: Compare performance of

    - Sequential Merge Sort
    - `std::sort(std::execution::par)`
    - Custom thread pool merge sort

17.2.7 Summary

- Divide-and-conquer is naturally parallelizable, and modern C++ provides **standard execution policies** for ease of use.
- For advanced workloads, **custom thread pools** provide more control over recursion depth, task granularity, and scheduling.
- A **hybrid approach**—execution policies for coarse-grained tasks, thread pools for fine-grained control—delivers robust and scalable divide-and-conquer solutions in modern C++.

17.3 Exercises — Median of Medians, Parallel Mergesort

Exercises are designed to help students and practitioners **internalize divide-and-conquer strategies** by implementing classic algorithms that highlight different aspects of the paradigm. Two essential problems are explored here: the **Median of Medians algorithm** for deterministic selection and a **Parallel Merge Sort** leveraging modern C++ parallelism and thread pools.

17.3.1 Exercise: Median of Medians

The **Median of Medians** algorithm addresses the problem of finding the *k-th smallest element* (selection) in an unsorted array in deterministic linear time. Unlike randomized Quickselect, which has expected linear time but worst-case quadratic time, Median of Medians guarantees **O(n)** performance.

1. **Divide-and-Conquer Strategy**

    - **Divide**: Partition the array into groups of five elements each.
    - **Conquer**: Find the median of each group (via sorting, since the groups are small).
    - **Recurse**: Use Median of Medians recursively to find the median of the medians.
    - **Partition**: Use this pivot to split the array into less-than and greater-than parts.
    - **Select**: Depending on k, recurse into the appropriate partition.

2. **Implementation Sketch in Modern C++**

```cpp
#include <algorithm>
#include <vector>

int partition(std::vector<int>& arr, int left, int right, int pivotIndex) {
    int pivotValue = arr[pivotIndex];
    std::swap(arr[pivotIndex], arr[right]);
    int storeIndex = left;
    for (int i = left; i < right; i++) {
        if (arr[i] < pivotValue) {
            std::swap(arr[storeIndex++], arr[i]);
        }
    }
    std::swap(arr[right], arr[storeIndex]);
    return storeIndex;
}

int medianOfMedians(std::vector<int>& arr, int left, int right, int k) {
    if (left == right) return arr[left];

    int n = right - left + 1;
    std::vector<int> medians;
    for (int i = 0; i < n/5; i++) {
        std::sort(arr.begin() + left + i*5, arr.begin() + left + i*5 + 5);
        medians.push_back(arr[left + i*5 + 2]); // median of 5
    }
    if (n % 5) {
        std::sort(arr.begin() + left + (n/5)*5, arr.begin() + right + 1);
        medians.push_back(arr[left + (n/5)*5 + (n%5)/2]);
    }

    int median = (medians.size() == 1) ? medians[0]
                : medianOfMedians(medians, 0, medians.size()-1,
                ↪  medians.size()/2);
```

```
    // partition around the chosen median
    int pivotIndex = std::find(arr.begin()+left, arr.begin()+right+1, median) -
    ↪  arr.begin();
    pivotIndex = partition(arr, left, right, pivotIndex);

    if (k == pivotIndex) return arr[k];
    else if (k < pivotIndex) return medianOfMedians(arr, left, pivotIndex-1,
    ↪  k);
    else return medianOfMedians(arr, pivotIndex+1, right, k);
}
```

3. **Key Points**

   - Groups of five ensure balanced partitions, avoiding worst-case quadratic behavior.
   - Time complexity: **O(n)** (deterministic).
   - Exercises:
     - Implement the algorithm with different group sizes (e.g., 3, 7) and compare performance.
     - Compare deterministic Median of Medians with randomized Quickselect for large datasets.
     - Extend to multidimensional median finding.

17.3.2 Exercise: Parallel Merge Sort

Merge Sort is a textbook **divide-and-conquer algorithm**. Its recursive structure makes it ideal for **parallelization**. The parallel implementation enhances performance on multi-core architectures by sorting subarrays concurrently.

1. **Divide-and-Conquer Strategy**

   - **Divide**: Recursively split the array into two halves.
   - **Conquer**: Sort both halves in parallel.
   - **Combine**: Merge sorted halves into a final sorted array.

2. **Parallel Merge Sort with Thread Pool**

```
#include <vector>
#include <future>
#include <algorithm>

void merge(std::vector<int>& arr, int left, int mid, int right) {
    std::vector<int> leftVec(arr.begin() + left, arr.begin() + mid + 1);
    std::vector<int> rightVec(arr.begin() + mid + 1, arr.begin() + right + 1);

    int i = 0, j = 0, k = left;
    while (i < (int)leftVec.size() && j < (int)rightVec.size()) {
        arr[k++] = (leftVec[i] <= rightVec[j]) ? leftVec[i++] : rightVec[j++];
    }
    while (i < (int)leftVec.size()) arr[k++] = leftVec[i++];
    while (j < (int)rightVec.size()) arr[k++] = rightVec[j++];
}

void parallelMergeSort(std::vector<int>& arr, int left, int right, int depth =
↪  0) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;

    if (depth < 3) { // limit recursion depth for parallelism
        auto leftFuture = std::async(std::launch::async, [&]{
        ↪  parallelMergeSort(arr, left, mid, depth+1); });
```

```
        parallelMergeSort(arr, mid+1, right, depth+1);
        leftFuture.get();
    } else {
        parallelMergeSort(arr, left, mid, depth+1);
        parallelMergeSort(arr, mid+1, right, depth+1);
    }

    merge(arr, left, mid, right);
}
```

3. **Modern C++ Execution Policy Variant**

   C++17 also allows:

```
#include <algorithm>
#include <execution>

std::sort(std::execution::par, arr.begin(), arr.end());
```

   While concise, this delegates details to the standard library and does not expose recursion depth or scheduling strategies. The custom implementation above allows fine-tuned **divide-and-conquer parallelism**.

4. **Exercises**

   (a) Implement **parallel mergesort** with:

   - `std::async`
   - Custom thread pool
   - `std::execution::par`

   Compare performance across methods.

(b) Experiment with different **parallel recursion cutoffs** (depth 2, 3, 4). Measure trade-offs between task overhead and speedup.

(c) Extend parallel mergesort to **external memory sorting** (disk-based mergesort), simulating large datasets that exceed memory capacity.

(d) Combine mergesort with **Median of Medians** for pivot-based partitioning experiments.

17.3.3 Summary

- The **Median of Medians** algorithm demonstrates deterministic selection with linear-time guarantees, making it a powerful example of the divide-and-conquer paradigm applied to order statistics.
- **Parallel Mergesort** illustrates how recursion maps naturally to parallel execution. With controlled task granularity, it achieves scalable performance on multi-core systems.
- These exercises strengthen mastery of **divide-and-conquer**, from deterministic selection to scalable parallel sorting, ensuring both theoretical understanding and practical application in modern C++.

Chapter 18

Dynamic Programming (DP)

18.1 Memoization vs. Tabulation — Idiomatic C++ Patterns

Dynamic Programming (DP) is one of the most versatile algorithmic techniques, widely used to solve problems involving **overlapping subproblems** and **optimal substructure**. At its core, DP can be implemented in two main ways: **memoization** (top-down recursion with caching) and **tabulation** (bottom-up iterative filling). In modern C++, each style benefits from idiomatic constructs such as `unordered_map`, `vector`, and views for iteration and efficient data representation.

18.1.1 Memoization (Top-Down Dynamic Programming)

Memoization augments a recursive solution by storing already-computed results in a cache, preventing repeated calculations. It retains the **recursive problem decomposition**, making code often easier to reason about, but requires careful handling of recursion limits and cache lookups.

1. **Idiomatic C++ Patterns**

   - **Cache with `unordered_map`**: Useful when subproblem states are sparse or not easily bounded (e.g., memoizing `(i, j)` states in grid DP).
   - **Cache with `vector`**: Preferable when the state space is well-bounded and indexable by integers.

2. **Example: Fibonacci with Memoization**

```cpp
#include <unordered_map>
#include <iostream>
```

```cpp
long long fib(int n, std::unordered_map<int, long long>& memo) {
    if (n <= 1) return n;
    if (memo.count(n)) return memo[n];
    return memo[n] = fib(n-1, memo) + fib(n-2, memo);
}

int main() {
    std::unordered_map<int, long long> memo;
    std::cout << fib(50, memo) << "\n"; // efficient due to caching
}
```

3. **Pros and Cons**

   - **Pros**:
     - Natural expression of recursion.
     - Flexibility with irregular state spaces.
   - **Cons**:
     - Recursion depth can cause stack overflows.
     - Overhead from hash lookups (`unordered_map`) if the state space is dense.

18.1.2 Tabulation (Bottom-Up Dynamic Programming)

Tabulation iteratively fills in a DP table starting from base cases up to the desired result. This avoids recursion and typically ensures better **constant-factor performance**. It requires knowing the full state space in advance.

1. **Idiomatic C++ Patterns**

- **`std::vector` for dense states**: Default choice for array-like DP tables.
- **Views (`std::ranges::views`)**: Allow expressive iteration and cleaner code when filling DP tables.

2. **Example: Fibonacci with Tabulation**

```cpp
#include <vector>
#include <iostream>

long long fib(int n) {
    if (n <= 1) return n;
    std::vector<long long> dp(n+1, 0);
    dp[1] = 1;
    for (int i = 2; i <= n; ++i) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}

int main() {
    std::cout << fib(50) << "\n"; // fast, no recursion overhead
}
```

3. **Pros and Cons**

- **Pros**:
  - Avoids recursion stack issues.
  - Often faster for dense, bounded problems.
- **Cons**:
  - Less intuitive for recursive problem formulations.

- Requires precomputing and storing the entire table, which can be memory-intensive.

18.1.3 Comparing Memoization vs. Tabulation in C++

Aspect	Memoization	Tabulation
Style	Top-down recursion with caching	Bottom-up iterative filling
Data Structure	`unordered_map`, `vector`	`vector` (dense), sometimes `array`
Best For	Sparse/unbounded states	Dense/bounded states
Performance	Slightly higher constant overhead	Generally faster and cache-friendly
Ease of Expression	Mirrors recursive definition	More mechanical but stack-safe
Memory Use	On-demand (only visited states)	Allocates full table upfront

18.1.4 Advanced Idiomatic Patterns

1. **`std::optional` for Safe Caching**

   Instead of using `unordered_map`, a `std::vector<std::optional<T>>` can represent uninitialized states efficiently when indices are bounded.

```cpp
#include <vector>
#include <optional>

long long fib(int n, std::vector<std::optional<long long>>& dp) {
    if (n <= 1) return n;
    if (dp[n]) return *dp[n];
    return *(dp[n] = fib(n-1, dp) + fib(n-2, dp));
}
```

2. **Iteration with Views**

   Using C++20 ranges:

```cpp
#include <ranges>
#include <vector>
#include <iostream>

int main() {
    int n = 10;
    std::vector<long long> dp(n+1, 0);
    dp[1] = 1;

    for (int i : std::views::iota(2, n+1)) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    std::cout << dp[n] << "\n";
}
```

   This improves readability and idiomatic use of modern C++.

18.1.5 Summary

- **Memoization** is best suited for problems with **sparse or irregular subproblems**, where caching avoids unnecessary computation. In idiomatic C++, `unordered_map` offers flexible caching, while `vector` provides fast indexable storage.
- **Tabulation** excels in **dense, bounded state spaces**, leveraging `vector` for performance and memory locality, with modern enhancements like **ranges/views** for expressive iteration.
- Choosing between the two depends on **problem characteristics**: recursion friendliness, memory constraints, and whether the state space is known in advance.
- Both approaches should be part of a C++ programmer's toolkit for building efficient and elegant DP solutions.

18.2 DP on Sequences, Trees, and Graphs — Common Templates and Optimizations (Space Reduction)

Dynamic Programming manifests differently depending on whether the problem domain is a **linear sequence**, a **tree**, or a **general graph**. Each structure imposes unique recursion patterns, state definitions, and opportunities for optimization. In C++, these can be expressed cleanly through reusable **templates**, memory-conscious **containers**, and careful **space reduction** techniques.

18.2.1 DP on Sequences

Sequences (arrays, strings, lists) are the most common context for DP. Problems like **Longest Common Subsequence (LCS)**, **Edit Distance**, or **Knapsack** rely on analyzing prefixes or indices.

1. **Template Pattern**

   Typical DP recurrence:

   $$dp[i][j] = f(dp[i-1][j], dp[i][j-1], dp[i-1][j-1], \dots)$$

   where `i`, `j` are sequence indices.

2. **C++ Example: Longest Common Subsequence**

```cpp
#include <vector>
#include <string>
#include <algorithm>
```

```
int lcs(const std::string& a, const std::string& b) {
    int n = a.size(), m = b.size();
    std::vector<std::vector<int>> dp(n+1, std::vector<int>(m+1, 0));

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (a[i-1] == b[j-1])
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = std::max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[n][m];
}
```

3. **Space Optimization**

   Because only the previous row is needed, memory can be reduced from `O(n*m)` to `O(min(n, m))`.

```
int lcsOptimized(const std::string& a, const std::string& b) {
    int n = a.size(), m = b.size();
    std::vector<int> prev(m+1, 0), curr(m+1, 0);

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (a[i-1] == b[j-1]) curr[j] = prev[j-1] + 1;
            else curr[j] = std::max(prev[j], curr[j-1]);
        }
        std::swap(prev, curr);
    }
```

```
    return prev[m];
}
```

18.2.2 DP on Trees

Tree DP solves problems such as **subtree sums**, **diameter**, or **maximum independent set on a tree**. The structure naturally supports recursion, with subproblem states defined per node and combined over children.

1. **Template Pattern**

   General recurrence:

$$dp[u] = g(\{dp[v] : v \in \text{children}(u)\})$$

2. **C++ Example: Maximum Independent Set on a Tree**

```
#include <vector>
#include <algorithm>

struct TreeDP {
    int n;
    std::vector<std::vector<int>> adj;
    std::vector<std::vector<int>> dp; // dp[u][0]=not taken, dp[u][1]=taken
    std::vector<bool> visited;

    TreeDP(int n) : n(n), adj(n), dp(n, std::vector<int>(2, 0)),
    ↪  visited(n,false) {}

    void addEdge(int u, int v) {
```

```
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    void dfs(int u) {
        visited[u] = true;
        dp[u][0] = 0;
        dp[u][1] = 1;
        for (int v : adj[u]) {
            if (!visited[v]) {
                dfs(v);
                dp[u][0] += std::max(dp[v][0], dp[v][1]);
                dp[u][1] += dp[v][0];
            }
        }
    }
};
```

3. **Optimizations**

   - **Re-rooting DP**: Sometimes results must be computed with each node as root (e.g., subtree sums). Efficient re-rooting transforms allow `O(n)` overall complexity.
   - **Iterative DFS**: Avoid recursion depth issues by using explicit stacks.

18.2.3 DP on Graphs

DP on graphs is more general and often intertwined with **shortest path algorithms** or **DAG DP**.

1. **DAG DP**

For Directed Acyclic Graphs:

$$dp[u] = \max_{(u,v)\in E}(w(u,v) + dp[v])$$

2. **Example: Longest Path in DAG**

```
#include <vector>
#include <stack>
#include <algorithm>

void topoSortUtil(int u, const std::vector<std::vector<int>>& adj,
                  std::vector<bool>& visited, std::stack<int>& st) {
    visited[u] = true;
    for (int v : adj[u]) if (!visited[v]) topoSortUtil(v, adj, visited, st);
    st.push(u);
}

int longestPathDAG(int n, const std::vector<std::vector<int>>& adj) {
    std::stack<int> st;
    std::vector<bool> visited(n, false);
    for (int i = 0; i < n; i++) if (!visited[i]) topoSortUtil(i, adj, visited,
    ↪  st);

    std::vector<int> dp(n, 0);
    while (!st.empty()) {
        int u = st.top(); st.pop();
        for (int v : adj[u]) {
            dp[v] = std::max(dp[v], dp[u] + 1);
        }
    }
    return *std::max_element(dp.begin(), dp.end());
}
```

3. **Graph DP Optimizations**

   - **Topological ordering**: Ensures linear-time evaluation for DAGs.
   - **Memoization on cycles**: For cyclic graphs, memoization + cycle detection prevents infinite recursion.
   - **Bitset compression**: For subset-DP problems (e.g., traveling salesman), bitsets can dramatically reduce memory and enable vectorized operations.

18.2.4 Space Reduction Techniques

1. **Rolling Arrays**

   Only store necessary slices of the DP table (e.g., last row, last column).

2. **In-Place DP**

   Reusing input containers (`vector`, `string`) to store states.

3. **Bitset Optimizations**

   Use `std::bitset` or `std::vector<bool>` for problems where state is binary (e.g., knapsack, subset sums).

```
#include <bitset>
const int MAXW = 10000;
std::bitset<MAXW+1> knapsack;
```

4. **Sparse State Representation**

   When the state space is sparse, prefer `unordered_map` instead of `vector` to minimize memory.

18.2.5 Summary

- **Sequences**: Standard tabular DP with opportunities for **row/column compression**.
- **Trees**: Natural recursion templates with subtree aggregation, optimized via **re-rooting** and **iterative traversal**.
- **Graphs**: Require **topological ordering** (DAGs) or careful memoization (cyclic graphs), often blending with shortest path techniques.
- **Space Reduction**: Achieved via rolling arrays, in-place updates, and bitset compression, enabling scalability to larger problems.

Dynamic Programming in C++ is not only about correct recurrence formulation but also about **writing memory- and cache-efficient code** that scales. Idiomatic use of `vector`, `unordered_map`, `bitset`, and even modern C++ features like ranges leads to concise, performant solutions.

18.3 Exercises: Knapsack Variants, Longest Increasing Subsequence with Patience Sorting (O(n log n))

Dynamic Programming (DP) is best mastered by tackling classic problems and then extending them into more challenging variants. Two cornerstone exercises that test both theoretical understanding and implementation skills in C++ are **knapsack problems** and the **Longest Increasing Subsequence (LIS)**. Together, they illustrate how to handle optimization under constraints and how to apply DP in conjunction with advanced algorithmic techniques such as binary search.

18.3.1 Knapsack Variants

The **0/1 Knapsack Problem** is a standard DP exercise: given a set of items, each with a weight and value, determine the maximum total value achievable under a capacity constraint. The state transition typically looks like this:

$$dp[i][w] = \max(dp[i-1][w],\ dp[i-1][w-w_i] + v_i) \quad \text{if } w \geq w_i$$

Here `i` indexes items, and `w` indexes current capacity.

- **a. Classic 0/1 Knapsack in C++**

```
int knapsack01(const vector<int>& weights, const vector<int>& values, int W) {
    int n = weights.size();
    vector<int> dp(W + 1, 0);

    for (int i = 0; i < n; ++i) {
        for (int w = W; w >= weights[i]; --w) {
```

```
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i]);
        }
    }
    return dp[W];
}
```

This uses space optimization (`O(W)`) by rolling the DP array from back to front.

- **b. Bounded Knapsack**

  Each item can be chosen up to a certain multiplicity. One approach is **binary splitting**: decompose item counts into sums of powers of two, reducing bounded knapsack to multiple 0/1 items.

- **c. Unbounded (Complete) Knapsack**

  Items can be chosen infinitely. Transition order differs — we must process capacities **forward** to allow repeated usage:

```
int unboundedKnapsack(const vector<int>& weights, const vector<int>& values,
↪  int W) {
    int n = weights.size();
    vector<int> dp(W + 1, 0);

    for (int i = 0; i < n; ++i) {
        for (int w = weights[i]; w <= W; ++w) {
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i]);
        }
    }
    return dp[W];
}
```

- **d. Multi-dimensional Knapsack**

  Constraints extend to multiple resources (e.g., weight and volume). The DP table becomes higher-dimensional, which may require pruning or heuristic approximations for efficiency.

18.3.2 Longest Increasing Subsequence (LIS)

The LIS problem asks for the length of the longest strictly increasing subsequence in a sequence of integers.

- **a. O(n²) DP Solution**

  For each element, compute LIS ending at that position:

$$dp[i] = 1 + \max(dp[j]) \quad \text{for all } j < i,\ a[j] < a[i]$$

```
int lisQuadratic(const vector<int>& nums) {
    int n = nums.size();
    vector<int> dp(n, 1);
    int ans = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            if (nums[j] < nums[i]) dp[i] = max(dp[i], dp[j] + 1);
        }
        ans = max(ans, dp[i]);
    }
    return ans;
}
```

- **b. O(n log n) with Patience Sorting**

Patience sorting borrows from card-game strategies: maintain piles where each new number replaces the smallest possible pile top greater than it.

```
int lisPatience(const vector<int>& nums) {
    vector<int> piles;
    for (int x : nums) {
        auto it = lower_bound(piles.begin(), piles.end(), x);
        if (it == piles.end()) piles.push_back(x);
        else *it = x;
    }
    return piles.size();
}
```

This method efficiently computes LIS length. To reconstruct the sequence, additional predecessor tracking arrays must be maintained.

18.3.3 C++ Patterns for Efficiency

- Use **`std::vector`** for cache efficiency in large DP states.
- Employ **`std::lower_bound`** (binary search) for LIS patience sorting.
- Leverage **move semantics** and `reserve()` when handling large intermediate DP states.
- For multidimensional DP, consider **compressed views** or `std::span` (C++20) to reduce memory overhead.

18.3.4 Exercises

1. Implement **all knapsack variants** (0/1, unbounded, bounded, multi-dimensional) and compare their complexities and memory usage.

2. Extend LIS to recover the actual subsequence using parent pointers.

3. Solve the **weighted LIS problem** (maximize sum of values instead of length).

4. Explore the **Longest Bitonic Subsequence** (increasing then decreasing) by combining LIS and Longest Decreasing Subsequence (LDS).

Chapter 19

Greedy Algorithms & Matroid Concepts

19.1 Greedy Correctness Proofs and C++ Greedy Idioms

Greedy algorithms represent one of the most elegant paradigms in algorithm design. They work by making a **locally optimal choice at each step** with the hope (and often guarantee) that the final result will be globally optimal. However, the main challenge is not the implementation—usually straightforward—but proving correctness and identifying when greedy strategies apply.

This section explores two central aspects:

1. **Correctness proofs** of greedy algorithms.

2. **Idiomatic C++ patterns** for implementing greedy strategies efficiently.

19.1.1 Greedy Algorithm Correctness Proofs

Greedy algorithms are not universally applicable. Their correctness depends on structural properties of the problem. Two fundamental techniques are used to establish correctness:

- **a. The Greedy Choice Property**

  A problem exhibits the **greedy choice property** if a globally optimal solution can always be arrived at by making a locally optimal choice at each step.

  Example: Activity Selection Problem

  - Task: Select the maximum number of non-overlapping activities, each with start and finish times.
  - Greedy rule: Always select the activity with the earliest finishing time that is compatible with the previously chosen activities.

  Proof sketch:

  1. Let `A*` be an optimal solution.
  2. If `A*` does not include the earliest finishing activity, we can swap its first chosen activity with the earliest finishing one without reducing the solution size.
  3. This exchange argument shows that the greedy choice is safe and still leads to an optimal solution.

- **b. Exchange Argument**

  The **exchange argument** demonstrates that any deviation from the greedy solution can be adjusted step by step into the greedy solution without harming optimality.

Example: Huffman Coding

- Greedy step: Always merge the two least frequent symbols.
- Proof idea: Show that in an optimal prefix code tree, the two least frequent symbols must be siblings at the greatest depth. If they are not, we can exchange nodes to create a tree of no worse cost.

- **c. Matroid Theory Connection**

  Many greedy algorithms succeed because the underlying problem structure forms a **matroid**.

  - A matroid is a combinatorial structure that generalizes independence from linear algebra and graph theory.
  - In matroids, greedy algorithms always yield an optimal solution.
  - Classic example: **Maximum Weight Independent Set in a Matroid** — selecting maximum-weight edges in a graphic matroid corresponds to Kruskal's algorithm for Minimum Spanning Trees.

19.1.2 C++ Greedy Idioms

Greedy strategies often map naturally to idiomatic patterns in C++. Efficiency is critical, since greedy algorithms are usually applied to large inputs.

- **a. Sorting and Iteration**

  Sorting is frequently the preprocessing step in greedy algorithms (e.g., interval scheduling, Kruskal's MST).

```
struct Activity {
    int start, finish;
};
bool cmpFinish(const Activity& a, const Activity& b) {
    return a.finish < b.finish;
}

vector<Activity> activities = { {1, 3}, {2, 5}, {4, 6} };
sort(activities.begin(), activities.end(), cmpFinish);
```

Here, `std::sort` with a custom comparator forms the backbone of many greedy implementations.

- **b. Priority Queues (`std::priority_queue`)**

  When repeatedly selecting the "best available" element, a heap-based structure is idiomatic.

```
priority_queue<int, vector<int>, greater<int>> minHeap; // for earliest
↪  deadlines
```

  This structure supports efficient extraction of minimum/maximum values. Used in problems like Huffman coding and interval partitioning.

- **c. Greedy with Maps and Sets**

  For problems requiring dynamic tracking of available resources, **ordered containers** (`std::set`, `std::multiset`) provide logarithmic-time insert/erase.

```
multiset<int> rooms; // track room availability times
```

- **d. Greedy with Custom Comparators**

  C++'s flexibility in defining **lambdas** and **functors** allows tailoring greedy strategies:

```
auto cmp = [](const pair<int,int>& a, const pair<int,int>& b) {
    return a.second > b.second; // prioritize smaller finish time
};
priority_queue<pair<int,int>, vector<pair<int,int>>, decltype(cmp)> pq(cmp);
```

- **e. Greedy and Range Views (C++20)**

  C++20 ranges simplify greedy preprocessing pipelines:

```
#include <ranges>

auto sorted = activities
    | std::views::filter([](auto a){ return a.start >= 0; })
    | std::views::transform([](auto a){ return make_pair(a.start, a.finish);
    ↪  });
```

  This functional style emphasizes clarity while remaining efficient.

19.1.3 Typical Greedy Patterns in Practice

1. **Interval scheduling** → sort by finish time, greedy selection.
2. **Minimum Spanning Tree (Kruskal's)** → greedy edge addition.
3. **Huffman coding** → greedy frequency merging with a priority queue.
4. **Fractional Knapsack** → greedy by value-to-weight ratio.

5. **Job sequencing with deadlines** → greedy scheduling with disjoint-set or multiset.

19.1.4 Key Takeaways for Graduate-Level Readers

- **Correctness is the hard part:** Proofs usually hinge on greedy choice property or exchange arguments.
- **Matroid perspective generalizes greedy success:** Understanding this concept helps in identifying when greedy will (or won't) work.
- **C++ provides direct support for greedy idioms:** Sorting, heaps, and ordered sets/maps form the foundation.
- **Efficiency considerations:** Use `std::priority_queue` for repeated selections, avoid unnecessary copies with move semantics, and leverage C++20 ranges for concise preprocessing.

19.2 Huffman Coding with Heaps and `std::priority_queue` Customization

Huffman coding is one of the most celebrated greedy algorithms in computer science. It provides an **optimal prefix code** for a set of symbols with known frequencies, minimizing the total cost of encoding messages. Its impact spans compression standards such as **JPEG, MP3, PNG, and DEFLATE (used in ZIP, GZIP)**. From an algorithmic perspective, Huffman coding is an archetypal example of the **greedy paradigm**, combined with efficient use of **heaps** for implementation.
This section discusses the theory, correctness, and detailed C++ implementations of Huffman coding, highlighting the use of `std::priority_queue` and custom comparators.

19.2.1 Problem Setting

Given:

- A set of `n` symbols with associated frequencies (probabilities of occurrence).
- Goal: Assign binary codes to each symbol such that:
    1. Codes are **prefix-free** (no code is a prefix of another).
    2. The total cost `Σ (frequency × code length)` is minimized.

19.2.2 Greedy Insight

The key greedy principle:

- At each step, **merge the two least frequent symbols** into a new combined symbol (with frequency equal to the sum of both).

- Repeat until one symbol remains (the root of the tree).

Why does this work?

- **Exchange argument:** In any optimal prefix code, the two least frequent symbols appear as siblings at maximum depth. If they are not, swapping them does not increase cost but restores greedy structure.
- This recursive structure guarantees global optimality.

19.2.3 Heap-Based Algorithm

The algorithm relies on an efficient **priority queue (min-heap)** to repeatedly extract the least frequent symbols.

Pseudocode

1. Insert all frequencies into a min-heap.
2. While heap size > 1:
    - Extract two smallest elements.
    - Create a new node with weight = sum of both.
    - Insert this node back into the heap.
3. The remaining node is the root of the Huffman tree.

19.2.4 C++ Implementation with `std::priority_queue`

By default, `std::priority_queue` is a **max-heap**. To use it as a **min-heap**, we need a **custom comparator**.

- **Node Structure**

```
struct Node {
    char symbol;
    int freq;
    Node* left;
    Node* right;

    Node(char s, int f) : symbol(s), freq(f), left(nullptr), right(nullptr) {}
};
```

- **Comparator for Min-Heap**

```
struct Compare {
    bool operator()(Node* a, Node* b) {
        return a->freq > b->freq; // higher freq = lower priority
    }
};
```

- **Priority Queue and Tree Construction**

```
#include <queue>
#include <vector>
#include <iostream>
using namespace std;

Node* buildHuffmanTree(const vector<pair<char,int>>& freq) {
    priority_queue<Node*, vector<Node*>, Compare> pq;

    // Insert all symbols
    for (auto& [ch, f] : freq)
```

```
        pq.push(new Node(ch, f));

    // Combine until single root remains
    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();

        Node* merged = new Node('\0', left->freq + right->freq);
        merged->left = left;
        merged->right = right;

        pq.push(merged);
    }
    return pq.top();
}
```

- **Traversal to Extract Codes**

```
void generateCodes(Node* root, string code,
                   unordered_map<char, string>& codes) {
    if (!root) return;

    if (root->symbol != '\0') { // leaf node
        codes[root->symbol] = code;
    }
    generateCodes(root->left, code + "0", codes);
    generateCodes(root->right, code + "1", codes);
}
```

Usage:

```
int main() {
    vector<pair<char,int>> freq = {
        {'a', 5}, {'b', 9}, {'c', 12},
        {'d', 13}, {'e', 16}, {'f', 45}
    };

    Node* root = buildHuffmanTree(freq);
    unordered_map<char, string> codes;
    generateCodes(root, "", codes);

    for (auto& [ch, code] : codes)
        cout << ch << ": " << code << "\n";
}
```

19.2.5 Performance Analysis

- Building the tree requires `O(n log n)` operations:
  - Each of the `n-1` merges involves 2 heap extractions and 1 insertion (`O(log n)` each).
- Space complexity: `O(n)` nodes.
- Code extraction is `O(n)` in the number of symbols.

This efficiency makes Huffman coding practical even for large alphabets in compression systems.

19.2.6 C++ Idioms and Customization

- **a. Using Lambdas for Comparators**

Instead of defining a struct, we can inline the comparator with a lambda:

```
auto cmp = [](Node* a, Node* b) { return a->freq > b->freq; };
priority_queue<Node*, vector<Node*>, decltype(cmp)> pq(cmp);
```

- **b. Smart Pointers for Safety**

  To avoid manual memory management:

  ```
  using NodePtr = shared_ptr<Node>;
  ```

  This integrates well with modern C++ and ensures no memory leaks.

- **c. Move Semantics**

  When symbols or frequency data are large, prefer `std::move` when inserting into the heap to reduce overhead.

19.2.7 Broader Connections

- Huffman coding exemplifies **greedy correctness via exchange argument**.
- It highlights **priority queues** as a fundamental tool in greedy algorithms.
- In graduate-level study, it connects to **information theory**, where the average code length approaches the **entropy bound**.

19.2.8 Key Takeaways

1. Huffman coding is the **canonical greedy algorithm** for compression.

2. Its correctness rests on **exchange arguments** ensuring least frequent symbols are merged first.

3. Efficient implementation relies on **heaps**, directly supported by `std::priority_queue`.

4. Modern C++ allows safe, flexible, and idiomatic implementations via **custom comparators, lambdas, and smart pointers**.

5. Huffman coding illustrates how **algorithmic theory** and **systems-level implementation** converge in practice.

19.3 Exercises: Activity Selection, Interval Scheduling

Greedy algorithms often shine in **scheduling and resource allocation problems** where the challenge is to maximize throughput or minimize conflicts under simple constraints. Two classic problems that highlight this paradigm are the **Activity Selection Problem** and the more general **Interval Scheduling Problem**. These exercises reinforce how greedy strategies, grounded in rigorous proofs, map to **clear and efficient C++ implementations**.

19.3.1 Activity Selection Problem

- **Problem Statement**

  Given `n` activities, each with a **start time** and a **finish time**, select the maximum number of activities that can be performed by a single person, assuming that one activity must finish before the next one starts.

- **Greedy Insight**

  - Always select the activity that finishes **earliest** among those compatible with the already chosen ones.
  - Proof of correctness comes from the **greedy choice property**: replacing any first activity in an optimal solution with the earliest finishing activity does not reduce the solution size.

- **Algorithm Steps**

  1. Sort activities by **finish time**.

2. Select the first activity.
3. Iteratively select the next activity whose start time is finish time of the last chosen activity.

- **C++ Implementation**

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Activity {
    int start, finish;
};

bool cmpFinish(const Activity& a, const Activity& b) {
    return a.finish < b.finish;
}

vector<Activity> activitySelection(vector<Activity>& acts) {
    sort(acts.begin(), acts.end(), cmpFinish);

    vector<Activity> result;
    int lastFinish = -1;

    for (const auto& act : acts) {
        if (act.start >= lastFinish) {
            result.push_back(act);
            lastFinish = act.finish;
        }
    }
    return result;
}
```

```
int main() {
    vector<Activity> acts = { {1, 3}, {2, 5}, {0, 6}, {5, 7},
                              {8, 9}, {5, 9} };
    auto chosen = activitySelection(acts);
    for (auto& act : chosen) {
        cout << "(" << act.start << ", " << act.finish << ") ";
    }
}
```

- **Complexity**
  - Sorting: `O(n log n)`
  - Selection: `O(n)`
  - Total: `O(n log n)`

19.3.2 Interval Scheduling Problem

The interval scheduling problem generalizes activity selection:

- **Input:** A set of intervals `[s , f )` with possible overlaps.
- **Goal:** Find the largest subset of mutually compatible intervals.

The greedy strategy remains identical:

- Sort by **earliest finish time**.
- Iteratively select compatible intervals.

This generalization extends to real-world problems such as:

- Meeting room allocation.
- CPU job scheduling.
- Airplane runway slot assignment.

C++ Example with Meetings

```
struct Interval {
    int start, end;
};

bool cmpInterval(const Interval& a, const Interval& b) {
    return a.end < b.end;
}

int maxNonOverlappingIntervals(vector<Interval>& intervals) {
    sort(intervals.begin(), intervals.end(), cmpInterval);
    int count = 0, lastEnd = -1;

    for (auto& iv : intervals) {
        if (iv.start >= lastEnd) {
            ++count;
            lastEnd = iv.end;
        }
    }
    return count;
}

int main() {
    vector<Interval> meetings = { {1, 4}, {3, 5}, {0, 6}, {5, 7},
                                  {8, 9}, {5, 9}, {2, 14}, {12, 16} };
    cout << "Maximum non-overlapping meetings: "
```

```
        << maxNonOverlappingIntervals(meetings) << endl;
}
```

19.3.3 Exercises for the Reader

1. **Basic Activity Selection**
    - Implement the standard greedy algorithm.
    - Verify output on small test cases where multiple intervals overlap.
2. **Weighted Interval Scheduling (Challenge)**
    - Extend the problem: each activity has a weight (profit).
    - Greedy fails here; use **Dynamic Programming** with binary search for compatibility.
3. **Room Allocation**
    - Given meeting intervals, allocate the minimum number of rooms.
    - Hint: Use a **min-heap** (`std::priority_queue` with `greater<>`) to track earliest finishing rooms.
4. **Randomized Testing**
    - Generate random intervals.
    - Compare greedy results against brute force (for small n) to experimentally confirm correctness.

19.3.4 Key Takeaways

- The **activity selection problem** is a canonical example of a greedy algorithm with a provable optimal solution.
- **Interval scheduling** generalizes this, reinforcing the power of sorting + greedy selection.
- Implementation in C++ leverages:
  - `std::sort` with custom comparators.
  - Efficient iteration patterns with minimal overhead.
- Extensions like **weighted interval scheduling** demonstrate the boundary where greedy breaks, motivating more advanced paradigms like **dynamic programming**.

Chapter 20

Randomized Algorithms & Probabilistic Methods

20.1 Random Number Generation in C++ (<random>), Reproducible Experiments, Seeds

Randomization plays a central role in algorithm design. Many algorithms rely on randomness to simplify implementation, reduce expected runtime, or provide probabilistic guarantees where deterministic approaches are either inefficient or overly complex. Examples include randomized quicksort, Monte Carlo methods, primality testing, and randomized data structures such as skip lists or treaps.
Modern C++ provides a robust and flexible framework for random number generation in the `<random>` header, replacing older and less reliable facilities such as `rand()` and `srand()`. The `<random>` library separates the concerns of randomness into three parts:

1. **Engines**: Deterministic algorithms that generate pseudo-random numbers based on an internal state. Examples include `std::mt19937` (Mersenne Twister) and

`std::linear_congruential_engine`.

2. **Distributions**: Mappings that transform uniform pseudo-random integers into numbers following specific probability distributions, such as uniform real, normal, Bernoulli, or exponential distributions.

3. **Seeds**: Initial values that determine the sequence of numbers generated by an engine, ensuring reproducibility when the same seed is used.

20.1.1 Engines: Generating Pseudo-Randomness

Engines are the core of `<random>`. A widely used engine is `std::mt19937`, a Mersenne Twister engine that provides high-quality randomness with a very long period (2^19937−1).
Example:

```
#include <iostream>
#include <random>

int main() {
    // Mersenne Twister engine
    std::mt19937 engine(42); // seed = 42

    // Generate raw numbers from the engine
    for (int i = 0; i < 5; ++i) {
        std::cout << engine() << "\n";
    }
}
```

The same program will always produce the same sequence of numbers because the engine is seeded deterministically with `42`.

20.1.2 Distributions: Mapping Randomness

Raw engine outputs are uniformly distributed integers. To simulate randomness in more meaningful ways, we apply distributions:

- `std::uniform_int_distribution<int>` → random integers in a given range.
- `std::uniform_real_distribution<double>` → random floating-point numbers in [a, b).
- `std::normal_distribution<double>` → numbers following a Gaussian distribution.
- `std::bernoulli_distribution` → coin-flip probabilities.

Example:

```
#include <iostream>
#include <random>

int main() {
    std::mt19937 engine(123);  // reproducible seed
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    for (int i = 0; i < 5; ++i) {
        std::cout << dist(engine) << "\n";
    }
}
```

Here, the distribution maps raw engine outputs into real numbers between 0.0 and 1.0.

20.1.3 Seeds: Ensuring Reproducibility

Randomized algorithms often require **reproducibility** to make debugging and experimental validation possible. Using a fixed seed ensures deterministic behavior across runs.

- **Fixed seed**: Ensures reproducible results.
- **Random device seed**: Provides non-deterministic seeding using `std::random_device`, if available.

Example:

```
std::random_device rd;   // may use hardware entropy source
std::mt19937 engine(rd()); // non-deterministic seed
```

This approach makes experiments less reproducible but closer to true randomness. For algorithm research, it is often useful to fix the seed during testing and switch to non-deterministic seeding in production.

20.1.4 Idiomatic Patterns in C++

A common idiom in randomized algorithm design is to encapsulate engines and distributions into utility functions:

```
#include <random>

inline double random_double(double a, double b) {
    static thread_local std::mt19937 engine(std::random_device{}());
    std::uniform_real_distribution<double> dist(a, b);
    return dist(engine);
}
```

- `static thread_local` ensures each thread has its own engine, avoiding contention.
- The function returns a reproducible, uniformly distributed floating-point number within `[a, b]`.

20.1.5 Reproducible Experiments in Algorithm Design

When evaluating randomized algorithms (e.g., testing randomized quicksort's expected complexity), reproducibility is crucial. Best practice includes:

1. Fixing the seed during benchmarking and recording it.
2. Reporting algorithm performance over multiple independent seeds to avoid bias.
3. Using controlled environments where distributions are explicitly defined.

20.1.6 Summary

- `<random>` provides modern facilities for robust random number generation.
- Engines produce deterministic sequences of numbers.
- Distributions transform raw outputs into useful probabilistic patterns.
- Seeds determine reproducibility, enabling experiments to be repeatable.
- Idiomatic C++ practices such as thread-local engines improve robustness in concurrent environments.

Random number generation is not just about simulating "randomness" but about carefully controlling and understanding probabilistic behavior in algorithms. Mastery of `<random>` is essential for implementing efficient randomized algorithms in C++.

20.2 QuickSelect, Hashing with Randomness, Monte Carlo Estimators

Randomized algorithms leverage **probability** to simplify solutions, improve expected performance, or provide approximate answers where deterministic methods are slow or complex. This section explores three key applications in modern algorithm design: **QuickSelect**, **hashing with randomness**, and **Monte Carlo estimators**. Each illustrates the balance between **efficiency** and **controlled randomness** in C++.

20.2.1 QuickSelect: Randomized Selection

- **Problem**

  Given an array of `n` elements, find the k-th smallest element efficiently. Deterministic selection algorithms exist but are often more complex and slower in practice.

- **Randomized Algorithm**

  - QuickSelect is a variant of QuickSort that recursively partitions the array but only processes the partition containing the k-th element.
  - Randomization improves expected performance by **choosing a pivot uniformly at random**, reducing the probability of worst-case partitions.

- **Algorithm Steps**

  1. Select a pivot randomly from the current subarray.
  2. Partition the array around the pivot: elements less than pivot to the left, greater to the right.

3. Recur into the side containing the k-th element.

- **C++ Implementation**

```cpp
#include <bits/stdc++.h>
using namespace std;

int quickSelect(vector<int>& arr, int left, int right, int k) {
    mt19937 rng(random_device{}());  // random pivot generator
    if (left == right) return arr[left];

    uniform_int_distribution<int> dist(left, right);
    int pivotIndex = dist(rng);
    swap(arr[pivotIndex], arr[right]);

    int pivot = arr[right];
    int storeIndex = left;
    for (int i = left; i < right; ++i) {
        if (arr[i] < pivot) swap(arr[i], arr[storeIndex++]);
    }
    swap(arr[storeIndex], arr[right]);

    if (k == storeIndex) return arr[k];
    else if (k < storeIndex) return quickSelect(arr, left, storeIndex - 1, k);
    else return quickSelect(arr, storeIndex + 1, right, k);
}

int main() {
    vector<int> data = {9, 2, 6, 3, 1, 8, 5};
    int k = 3; // 0-based index
    cout << "3rd smallest element: "
         << quickSelect(data, 0, data.size() - 1, k) << endl;
}
```

Complexity:

- Expected: `O(n)`
- Worst-case: `O(n^2)`, extremely unlikely with random pivot selection.

20.2.2 Hashing with Randomness

Hash tables are central to many algorithms. Deterministic hashing can lead to **clustering and collisions** under adversarial inputs. Randomized hashing techniques address this:

- **Universal Hashing**
  - Use a randomly chosen hash function from a **universal family** to map keys.
  - Guarantees **low expected collision probability** regardless of input distribution.
- **Example: Randomized Modulo Hashing**

```cpp
#include <bits/stdc++.h>
using namespace std;

struct RandomHash {
    size_t operator()(uint64_t x) const {
        static mt19937_64 rng(random_device{}());
        static uniform_int_distribution<uint64_t> dist;
        uint64_t randomSeed = dist(rng) | 1; // odd number
        x ^= x >> 33;
        x *= randomSeed;
        x ^= x >> 33;
        x *= randomSeed;
```

```
        x ^= x >> 33;
        return x;
    }
};

int main() {
    unordered_map<uint64_t, string, RandomHash> table;
    table[42] = "Answer";
    cout << table[42] << endl;
}
```

Benefits:

- Robust against input patterns.
- Reduces worst-case collisions in competitive programming or adversarial applications.

20.2.3 Monte Carlo Estimators

Monte Carlo methods estimate **numerical quantities using randomness**, often where exact computation is expensive or infeasible.

- **Principle**
  - Randomly sample input or scenarios.
  - Use the **law of large numbers** to approximate expected values.
  - Accuracy improves as the number of samples increases.
- **Example: Estimating**

```cpp
#include <bits/stdc++.h>
using namespace std;

double estimatePi(int samples) {
    mt19937 rng(random_device{}());
    uniform_real_distribution<double> dist(0.0, 1.0);
    int inside = 0;

    for (int i = 0; i < samples; ++i) {
        double x = dist(rng), y = dist(rng);
        if (x*x + y*y <= 1.0) ++inside;
    }
    return 4.0 * inside / samples;
}

int main() {
    cout << "Estimated Pi: " << estimatePi(1'000'000) << endl;
}
```

Complexity and Accuracy:

- Simple to implement; parallelizable easily.
- Error decreases as `O(1/√n)`, where `n` is the number of samples.

20.2.4 Idiomatic C++ Considerations

- Use `<random>` for **reproducible experiments** via fixed seeds.
- Encapsulate engines and distributions for **thread safety** in parallel computations (`thread_local` engines).
- Combine randomization with standard algorithms:

- `std::shuffle` for randomized quicksort or permutation testing.
- `std::sample` for Monte Carlo subsampling.

20.2.5 Key Takeaways

1. **QuickSelect** shows how random pivot selection improves expected performance in selection problems.
2. **Randomized hashing** ensures robust performance under unpredictable or adversarial inputs.
3. **Monte Carlo estimators** demonstrate practical approximate solutions using probabilistic sampling.
4. Modern C++ `<random>` facilities allow **safe, reproducible, and high-quality randomness**, essential for both experimental evaluation and real-world algorithm deployment.

These exercises bridge **theoretical probabilistic methods** with **practical C++ implementations**, preparing the reader for randomized algorithms in optimization, data structures, and approximate computations.

20.3 Exercises: Randomized Algorithms for Median, Bloom Filter Sketch

Randomized algorithms are not only theoretically elegant but also **highly practical** for handling large datasets and streaming data where deterministic approaches may be inefficient. This section provides exercises that illustrate both **exact randomized selection** and **probabilistic data structures**, connecting theory with modern C++ implementations.

20.3.1 Randomized Median Selection

- **Problem**

  Given an unsorted array of `n` elements, find the median (or k-th smallest element) efficiently using randomness.

- **Randomized Approach**

  - Implement **QuickSelect** (discussed in Section 2) with **random pivot selection**.
  - Randomized pivoting ensures **expected linear time**.

- **Exercise Goals**

  - Implement randomized median selection using `std::mt19937` and `std::uniform_int_distribution`.
  - Verify correctness across multiple randomized seeds.
  - Compare runtime against a deterministic median algorithm for large arrays.

- **C++ Template**

```cpp
#include <bits/stdc++.h>
using namespace std;

double estimatePi(int samples) {
    mt19937 rng(random_device{}());
    uniform_real_distribution<double> dist(0.0, 1.0);
    int inside = 0;

    for (int i = 0; i < samples; ++i) {
        double x = dist(rng), y = dist(rng);
        if (x*x + y*y <= 1.0) ++inside;
    }
    return 4.0 * inside / samples;
}

int main() {
    cout << "Estimated Pi: " << estimatePi(1'000'000) << endl;
}
```

Exercise Extension:

- Test on large vectors (10^6 elements).
- Measure average runtime over multiple seeds to observe expected linear behavior.

20.3.2 Bloom Filter Sketch

- **Concept**

  A **Bloom filter** is a **probabilistic data structure** for set membership testing:

- Supports **insert** and **query** operations.
- May yield **false positives** (element appears to be present when it is not) but **never false negatives**.
- Space-efficient: uses a bit array and multiple hash functions.

Randomization enters in **hash function selection** and **collision handling**. Bloom filters are widely used in:

- Network routers for packet filtering.
- Databases for approximate membership queries.
- Large-scale caching systems.

- **Exercise Goals**
  - Implement a Bloom filter using **randomized hash functions**.
  - Use `<random>` for hash seeds and simulate **k hash functions**.
  - Test the false positive rate empirically.

- **C++ Template Implementation**

```
#include <bits/stdc++.h>
using namespace std;

class BloomFilter {
    vector<bool> bits;
    int size, numHashes;
    mt19937 rng;

    vector<size_t> hashIndices(const string& s) {
```

```
        vector<size_t> indices;
        hash<string> hasher;
        for (int i = 0; i < numHashes; ++i) {
            size_t h = hasher(s) ^ (rng() + i*0x9e3779b9);
            indices.push_back(h % size);
        }
        return indices;
    }

public:
    BloomFilter(int nBits, int kHashes)
        : bits(nBits, false), size(nBits), numHashes(kHashes),
        ↪  rng(random_device{}()) {}

    void insert(const string& s) {
        for (auto idx : hashIndices(s)) bits[idx] = true;
    }

    bool contains(const string& s) {
        for (auto idx : hashIndices(s))
            if (!bits[idx]) return false;
        return true;
    }
};

int main() {
    BloomFilter bf(1000, 3);
    bf.insert("apple");
    bf.insert("banana");

    cout << bf.contains("apple") << endl;  // true
    cout << bf.contains("grape") << endl;  // probably false
```

```
}
```

Exercise Extension:

- Experiment with different bit array sizes and number of hash functions.
- Compute empirical **false positive probability** by querying non-inserted elements.
- Compare performance against `std::unordered_set` for memory efficiency.

20.3.3 Learning Objectives

These exercises aim to:

1. Reinforce **randomized selection algorithms** with QuickSelect.
2. Illustrate **probabilistic data structures** (Bloom filters) for large-scale approximate queries.
3. Show practical C++ usage of:
   - `<random>` for reproducibility and hash function variability.
   - Templates and functional programming patterns for reusable code.
4. Demonstrate the trade-off between **accuracy** and **space/time efficiency** in randomized algorithms.

20.3.4 Suggested Practice

- Extend Bloom filters to support **deletion** using **counting Bloom filters**.
- Combine randomized QuickSelect with Bloom filter sketches to **efficiently estimate medians in streaming data**.
- Benchmark randomized vs deterministic approaches to understand expected vs worst-case behaviors.

Chapter 21

Approximation Algorithms & NP-Hard Problems

21.1 Common Approximation Strategies Implemented in C++

Many combinatorial problems are **NP-hard**, meaning that no polynomial-time algorithm is known for computing exact solutions. Approximation algorithms provide **efficient, near-optimal solutions** with **provable guarantees**. Modern C++ offers tools to implement these strategies efficiently, leveraging **templates, STL containers, and functional programming idioms**.

This section covers common strategies and demonstrates how they can be **implemented idiomatically in C++**.

21.1.1 Greedy Approximation

- **Concept**
  - Many NP-hard problems, like **vertex cover** or **set cover**, can be approximated using greedy heuristics.
  - Greedy strategies make a **locally optimal choice** at each step with provable approximation bounds.
- **Example: Vertex Cover (2-Approximation)**
  - **Problem:** Given a graph `G(V, E)`, select a subset of vertices such that every edge has at least one endpoint in the subset.
  - **Greedy Approach:**
    1. Pick any uncovered edge `(u, v)`.
    2. Add both `u` and `v` to the vertex cover.
    3. Remove all edges incident to `u` or `v`.
    4. Repeat until all edges are covered.

C++ Implementation

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> vertexCoverApprox(int n, vector<pair<int,int>>& edges) {
    vector<bool> covered(n, false);
    vector<int> cover;

    for (auto& [u, v] : edges) {
        if (!covered[u] && !covered[v]) {
```

```
            cover.push_back(u);
            cover.push_back(v);
            covered[u] = covered[v] = true;
        }
    }
    return cover;
}

int main() {
    int n = 5;
    vector<pair<int,int>> edges = { {0,1}, {1,2}, {2,3}, {3,4}, {0,4} };
    auto cover = vertexCoverApprox(n, edges);
    for (int v : cover) cout << v << " ";
}
```

  - **Approximation guarantee:** The size of the computed cover is at most **2× the optimal**.

21.1.2 Linear Programming Relaxation

- **Concept**
  - Many NP-hard problems can be formulated as **integer linear programs (ILPs)**.
  - **Relaxing integrality constraints** to allow fractional values gives a **polynomial-time solvable linear program (LP)**.
  - Rounded solutions provide **approximation bounds**.
- **C++ Implementation Notes**

- Use libraries such as **Eigen**, **GLPK**, or **COIN-OR** for LP solving.
- Example: Fractional vertex cover solution can be rounded by including vertices with fractional values 0.5.

21.1.3 Randomized Rounding

- **Concept**
  - Apply **randomized choices** to fractional solutions of LPs or probability-based heuristics.
  - Guarantees hold **in expectation**, making it a powerful tool in approximation algorithm design.
- **Example: Max-Cut Approximation**
  - Solve a relaxed LP or SDP.
  - Assign vertices to partitions randomly based on fractional values.
  - Expected cut size is at least **0.878 × optimal** (Goemans–Williamson algorithm).

  C++ Implementation Note:
  - Use `<random>` engines to sample from probability distributions.
  - Thread-local engines for parallelized randomized rounding.

21.1.4 Local Search Heuristics

- **Concept**
  - Start with an initial feasible solution.

  - Iteratively **improve by local modifications** until no better solution exists.
  - Useful for problems like **k-median**, **facility location**, and **traveling salesman problem (TSP)**.

- **C++ Implementation Tips**
  - Represent solutions with `std::vector<int>` or `std::bitset`.
  - Use `std::shuffle` for random perturbations.
  - Maintain a **priority queue** or **set** for evaluating neighboring solutions efficiently.

21.1.5 PTAS / FPTAS Approaches

- **PTAS (Polynomial-Time Approximation Scheme):**
  - Produces `(1+ )`-approximate solutions in `O(n^(f(1/ )))` time.
- **FPTAS (Fully Polynomial-Time Approximation Scheme):**
  - Produces `(1+ )`-approximate solutions in `O(poly(n,1/ ))` time.
- Implementation often involves **dynamic programming with scaled or rounded values**.

C++ Implementation Example: Knapsack FPTAS

- Scale item values by and round down to reduce state space.
- Standard DP computes approximate maximum efficiently.

```
// See Section 2 of Chapter 2 for DP templates on sequences and knapsack
```

21.1.6 Idiomatic C++ Patterns for Approximation

1. **Templates and generic programming:** reusable solution for various problem sizes.
2. **STL containers:**
   - `std::vector`, `std::set`, `std::unordered_map` for efficient state representation.
   - `std::priority_queue` for greedy choices.
3. **Random number generators:** `<random>` for randomized algorithms.
4. **Functional constructs:** `std::transform`, `std::accumulate` for concise state updates.

21.1.7 Exercises

1. **Vertex Cover:** Implement the 2-approximation and compare with exact ILP solution for small graphs.
2. **Set Cover:** Implement greedy set cover approximation; measure solution size against optimal small instances.
3. **Randomized Max-Cut:** Implement Goemans–Williamson-style randomized rounding; validate expected cut size.
4. **Knapsack FPTAS:** Apply dynamic programming with scaling to produce `(1+ )`-approximate solution.

5. **Local Search TSP:** Implement 2-opt or 3-opt improvement heuristics and measure runtime vs quality.

21.1.8 Key Takeaways

- Approximation algorithms provide **practical solutions for NP-hard problems**.
- C++ offers modern facilities—**templates, STL, `<random>`**, and libraries for LP/SDP—to implement these methods efficiently.
- Mastery of greedy, LP relaxation, randomized rounding, and local search strategies is essential for **efficient, provable algorithmic design**.

21.2 Local Search, Greedy Approximation, PTAS Examples Where Applicable

Approximation algorithms provide efficient solutions to NP-hard problems when exact algorithms are infeasible. This section explores **local search heuristics**, **greedy approximations**, and **polynomial-time approximation schemes (PTAS)**, demonstrating practical implementations and idiomatic C++ techniques for each strategy.

21.2.1 Local Search Heuristics

- **Concept**

  Local search iteratively improves a feasible solution by exploring **neighboring solutions**. The algorithm continues until no better solution exists in the neighborhood.

  - Useful for problems such as:
    - **Traveling Salesman Problem (TSP)**
    - **k-Median and Facility Location**
    - **Graph partitioning**

- **Algorithm Pattern**

  1. Start with an initial solution (random or greedy).
  2. Define a **neighborhood function** that produces small modifications.
  3. Move to the neighbor if it improves the objective.
  4. Repeat until a **local optimum** is reached.

- **C++ Implementation Example: 2-Opt TSP**

```cpp
#include <bits/stdc++.h>
using namespace std;

double distance(pair<int,int> a, pair<int,int> b) {
    return hypot(a.first-b.first, a.second-b.second);
}

double totalLength(const vector<pair<int,int>>& tour) {
    double sum = 0.0;
    for (size_t i = 0; i < tour.size(); ++i)
        sum += distance(tour[i], tour[(i+1) % tour.size()]);
    return sum;
}

void twoOptSwap(vector<pair<int,int>>& tour, int i, int k) {
    reverse(tour.begin() + i, tour.begin() + k + 1);
}

void localSearchTSP(vector<pair<int,int>>& tour) {
    bool improved = true;
    while (improved) {
        improved = false;
        for (size_t i = 1; i < tour.size() - 1; ++i) {
            for (size_t k = i+1; k < tour.size(); ++k) {
                auto oldLength = totalLength(tour);
                twoOptSwap(tour, i, k);
                if (totalLength(tour) < oldLength) {
                    improved = true;
                } else {
                    twoOptSwap(tour, i, k); // revert
                }
```

```
            }
        }
    }
}
```

- **Notes:**
  - Can combine with **randomized restarts** for better global search.
  - Using `std::shuffle` or `std::random_device` helps generate diverse starting solutions.

21.2.2 Greedy Approximation

- **Concept**

  Greedy algorithms iteratively make the **locally optimal choice**.

  - Efficient and often produces provable **approximation bounds**.
  - Typical problems: **Vertex Cover**, **Set Cover**, **Interval Scheduling**, **Weighted Matching**.

- **Example: Interval Scheduling**

  - **Problem:** Select maximum number of non-overlapping intervals.
  - **Greedy Approach:**
    1. Sort intervals by finish time.
    2. Select the first interval that starts after the last selected interval.

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Interval { int start, end; };
bool cmp(const Interval& a, const Interval& b) { return a.end < b.end; }

int maxNonOverlappingIntervals(vector<Interval>& intervals) {
    sort(intervals.begin(), intervals.end(), cmp);
    int count = 0, lastEnd = INT_MIN;
    for (auto& iv : intervals) {
        if (iv.start >= lastEnd) {
            lastEnd = iv.end;
            ++count;
        }
    }
    return count;
}

int main() {
    vector<Interval> intervals = {{1,3},{2,5},{4,7},{6,8}};
    cout << "Max non-overlapping: " << maxNonOverlappingIntervals(intervals) <<
    ↪  endl;
}
```

- **Approximation Guarantees:**
  - For many NP-hard problems like **Vertex Cover**, greedy achieves factor-2 approximation.

21.2.3 Polynomial-Time Approximation Schemes (PTAS)

- **Concept**

  - PTAS produces solutions arbitrarily close to optimal: `(1+ )` factor.
  - Runtime may depend polynomially on input size, with degree depending on `1/ `.
  - Useful for:
    * **Knapsack Problem**
    * **Euclidean TSP (via Arora's Algorithm)**
    * **Packing and Scheduling Problems**

- **Example: Knapsack PTAS**
  - **Idea:** Scale item values to reduce DP table size.
  - **C++ Implementation Notes:**
    * Let `v_max` be maximum item value, desired accuracy.
    * Scale values: `v_i' = floor(v_i * n / (  * v_max))`
    * Solve scaled DP to get approximate solution.

```
// Template DP for knapsack is already discussed in Chapter 2, Section 3
// For PTAS, scale values and use same DP approach
```

  - **Trade-off:** Accuracy vs runtime: higher → faster computation, less accuracy.

21.2.4 Idiomatic C++ Patterns

- **Templates and STL:** Generic containers (`vector`, `bitset`, `unordered_map`) allow flexible solution representation.

- **Randomized restarts:** `<random>` + `std::shuffle` for diverse initial solutions in local search.
- **Functional utilities:** `std::accumulate`, `std::transform` simplify objective calculations.
- **Priority queues:** `std::priority_queue` efficiently supports greedy selections.

21.2.5 Exercises

1. Implement **local search for k-median problem** using Euclidean distance.
2. Solve **weighted interval scheduling** using greedy approximation.
3. Implement **Knapsack PTAS** and experiment with different values, comparing approximate and exact solutions.
4. Extend TSP 2-opt local search with **randomized restarts** and evaluate solution quality vs iterations.

21.2.6 Key Takeaways

- **Local search** provides flexible heuristics for NP-hard problems.
- **Greedy algorithms** are simple, efficient, and often provably approximate.
- **PTAS** allows arbitrary closeness to optimal with controllable trade-offs.
- Modern C++ idioms enable clean, reusable, and high-performance implementations of approximation algorithms.

These strategies form a **practical toolkit** for solving NP-hard problems efficiently while maintaining provable solution quality, bridging theoretical computer science with real-world algorithm engineering.

21.3 Exercises: Vertex Cover Approximation, Traveling Salesman Heuristics

This section provides hands-on exercises to implement **approximation strategies** for classic NP-hard problems. The focus is on **vertex cover** and **Traveling Salesman Problem (TSP)**, illustrating **greedy, local search, and heuristic-based approaches** in modern C++.

21.3.1 Vertex Cover Approximation Exercises

- **Problem Statement**
  - Given a graph $G = (V, E)$, find a subset $C \subseteq V$ such that each edge has at least one endpoint in C.
  - Exact solution is NP-hard; approximation algorithms are efficient and provide guarantees.
- **Exercise 1.1: Greedy 2-Approximation**

  Instructions:

  1. Implement the **greedy 2-approximation algorithm**:
     - While edges remain uncovered:
       * Pick any uncovered edge (u,v).
       * Add both u and v to the cover.
       * Remove all edges incident to u or v.
  2. Use **STL containers** (`vector`, `unordered_set`) for representation.
  3. Verify the solution satisfies all edges.

4. Compare the size of the cover with the number of vertices; it should be at most twice the optimal in small test cases.

```
#include <bits/stdc++.h>
using namespace std;

vector<int> vertexCoverApprox(int n, vector<pair<int,int>>& edges) {
    vector<bool> covered(n, false);
    vector<int> cover;

    for (auto& [u, v] : edges) {
        if (!covered[u] && !covered[v]) {
            cover.push_back(u);
            cover.push_back(v);
            covered[u] = covered[v] = true;
        }
    }
    return cover;
}

int main() {
    int n = 6;
    vector<pair<int,int>> edges = {{0,1},{1,2},{2,3},{3,4},{4,5},{0,5}};
    auto cover = vertexCoverApprox(n, edges);
    cout << "Approximate Vertex Cover: ";
    for (int v : cover) cout << v << " ";
}
```

Extension Exercises:

– Implement a **randomized vertex cover** by selecting edges randomly each iteration.

- Compare average cover size over multiple random runs.
- Explore **local search improvements**: try removing vertices from the cover and check if edges remain covered.

21.3.2 Traveling Salesman Problem (TSP) Heuristics

- **Problem Statement**
  - Given a set of cities and distances, find the shortest tour visiting all cities exactly once.
  - Exact solution is NP-hard; heuristics produce **good-enough solutions** efficiently.
- **Exercise 2.1: Nearest Neighbor Heuristic**

  Instructions:

  1. Start from any city.
  2. Iteratively move to the nearest unvisited city.
  3. Repeat until all cities are visited, then return to the starting city.

```
#include <bits/stdc++.h>
using namespace std;

double distance(pair<int,int> a, pair<int,int> b) {
    return hypot(a.first-b.first, a.second-b.second);
}

vector<int> nearestNeighborTSP(const vector<pair<int,int>>& cities) {
    int n = cities.size();
```

```
    vector<int> tour;
    vector<bool> visited(n, false);

    int current = 0;  // starting city
    tour.push_back(current);
    visited[current] = true;

    for (int i = 1; i < n; ++i) {
        double bestDist = numeric_limits<double>::max();
        int nextCity = -1;
        for (int j = 0; j < n; ++j) {
            if (!visited[j]) {
                double d = distance(cities[current], cities[j]);
                if (d < bestDist) {
                    bestDist = d;
                    nextCity = j;
                }
            }
        }
        current = nextCity;
        tour.push_back(current);
        visited[current] = true;
    }
    return tour;
}
```

- **Exercise 2.2: 2-Opt Local Search**

  Instructions:

  1. Start with an initial tour (e.g., nearest neighbor solution).
  2. Iteratively swap two edges (2-opt) to reduce total distance.

3. Repeat until no improvement occurs.

```
void twoOptSwap(vector<int>& tour, int i, int k) {
    reverse(tour.begin() + i, tour.begin() + k + 1);
}

void twoOptLocalSearch(vector<int>& tour, const vector<pair<int,int>>& cities)
↪  {
    bool improved = true;
    auto tourLength = [&](const vector<int>& t) {
        double sum = 0;
        for (size_t i = 0; i < t.size(); ++i)
            sum += distance(cities[t[i]], cities[t[(i+1)%t.size()]]);
        return sum;
    };

    while (improved) {
        improved = false;
        for (size_t i = 1; i < tour.size() - 1; ++i) {
            for (size_t k = i+1; k < tour.size(); ++k) {
                auto oldLen = tourLength(tour);
                twoOptSwap(tour, i, k);
                if (tourLength(tour) < oldLen)
                    improved = true;
                else
                    twoOptSwap(tour, i, k);  // revert
            }
        }
    }
}
```

Extension Exercises:

- Compare nearest neighbor tour vs 2-opt improved tour in terms of total distance.
- Implement **randomized restarts** to escape local minima.
- For **Euclidean TSP**, experiment with PTAS-style heuristics (like Arora's method) for small instances.

21.3.3 Learning Objectives

1. Apply **greedy and local search heuristics** to NP-hard problems.
2. Implement **randomized and deterministic approximation algorithms** efficiently in modern C++.
3. Understand the **trade-off between runtime and solution quality**.
4. Explore **improvements through local optimization and randomized restarts**.

21.3.4 Suggested Practice

- Extend vertex cover approximation to **weighted vertex cover**.
- Integrate **Bloom filter or hash-based sampling** to speed up large graph heuristics.
- Analyze **empirical approximation ratio** vs optimal (on small test instances).
- Combine **2-opt local search with randomized nearest neighbor starts** for TSP optimization.

Part VI

Performance, Concurrency & Low-level Concerns (C++ focused)

Chapter 22

Memory & Cache-aware Algorithm Design

22.1 Data Layout, Locality, and Structure-of-Arrays vs Array-of-Structures

Efficient memory usage and **cache-aware programming** are critical in high-performance C++ applications. Modern CPUs rely heavily on **cache hierarchies** (L1, L2, L3) to reduce memory latency, making **data layout** a key factor in performance optimization. This section discusses **array-of-structures (AoS)** versus **structure-of-arrays (SoA)** layouts, their impact on **data locality**, and practical C++ implementation strategies.

22.1.1 Memory Locality and Cache Basics

- **Spatial locality:** Accessing contiguous memory locations improves cache line utilization.
- **Temporal locality:** Reusing recently accessed data avoids cache misses.
- **Cache line size:** Typically 64 bytes in modern x86 and ARM CPUs. Aligning frequently accessed data to cache lines improves performance.

Example: Iterating over a large array of structures may result in cache misses if fields not needed are loaded unnecessarily.

22.1.2 Array-of-Structures (AoS)

- **Concept**
  - Data is stored as a **sequence of complete structures**, each containing multiple fields.
  - Typical in object-oriented designs.
- **Example in C++**

```
struct Particle {
    float x, y, z;   // position
    float vx, vy, vz; // velocity
};

std::vector<Particle> particles(1000000);
```

- **Characteristics**

  - **Pros:**
    * Intuitive and natural for OOP.
    * Easy to pass around as objects.
  - **Cons:**
    * Accessing only `x` coordinates requires loading **all fields**, causing cache inefficiency.
    * Poor spatial locality for field-specific operations (e.g., SIMD vectorization).

22.1.3 Structure-of-Arrays (SoA)

- **Concept**
  - Data is stored as separate **arrays for each field**, improving access to a single attribute across all objects.
- **Example in C++**

```
struct ParticlesSoA {
    std::vector<float> x, y, z;
    std::vector<float> vx, vy, vz;

    ParticlesSoA(size_t n)
        : x(n), y(n), z(n), vx(n), vy(n), vz(n) {}
};
```

- **Advantages**
  1. **Better cache performance**: Iterating over one attribute touches contiguous memory only.

2. **Vectorization-friendly**: SIMD instructions can operate on continuous arrays efficiently.
3. **Reduced cache pollution**: Unused fields are not loaded into cache.

- **Example Usage**

```
void updatePositions(ParticlesSoA& p, float dt, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        p.x[i] += p.vx[i] * dt;
        p.y[i] += p.vy[i] * dt;
        p.z[i] += p.vz[i] * dt;
    }
}
```

22.1.4 Performance Implications

Feature	AoS	SoA
Memory access for one field	Poor (cache lines contain unused fields)	Excellent (contiguous memory)
SIMD/vectorization	Difficult	Easy
Ease of use	Natural for objects	Requires separate arrays
Insertion/removal	Easy	More complex

Rule of thumb:

- Use AoS when operations require **all fields together**.

- Use SoA when operations **focus on a subset of fields**, especially for large-scale numeric computations or simulations.

22.1.5 Hybrid Approaches

- **Array-of-Structures-of-Arrays (AoSoA):**
  - Combines AoS and SoA advantages by **blocking structures** into small arrays to maintain cache-friendly access and object-oriented abstraction.
- Example: Particle blocks of size 32 stored contiguously.

```
struct ParticleBlock {
    float x[32], y[32], z[32];
    float vx[32], vy[32], vz[32];
};
std::vector<ParticleBlock> blocks;
```

- Provides **vectorization opportunities** and **object-like access**.

22.1.6 C++ Techniques for Cache Awareness

1. **Align data:** Use `alignas(64)` or `std::aligned_alloc` for cache-line alignment.
2. **Prefetching:** Use compiler intrinsics (`__builtin_prefetch`) for predictable memory access patterns.
3. **SIMD-friendly structures:** SoA layout allows use of **`std::simd`** or compiler intrinsics (`_mm256_load_ps`).

4. **Memory pools:** Avoid dynamic allocations per object; allocate in **large contiguous blocks**.

5. **Avoid false sharing:** Ensure frequently modified fields are not shared across threads on the same cache line.

22.1.7 Exercises

1. Implement AoS and SoA particle systems for `1,000,000` particles; measure runtime of **position updates**.

2. Vectorize SoA position updates using **SIMD intrinsics** and compare performance with AoS.

3. Experiment with AoSoA layout for **cache-blocked particle processing**; measure cache miss reduction using hardware performance counters.

4. Evaluate memory usage and runtime trade-offs for different data layouts.

22.1.8 Key Takeaways

- **Data layout directly affects cache performance** and overall runtime in C++ programs.
- **AoS is intuitive** but can be inefficient for field-specific operations.
- **SoA improves locality, cache efficiency, and vectorization**, especially for numeric or simulation workloads.
- **Hybrid layouts (AoSoA)** allow a balance between object-oriented programming and high-performance computing.

- Modern C++ tools (`std::vector`, `alignas`, SIMD abstractions) provide mechanisms to implement **cache-aware algorithms efficiently**.

22.2 Algorithms Optimized for Cache (Blocking, Tiling) with C++ Examples

Efficient utilization of CPU caches is critical for high-performance C++ programs, particularly in **numerical computation**, **matrix operations**, and **data-intensive algorithms**. This section explores **cache-aware algorithm design techniques** such as **blocking** and **tiling**, demonstrating practical C++ implementations.

22.2.1 Cache Optimization Principles

- **CPU caches** are small, fast memory stores between main memory (RAM) and the processor.
- **Cache miss penalty** can dominate runtime in large datasets.
- Optimizing memory access patterns can drastically improve performance.
- **Temporal locality:** Reuse data already in cache.
- **Spatial locality:** Access contiguous memory regions sequentially.

22.2.2 Blocking / Tiling Technique

Concept

- Divide large datasets into **small blocks (tiles)** that fit in cache.
- Operate on blocks completely before moving to the next.
- Reduces **cache evictions** and **reloading of data**.

Applications:

- Matrix multiplication
- Convolution in image processing
- Large array operations

22.2.3 Example: Matrix Multiplication

- **Naive Implementation (Poor Cache Usage)**

```
#include <vector>
using namespace std;

void matMulNaive(const vector<vector<double>>& A,
                 const vector<vector<double>>& B,
                 vector<vector<double>>& C,
                 int n) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                C[i][j] += A[i][k] * B[k][j];
}
```

  Problems:

  – Accessing `B[k][j]` repeatedly causes **cache misses** due to column-major access (in row-major storage).

- **Cache-Blocked / Tiled Implementation**

```
void matMulBlocked(const vector<vector<double>>& A,
                   const vector<vector<double>>& B,
                   vector<vector<double>>& C,
                   int n, int blockSize) {
    for (int ii = 0; ii < n; ii += blockSize)
        for (int jj = 0; jj < n; jj += blockSize)
            for (int kk = 0; kk < n; kk += blockSize)
                for (int i = ii; i < min(ii+blockSize, n); ++i)
                    for (int j = jj; j < min(jj+blockSize, n); ++j) {
                        double sum = 0.0;
                        for (int k = kk; k < min(kk+blockSize, n); ++k)
                            sum += A[i][k] * B[k][j];
                        C[i][j] += sum;
                    }
}
```

Notes:

- `blockSize` is tuned to **fit cache line sizes** (typically 64–256 elements).
- Dramatically improves **cache hit ratio** for large matrices.
- Enables compiler **vectorization** within blocks.

22.2.4 Tiling for Multi-Dimensional Arrays

Concept

- Generalization of blocking for **2D/3D arrays**.
- Access small sub-blocks sequentially in **nested loops** to maximize cache reuse.

```
void tiled2DOperation(vector<vector<double>>& data, int n, int tileSize) {
    for (int i0 = 0; i0 < n; i0 += tileSize)
        for (int j0 = 0; j0 < n; j0 += tileSize)
            for (int i = i0; i < min(i0+tileSize, n); ++i)
                for (int j = j0; j < min(j0+tileSize, n); ++j)
                    data[i][j] = data[i][j] * 2.0;  // Example operation
}
```

- Each tile fits in cache → **minimal cache misses**.

22.2.5 C++ Idiomatic Patterns

1. **Use contiguous storage**: `std::vector` or `std::array` for 1D layout of matrices for better spatial locality.

```
std::vector<double> mat(n * n);
#define IDX(i,j) ((i)*n + (j))
mat[IDX(i,j)] = ...
```

1. **SIMD-friendly layout**: Structure-of-Arrays (SoA) for multi-field data.
2. **`std::min` in loop bounds** ensures no out-of-range access.
3. **Compile-time block size tuning**: Use `constexpr int blockSize = 64;` for performance portability.
4. **Parallelization**: Combine blocking with `std::execution::par` for multi-threaded loops.

22.2.6 Exercises

1. Implement **cache-blocked matrix multiplication** for various block sizes; compare runtime against naive implementation.

2. Optimize **3D array convolution** using tiling; measure cache misses using hardware counters.

3. Experiment with **row-major vs column-major storage** for tiled operations; analyze performance differences.

4. Implement **parallel blocked matrix multiplication** using `std::thread` or `std::execution::par` and analyze speedup.

22.2.7 Key Takeaways

- **Blocking and tiling** reduce cache misses and improve memory access efficiency.
- Performance gains are most significant for **large datasets** that exceed cache size.
- Modern C++ features (`std::vector`, `std::array`, `std::execution`) enable **cache-aware algorithm design** with readable and maintainable code.
- Tuning **block size** for the target CPU cache is critical for achieving optimal performance.

Chapter 23

Parallel & Concurrent Algorithms

23.1 Threading Primitives in C++ (std::thread, Atomics, Mutexes) and Lock-Free Ideas

Modern C++ offers a rich set of concurrency primitives, enabling developers to write parallel and concurrent algorithms that can take full advantage of multi-core architectures. Understanding these primitives is essential for designing algorithms that balance correctness, performance, and scalability. In this section, we explore threading primitives, synchronization mechanisms, and the foundations of lock-free programming.

23.1.1 The Role of Threads in Modern C++

Threads represent the fundamental unit of execution that runs independently but shares the same address space with other threads in a process. C++11 introduced `std::thread`, providing a standardized abstraction over OS-level threads. By spawning multiple threads, algorithms can be parallelized to perform independent tasks concurrently, improving throughput on multicore systems.

Example: Creating and Joining Threads

```
#include <iostream>
#include <thread>

void worker(int id) {
    std::cout << "Worker " << id << " is running\n";
}

int main() {
    std::thread t1(worker, 1);
    std::thread t2(worker, 2);

    // Join ensures main waits for both threads to finish
    t1.join();
    t2.join();
}
```

Here, two threads execute the `worker` function concurrently, while `main` waits for completion with `join()`.

23.1.2 Synchronization Primitives

- **a) Mutexes**

  Mutual exclusion locks (`std::mutex`) protect critical sections where shared resources are accessed. They prevent race conditions but must be used carefully to avoid deadlocks.

  ```
  #include <iostream>
  #include <thread>
  #include <mutex>
  ```

```
int counter = 0;
std::mutex mtx;

void increment() {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(mtx); // RAII style
        ++counter;
    }
}

int main() {
    std::thread t1(increment), t2(increment);
    t1.join();
    t2.join();

    std::cout << "Final counter: " << counter << "\n";
}
```

The `lock_guard` ensures the mutex is automatically released at scope exit.

- **b) Atomics**

  `std::atomic` provides lock-free, thread-safe operations on primitive data types when supported by hardware. They allow concurrent modification of shared variables without explicit mutexes.

```
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<int> counter{0};
```

```
void increment() {
    for (int i = 0; i < 1000; ++i) {
        counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    std::thread t1(increment), t2(increment);
    t1.join();
    t2.join();

    std::cout << "Final counter: " << counter.load() << "\n";
}
```

Atomics avoid the overhead of mutex locking when only basic operations are required.

- **c) Condition Variables**

  Condition variables (`std::condition_variable`) allow threads to wait for specific conditions, enabling synchronization patterns like producer–consumer.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>

std::queue<int> dataQueue;
std::mutex mtx;
std::condition_variable cv;
```

```
bool finished = false;

void producer() {
    for (int i = 0; i < 5; ++i) {
        {
            std::lock_guard<std::mutex> lock(mtx);
            dataQueue.push(i);
        }
        cv.notify_one();
    }
    finished = true;
    cv.notify_all();
}

void consumer() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [] { return !dataQueue.empty() || finished; });
        if (!dataQueue.empty()) {
            int val = dataQueue.front();
            dataQueue.pop();
            std::cout << "Consumed: " << val << "\n";
        } else if (finished) {
            break;
        }
    }
}
```

23.1.3 Lock-Free and Wait-Free Ideas

Lock-free algorithms aim to guarantee progress for at least one thread without requiring locks, thereby avoiding deadlocks and reducing contention. Wait-free algorithms strengthen this guarantee by ensuring every thread makes progress in a bounded number of steps.

- **Characteristics:**
  - **Atomic primitives** such as Compare-and-Swap (CAS) are often used.
  - They are challenging to design but critical in high-performance, low-latency systems.
  - C++ provides `std::atomic` with methods like `compare_exchange_strong` to enable these designs.
- **Example: Lock-Free Stack Sketch**

```
#include <atomic>
#include <memory>

template<typename T>
class LockFreeStack {
    struct Node {
        T data;
        Node* next;
        Node(T val) : data(val), next(nullptr) {}
    };
    std::atomic<Node*> head{nullptr};

public:
    void push(T value) {
```

```
        Node* newNode = new Node(value);
        newNode->next = head.load();
        while (!head.compare_exchange_weak(newNode->next, newNode)) {
            // retry until success
        }
    }

    bool pop(T& result) {
        Node* oldHead = head.load();
        while (oldHead &&
               !head.compare_exchange_weak(oldHead, oldHead->next)) {}
        if (!oldHead) return false;
        result = oldHead->data;
        delete oldHead;
        return true;
    }
};
```

This sketch demonstrates non-blocking stack operations using CAS. However, real-world lock-free algorithms require careful handling of memory reclamation (e.g., hazard pointers, epoch-based reclamation).

23.1.4 Guidelines for Using Concurrency Primitives

1. **Prefer high-level abstractions** (like `std::async` or parallel STL in C++17) unless low-level control is necessary.
2. **Use atomics for simple counters and flags** instead of mutexes for efficiency.
3. **Keep critical sections small** to reduce contention.
4. **Avoid mixing synchronization mechanisms** without a clear design.

5. **Consider lock-free only when mutexes become a bottleneck**, as correctness and memory management are significantly harder.

23.1.5 Summary

Threading primitives in modern C++ empower developers to harness multicore hardware efficiently. While `std::thread`, `std::mutex`, and `std::atomic` provide the essential building blocks, designing correct and performant concurrent algorithms requires deep understanding of synchronization trade-offs. Lock-free programming represents the frontier of performance optimization, but must be approached with caution due to its complexity. Together, these tools form the foundation for the advanced parallel and concurrent algorithms discussed in the remainder of this chapter.

23.2 Parallel Algorithms (`std::execution`) and Work-Stealing Patterns

With the rise of multicore architectures, leveraging parallelism has become a cornerstone of algorithm design. C++17 introduced standardized **parallel algorithms** through the `<execution>` library, providing high-level abstractions for parallel execution of existing STL algorithms. In addition, modern runtimes often use **work-stealing schedulers** to maximize CPU utilization and reduce idle time. This section explores both approaches and their significance for high-performance algorithm design in C++.

23.2.1 Parallel Algorithms in C++17 and Beyond

Traditionally, developers had to explicitly manage threads, synchronization, and partitioning of work to benefit from concurrency. The standardization of parallel algorithms offers a declarative way to express parallel intent without manually orchestrating thread management.

- **Execution Policies**

  The `<execution>` header provides three major execution policies:

  1. **`std::execution::seq`**
     Executes sequentially (default, same as pre-C++17 STL).
  2. **`std::execution::par`**
     Executes in parallel, dividing the work among multiple threads.
  3. **`std::execution::par_unseq`**
     Executes in parallel and allows vectorization (SIMD), enabling hardware-level parallelism.

- **Example: Parallel Sorting**

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> data(1'000'000);
    std::iota(data.begin(), data.end(), 0);
    std::shuffle(data.begin(), data.end(),
    ↪  std::mt19937{std::random_device{}()});

    // Parallel sort using execution policy
    std::sort(std::execution::par, data.begin(), data.end());

    std::cout << "First 5 elements after sort: ";
    for (int i = 0; i < 5; ++i) std::cout << data[i] << " ";
}
```

Here, the sorting workload is distributed among available CPU cores, significantly reducing runtime for large datasets compared to sequential execution.

23.2.2 Benefits of Parallel STL

- **Ease of use:** Minimal code changes required; just add an execution policy.
- **Portability:** Rely on the standard library to optimize for the platform.
- **Safety:** The algorithms automatically handle partitioning and synchronization, reducing the chance of race conditions.
- **Scalability:** Designed to scale with increasing core counts.

23.2.3 Limitations and Considerations

- **Data Dependencies:** Algorithms must not introduce data races; for example, `std::for_each(std::execution::par, …)` should not modify the same element from multiple threads.
- **Non-determinism:** Parallel execution may result in non-deterministic iteration order.
- **Overheads:** For small problem sizes, parallelism overhead may outweigh benefits.
- **Implementation-Dependent:** Performance varies depending on the standard library implementation and hardware.

23.2.4 Work-Stealing Patterns

While the parallel STL abstracts parallelism at a high level, the runtime often relies on **work-stealing** schedulers to balance load across threads.

- **What is Work-Stealing?**

  Work-stealing is a scheduling strategy where:

  - Each worker thread maintains its own deque (double-ended queue) of tasks.
  - When a worker thread runs out of work, it attempts to "steal" tasks from the deques of other workers.
  - Typically, threads pop tasks from the front of their own deque but steal from the back of another thread's deque, minimizing contention.

- **Advantages of Work-Stealing**

  1. **Load Balancing:** Dynamically redistributes work to keep all cores busy.

2. **Scalability:** Reduces bottlenecks in centralized task queues.
3. **Cache Efficiency:** Threads preferentially execute their own tasks, benefiting from locality.
4. **Robustness:** Handles irregular workloads, e.g., recursive algorithms like quicksort or graph traversals.

23.2.5 Example: Work-Stealing in Practice

Although the standard does not expose work-stealing directly, many implementations of `std::execution::par` rely on it. For custom control, developers may use frameworks like **Intel Threading Building Blocks (TBB)**, which pioneered task-based parallelism.

Example with TBB (conceptual)

```
#include <tbb/parallel_for.h>
#include <vector>
#include <iostream>

int main() {
    std::vector<int> data(1'000'000, 1);
    int sum = 0;

    tbb::parallel_for(size_t(0), data.size(), [&](size_t i) {
        // Each chunk of work can be scheduled dynamically
        sum += data[i];
    });

    std::cout << "Sum = " << sum << "\n";
}
```

Here, TBB uses a work-stealing scheduler to distribute chunks of the loop iteration space across threads efficiently.

23.2.6 Combining Parallel STL and Work-Stealing

The key insight is that **parallel STL algorithms abstract away thread management**, while internally leveraging **efficient scheduling techniques like work-stealing**. Developers benefit from:

- Expressive, high-level code (parallel STL).
- Scalable, load-balanced execution (work-stealing).
- Reduced maintenance complexity compared to hand-rolled threading logic.

23.2.7 Guidelines for Use

1. **Prefer `std::execution::par`** for data-parallel problems where elements can be processed independently.
2. **Use `par_unseq`** when vectorization and multicore parallelism can both apply, but ensure no data races.
3. **Measure performance**—parallelization may introduce overhead for small datasets.
4. **Fall back to libraries like TBB or OpenMP** when finer-grained control of work-stealing or task scheduling is necessary.
5. **Design algorithms with locality in mind** so that stolen tasks benefit from cache reuse.

23.2.8 Summary

Parallel algorithms via `std::execution` mark a milestone in C++'s evolution, making parallelism accessible through standardized, STL-like constructs. Meanwhile, work-stealing patterns ensure that underlying schedulers distribute workloads efficiently, especially for irregular or recursive tasks. Together, these techniques enable developers to design algorithms that are both high-level in expression and low-level in performance efficiency, making them indispensable tools for modern C++ programmers targeting multicore and manycore architectures.

23.3 Exercises: Parallel Prefix Sum, Concurrent Queues

Practical exercises are essential for reinforcing the theory of concurrency and parallelism. This section presents two core exercises that combine algorithmic concepts with modern C++ concurrency tools:

1. **Parallel Prefix Sum (Scan):** A fundamental building block in parallel programming.
2. **Concurrent Queues:** A staple data structure for producer–consumer patterns and task scheduling.

Both exercises highlight the challenges and strategies for efficient and safe parallelism.

23.3.1 Parallel Prefix Sum (Scan)

- **Problem Statement**

  Given an array `A` of length `n`, compute an array `P` where:

  $$P[i] = A[0] + A[1] + \cdots + A[i]$$

  This is a *prefix sum* or *scan*. Sequentially, it runs in $O(n)$. The challenge is to design a parallel algorithm that achieves sublinear depth while preserving correctness.

- **Parallel Algorithm: Up-Sweep and Down-Sweep**

  A standard approach is the **Blelloch scan**:

1. **Up-Sweep (Reduction):** Build a tree of partial sums.
2. **Down-Sweep:** Propagate prefix sums down the tree to compute final results.

Both phases run in $O(\log n)$ depth and $O(n)$ total work.

- **C++ Implementation with Threads**

```
#include <iostream>
#include <vector>
#include <thread>
#include <execution>
#include <numeric>

void parallel_prefix_sum(std::vector<int>& data) {
    size_t n = data.size();
    std::vector<int> prefix(n);

    // Use std::inclusive_scan with parallel execution policy
    std::inclusive_scan(std::execution::par,
                        data.begin(), data.end(),
                        prefix.begin());

    data = std::move(prefix);
}

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5};
    parallel_prefix_sum(arr);

    for (auto x : arr) std::cout << x << " ";
    // Output: 1 3 6 10 15
}
```

Here, `std::inclusive_scan` with `std::execution::par` leverages built-in parallelism, abstracting the up-sweep/down-sweep phases.

- **Key Insights**
  - Parallel prefix sums are used in parallel radix sort, polynomial evaluation, and GPU algorithms.
  - On GPUs, prefix sums are ubiquitous because they transform irregular problems into structured ones.
  - In C++20, `std::inclusive_scan` and `std::exclusive_scan` provide direct, standard-conforming solutions.

23.3.2 Concurrent Queues

- **Motivation**

  Concurrent queues are indispensable in producer–consumer systems, task schedulers, and messaging frameworks. They allow multiple threads to safely push and pop elements without corrupting state.

- **Sequential Baseline**

```
#include <queue>
#include <mutex>

template <typename T>
class ThreadSafeQueue {
    std::queue<T> q;
    std::mutex m;
```

```
public:
    void push(const T& value) {
        std::lock_guard<std::mutex> lock(m);
        q.push(value);
    }

    bool try_pop(T& result) {
        std::lock_guard<std::mutex> lock(m);
        if (q.empty()) return false;
        result = q.front();
        q.pop();
        return true;
    }
};
```

This implementation uses a single mutex to serialize access. While correct, it can become a bottleneck under heavy contention.

- **Lock-Free Ideas with Atomics**

  Lock-free queues improve scalability by using atomic operations (e.g., compare-and-swap) rather than coarse-grained locks. The **Michael–Scott queue** is a classic example.

  Simplified C++ sketch:

```
#include <atomic>
#include <memory>

template <typename T>
class LockFreeQueue {
```

```
    struct Node {
        std::shared_ptr<T> data;
        std::atomic<Node*> next{nullptr};
        Node(T val) : data(std::make_shared<T>(val)) {}
    };

    std::atomic<Node*> head;
    std::atomic<Node*> tail;

public:
    LockFreeQueue() {
        Node* dummy = new Node(T{});
        head.store(dummy);
        tail.store(dummy);
    }

    void push(T value) {
        Node* new_node = new Node(value);
        Node* old_tail = tail.exchange(new_node);
        old_tail->next.store(new_node);
    }

    std::shared_ptr<T> pop() {
        Node* old_head = head.load();
        Node* next = old_head->next.load();
        if (!next) return nullptr; // empty
        head.store(next);
        std::shared_ptr<T> res = next->data;
        delete old_head;
        return res;
    }
};
```

This demonstrates the use of atomics for non-blocking enqueue/dequeue. Implementations like this are the foundation of high-performance concurrent systems.

23.3.3 Exercise Variations

1. **Parallel Prefix Sum Exercises**
   - Implement the Blelloch scan manually using recursive divide-and-conquer and threads.
   - Compare performance of sequential `std::inclusive_scan` vs parallel execution.
   - Extend prefix sums to two dimensions (e.g., cumulative sums for matrices).
2. **Concurrent Queue Exercises**
   - Extend the thread-safe queue to support blocking `pop` with `std::condition_variable`.
   - Benchmark lock-based vs lock-free queue implementations under varying contention.
   - Apply concurrent queues to implement a thread pool, where worker threads repeatedly pull tasks.

23.3.4 Summary

- **Parallel prefix sums** showcase how associative operations can be restructured to exploit parallel hardware.
- **Concurrent queues** embody the challenges of synchronization and lock-free design in multi-threaded systems.

- These exercises bridge algorithmic thinking with systems-level concurrency, reinforcing the skills necessary for building efficient and scalable software in C++.

Chapter 24

Metaprogramming & Compile-time Algorithms

24.1 Template Metaprogramming Basics for Algorithmic Tasks

Template metaprogramming (TMP) in C++ refers to the use of the compiler's template system to perform computations at **compile time**. Initially discovered as a side effect of C++ templates in the 1990s, TMP has since evolved into a core paradigm, especially in high-performance and systems programming. With the advent of **constexpr** and **concepts** in C++11 through C++20, TMP is more accessible, expressive, and integrated with the rest of the language.

This section introduces foundational techniques in template metaprogramming, focusing on **compile-time factorial** as an example of recursive computation, and **type lists** as a structure for manipulating types at compile time.

24.1.1 Compile-Time Computation with Templates

Before `constexpr` functions were introduced, recursive template instantiation was the main way to achieve compile-time evaluation. The compiler effectively "executes" the recursion during template instantiation.

Example: Compile-Time Factorial

```
#include <iostream>

// Template recursion
template <int N>
struct Factorial {
    static constexpr int value = N * Factorial<N - 1>::value;
};
```

```cpp
// Base case specialization
template <>
struct Factorial<0> {
    static constexpr int value = 1;
};

int main() {
    constexpr int f5 = Factorial<5>::value; // Computed at compile time
    std::cout << "Factorial(5) = " << f5 << '\n'; // Output: 120
}
```

- The recursion unfolds at compile time, generating constants.
- `Factorial<5>::value` does not require runtime computation; it's replaced directly with `120` by the compiler.

With modern C++, the same can be expressed more clearly with `constexpr`:

```cpp
constexpr int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}

int main() {
    constexpr int f6 = factorial(6); // Evaluated at compile time
}
```

Comparison: TMP recursion is powerful but verbose. `constexpr` is now preferred for numeric computations, though TMP remains indispensable for **type-level programming**.

24.1.2 Type Lists: Computation with Types

While numeric metaprogramming demonstrates compile-time arithmetic, type-level programming is TMP's most distinctive capability. A **type list** is a compile-time container of types, enabling transformations, filtering, and querying during compilation.

- **Basic Type List Definition**

```
// A simple variadic template to hold a list of types
template <typename... Ts>
struct TypeList {};
```

This `TypeList` can hold any number of types:

```
using MyTypes = TypeList<int, double, char>;
```

- **Operations on Type Lists**

Type lists enable algorithms where types are the input. For example:

1. **Length of a Type List**

```
template <typename TList>
struct Length;

template <typename... Ts>
struct Length<TypeList<Ts...>> {
    static constexpr size_t value = sizeof...(Ts);
};

using L = TypeList<int, float, double>;
static_assert(Length<L>::value == 3);
```

2. Accessing the N-th Type

```
template <size_t N, typename TList>
struct TypeAt;

template <size_t N, typename Head, typename... Tail>
struct TypeAt<N, TypeList<Head, Tail...>> : TypeAt<N - 1,
↪  TypeList<Tail...>> {};

template <typename Head, typename... Tail>
struct TypeAt<0, TypeList<Head, Tail...>> {
    using type = Head;
};

using T = TypeAt<1, TypeList<int, double, char>>::type;
// T is double
```

3. Appending a Type

```
template <typename TList, typename NewType>
struct Append;

template <typename... Ts, typename NewType>
struct Append<TypeList<Ts...>, NewType> {
    using type = TypeList<Ts..., NewType>;
};

using Extended = Append<TypeList<int, char>, double>::type;
// Extended = TypeList<int, char, double>
```

24.1.3 Applications of Template Metaprogramming

- **Static Assertions:** Verify properties of types at compile time.
- **Type Traits:** Implement custom traits (e.g., checking if a type is integral).
- **Policy-Based Design:** Compose classes with type-level parameters.
- **Generic Libraries:** Libraries like Boost MPL and modern C++ ranges rely on type-level computations.

24.1.4 Modern TMP vs Historical TMP

- **Classic TMP (C++98/03):** Relied heavily on recursive template instantiation (e.g., factorial, Fibonacci). Powerful but cryptic.
- **Modern TMP (C++11 and beyond):** Introduces `constexpr`, `decltype`, `constexpr if`, fold expressions, and concepts, making compile-time algorithms more intuitive while preserving type-level metaprogramming when needed.

Example using fold expression for factorial:

```
template <int... Ns>
constexpr int factorial_pack() {
    return ( ... * Ns ); // fold expression
}
```

24.1.5 Summary

Template metaprogramming elevates C++ beyond runtime computation by embedding **compile-time algorithms** into the compilation process. The compile-time factorial

illustrates recursive instantiation, while type lists show how to manipulate **types as data**. These building blocks form the foundation for more advanced metaprogramming topics such as SFINAE, concepts, constexpr metaprogramming, and expression templates, all of which are essential for writing high-performance, generic C++ code.

24.2 Concepts & `constexpr` Algorithms in C++20/23 (constexpr Sorting, Compile-time DP)

C++20 and C++23 introduced new tools that elevate metaprogramming to a more expressive and approachable paradigm. With **Concepts** and expanded support for `constexpr`, algorithms once thought to be runtime-only can now be executed during compilation, enabling compile-time checks, precomputed values, and powerful optimization opportunities. This section explores how these features can be leveraged for algorithmic design.

24.2.1 Concepts and Constrained Algorithms

Concepts are compile-time predicates that allow developers to constrain template parameters more precisely than with SFINAE (Substitution Failure Is Not An Error). This ensures that algorithms fail early and meaningfully when instantiated with incompatible types.
For example, a generic sorting function might be constrained to only work with **random-access iterators**:

```
#include <concepts>
#include <iterator>
#include <algorithm>
#include <vector>

template <std::random_access_iterator Iter>
constexpr void constrained_sort(Iter first, Iter last) {
    std::sort(first, last);
}
```

Here, `std::random_access_iterator` is a standard concept, ensuring the algorithm

cannot be instantiated with, for instance, a forward-only list iterator. This prevents misuse and generates clearer diagnostics at compile time.
Concepts also allow you to design domain-specific constraints, such as restricting algorithms to integral types or floating-point types:

```
#include <concepts>
#include <numeric>

template <std::integral T>
constexpr T sum_to(T n) {
    return (n * (n + 1)) / 2;
}
```

This guarantees that `sum_to` cannot be called with floating-point or complex number types, providing both safety and clarity.

24.2.2 Expanded `constexpr` in C++20/23

C++20 significantly expanded the set of constructs allowed in `constexpr` functions, including dynamic allocation (in some contexts), try/catch (though exceptions cannot be thrown), and most of the standard library algorithms. This allows algorithms to execute during compilation if provided with constant expressions.
C++23 continued this expansion, making more of the standard library usable at compile time. For instance, container operations with `std::vector` and `std::string` can now often be performed in `constexpr` contexts.

24.2.3 Example: `constexpr` Sorting

Sorting at compile time is a canonical demonstration of modern `constexpr` power. A simple implementation might use selection sort or bubble sort, since these are easier to

implement with recursion and fixed bounds:

```
#include <array>
#include <utility>

template <typename T, std::size_t N>
constexpr void bubble_sort(std::array<T, N>& arr) {
    for (std::size_t i = 0; i < N; ++i) {
        for (std::size_t j = 0; j < N - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}

constexpr auto sorted = [] {
    std::array<int, 5> arr{5, 3, 4, 1, 2};
    bubble_sort(arr);
    return arr;
}();
```

The array `sorted` is fully computed at compile time, and the compiler embeds the sorted result directly in the binary.

Although bubble sort is inefficient at runtime, it is often acceptable for compile-time computation where the input size is small and bounded. For larger datasets, more sophisticated constexpr algorithms (like quicksort or mergesort) can be written with recursion.

24.2.4 Compile-time Dynamic Programming

Dynamic Programming (DP) can also be applied in a `constexpr` context. This is especially useful for algorithmic tasks where precomputed results save runtime effort, such as computing Fibonacci numbers, binomial coefficients, or shortest paths in small graphs.

Example: Fibonacci with Memoization

```
#include <array>

constexpr int fib(int n) {
    if (n <= 1) return n;

    // Compile-time memoization using recursion and static array
    constexpr int maxN = 50;
    static std::array<int, maxN + 1> memo{};
    static bool initialized = false;

    if (!initialized) {
        memo[0] = 0;
        memo[1] = 1;
        for (int i = 2; i <= maxN; ++i) {
            memo[i] = memo[i - 1] + memo[i - 2];
        }
        initialized = true;
    }
    return memo[n];
}

constexpr int fib20 = fib(20);  // Computed entirely at compile time
```

This approach precomputes Fibonacci numbers during compilation, embedding them

directly in the program binary without runtime overhead.

24.2.5 Practical Applications of `constexpr` Algorithms

- **Precomputation**: Embed lookup tables, prime sieves, or sorting orders directly into the executable.
- **Validation**: Perform structural checks (e.g., graph connectivity, type constraints) before the program runs.
- **Optimization**: Avoid runtime costs for fixed or configuration-dependent data.
- **Safer APIs**: Combine concepts and constexpr to reject invalid inputs at compile time.

24.2.6 Limitations and Best Practices

- `constexpr` algorithms should be bounded; deeply recursive or unbounded computations may cause long compile times.
- Prefer simpler algorithms (like bubble sort) for small fixed inputs; only implement complex algorithms if compile-time efficiency matters.
- Use `concepts` to guard against unintended instantiations and ensure meaningful diagnostics.
- Reserve compile-time computation for data or checks that genuinely benefit from being precomputed.

Summary:
C++20/23's `constexpr` expansion and `concepts` empower developers to implement meaningful compile-time algorithms. From `constexpr` sorting to dynamic programming,

these features push computation from runtime to compile time, enhancing performance and safety. By blending constraints and precomputation, C++ now offers a powerful paradigm for writing algorithms that are both correct and efficient before the program ever runs.

24.3 Exercises: Static-Sequence Algorithms, `consteval` Usage

This section provides exercises that encourage hands-on practice with compile-time techniques, focusing on **static-sequence algorithms** and the use of **`consteval` functions** in C++20/23. By experimenting with these exercises, readers gain practical familiarity with how compile-time programming can replace runtime work, ensure correctness, and improve program efficiency.

24.3.1 Static-Sequence Algorithms

Static sequences, such as `std::integer_sequence` or `std::index_sequence`, are central to modern metaprogramming. They provide a way to represent lists of numbers or indices **at compile time**, which can then be used to parameterize templates, unroll loops, or generate code.

- **Example 1: Generating Factorials with `std::integer_sequence`**

```
#include <array>
#include <utility>
#include <cstddef>

constexpr std::size_t factorial(std::size_t n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}

template <std::size_t... Indices>
constexpr auto make_factorial_array(std::index_sequence<Indices...>) {
    return std::array<std::size_t, sizeof...(Indices)>{ factorial(Indices)...
    ↪  };
```

```
}

constexpr auto factorials =
↪  make_factorial_array(std::make_index_sequence<11>{});
// factorials = {0! = 1, 1! = 1, 2! = 2, ..., 10! = 3628800}
```

Here, `std::index_sequence` generates a static sequence of integers, which we expand into calls to `factorial`. The result is a precomputed array of factorial values, embedded directly into the program.

- **Example 2: Compile-time Prime Check for Static Sequences**

```
#include <array>
#include <utility>
#include <cstddef>

constexpr bool is_prime(std::size_t n) {
    if (n < 2) return false;
    for (std::size_t i = 2; i * i <= n; ++i) {
        if (n % i == 0) return false;
    }
    return true;
}

template <std::size_t... Ns>
constexpr auto prime_table(std::index_sequence<Ns...>) {
    return std::array<bool, sizeof...(Ns)>{ is_prime(Ns)... };
}

constexpr auto primes_up_to_20 = prime_table(std::make_index_sequence<21>{});
// primes_up_to_20[i] is true if i is prime
```

This technique demonstrates how entire tables of computed properties (like primality) can be constructed during compilation.

24.3.2 `consteval` Usage

The `consteval` specifier, introduced in C++20, marks functions that **must** be evaluated at compile time. Unlike `constexpr`, which permits both runtime and compile-time evaluation depending on the context, `consteval` enforces compile-time execution. This is useful for guaranteeing that certain values or computations are always resolved before runtime.

- **Example 3: Enforcing Compile-time ID Generation**

```
#include <string_view>

consteval unsigned int fixed_hash(std::string_view str) {
    unsigned int hash = 0;
    for (char c : str) {
        hash = hash * 131 + static_cast<unsigned int>(c);
    }
    return hash;
}

constexpr unsigned int id_user = fixed_hash("user");
constexpr unsigned int id_admin = fixed_hash("admin");
```

Here, any attempt to call `fixed_hash` with a runtime string will fail to compile, ensuring identifiers are determined at build time.

- **Example 4: Compile-time Dimension Validation**

```cpp
#include <array>
#include <cstddef>

template <std::size_t N>
consteval void validate_size() {
    static_assert(N > 0 && N <= 16, "Matrix dimension must be between 1 and
    ↪  16");
}

template <std::size_t N>
struct Matrix {
    std::array<std::array<double, N>, N> data{};
    consteval Matrix() { validate_size<N>(); }
};

constexpr Matrix<4> m1;   // OK
// constexpr Matrix<32> m2; // Compile-time error
```

This exercise shows how `consteval` can enforce constraints early, preventing invalid instantiations of templates.

24.3.3 Exercise Ideas

1. **Compile-time Fibonacci Sequence with `std::integer_sequence`**
   Generate a fixed array of Fibonacci numbers at compile time using recursive `constexpr` functions and sequence expansion.

2. **Static Prefix Sum Array**
   Create a compile-time prefix sum array (e.g., `arr[i] = sum of 0..i`) and use it in a `constexpr` context.

3. **`consteval` String Length Validator**
   Implement a `consteval` function that validates that a string literal's length does not exceed a given bound, ensuring compile-time safety for buffer sizes.

4. **Compile-time Permutation Generator**
   Using `std::integer_sequence`, implement a function that generates all permutations of `{0, 1, …, N-1}` at compile time (for small `N`).

5. **`consteval` Configuration Enforcement**
   Write a configuration header where parameters (like buffer sizes, thread counts, etc.) must be compile-time constants enforced with `consteval`.

24.3.4 Best Practices for Exercises

- Use `std::integer_sequence` and `std::index_sequence` for elegant static-sequence algorithms.
- Prefer `consteval` when you want *strict compile-time guarantees*.
- Avoid heavy or unbounded compile-time computation, as it can slow compilation drastically.
- Combine `concepts`, `constexpr`, and `consteval` for maximum safety and expressiveness.

Summary:
These exercises demonstrate how static sequences and `consteval` functions enrich compile-time algorithm design. They guide learners in precomputing tables, validating constraints, and enforcing compile-time guarantees, providing both safety and efficiency. By mastering these tools, programmers can push more correctness into the compiler and reduce runtime overhead.

Chapter 25

Profiling, Benchmarking & Optimization Workflow

25.1 Using Profilers (gprof, perf), Sanitizers (ASAN, UBSAN), and Compiler Flags

Effective optimization in C++ does not start with guesswork but with **measurement and diagnostics**. Modern toolchains provide a wealth of facilities for **profiling performance**, **detecting correctness issues**, and **fine-tuning code generation**. This section introduces three essential categories of tools: **profilers**, **sanitizers**, and **compiler flags**, each of which plays a distinct role in an optimization workflow.

25.1.1 Profilers

Profilers measure how a program spends its time and resources during execution. They identify performance bottlenecks, guide optimizations, and ensure improvements target

the right hotspots.

- **gprof (GNU profiler)**
  - **Integration**: Requires compiling with `-pg` flag.
  - **Workflow**:
    1. Compile and link with `-pg`.
    2. Run the program; it produces a `gmon.out` file.
    3. Analyze with `gprof ./a.out gmon.out`.
  - **Output**:
    * Flat profile: shows time spent per function.
    * Call graph: shows call relationships and percentages of execution time.
  - **Limitations**:
    * Sampling resolution is low by modern standards.
    * Overhead can affect runtime behavior.
    * Largely superseded by newer tools but still useful for small projects or teaching.
- **perf (Linux performance analysis tool)**
  - **Integration**: Available on modern Linux systems, requires no recompilation.
  - **Workflow**:
    * Run `perf record ./program` to collect performance events.
    * Use `perf report` or `perf annotate` for detailed function-level or instruction-level insights.
  - **Advantages**:

    * Low-overhead sampling using CPU hardware counters.
    * Works across languages and binaries.
    * Can measure cache misses, branch mispredictions, and other microarchitectural events.
  - **Use Cases**: Identifying slow code paths, understanding instruction-level inefficiencies, and tuning performance-critical systems.

25.1.2 Sanitizers

Sanitizers extend debugging and runtime validation by detecting **undefined behavior**, **memory errors**, and **threading issues** that are otherwise difficult to catch. They are part of GCC and Clang toolchains.

- **AddressSanitizer (ASAN)**
  - Detects memory errors such as:
    * Out-of-bounds reads and writes.
    * Use-after-free bugs.
    * Memory leaks (with leak detection enabled).
  - Usage: Compile with `-fsanitize=address -g -O1`.
  - Example:

```
#include <iostream>

int main() {
    int arr[3] = {1, 2, 3};
    std::cout << arr[3] << "\n"; // out-of-bounds
}
```

Running under ASAN produces a detailed runtime error pointing to the source of the memory violation.

- **Undefined Behavior Sanitizer (UBSAN)**
  - Detects undefined behaviors such as:
    - Integer overflows (signed).
    - Invalid shifts.
    - Misaligned memory accesses.
    - Use of null references.
  - Usage: Compile with `-fsanitize=undefined`.
  - Provides runtime warnings instead of immediate crashes.
- **Other sanitizers worth noting**
  - **ThreadSanitizer (TSAN):** Detects data races in multithreaded programs.
  - **MemorySanitizer (MSAN):** Detects uses of uninitialized memory.
  - Each sanitizer introduces overhead but dramatically reduces debugging time for correctness issues.

25.1.3 Compiler Flags

Compiler flags affect both performance diagnostics and code generation. Correct flag usage can surface hidden issues, enforce standards compliance, and squeeze more efficiency out of algorithms.

- **Optimization Levels**
  - `-O0`: No optimizations (useful for debugging).

  - `-O1`, `-O2`, `-O3`: Increasing levels of optimization (loop unrolling, inlining, vectorization).
  - `-Ofast`: Aggressive optimizations, may disregard strict standards compliance.
  - `-Os`: Optimize for binary size.

- **Warning and Error Flags**

  - `-Wall -Wextra`: Enables common warnings.
  - `-Werror`: Treats warnings as errors, enforcing stricter coding discipline.
  - `-pedantic`: Enforces standard-compliance.

- **Profiling/Debugging Flags**

  - `-pg`: Instrumentation for `gprof`.
  - `-fno-omit-frame-pointer`: Retain stack frame pointers for accurate profiling and debugging.
  - `-g`: Generate debug symbols for use with profilers and sanitizers.

- **Architecture-Specific Flags**

  - `-march=native`: Enable all CPU-specific optimizations available on the build machine.
  - `-mtune=...`: Tune performance for a specific CPU family.
  - `-funroll-loops`: Aggressively unroll loops where beneficial.

25.1.4 Recommended Workflow

1. **Start with Correctness**:
    - Enable sanitizers (`ASAN`, `UBSAN`) and warnings (`-Wall -Wextra`) to ensure no hidden bugs.
    - Run comprehensive tests under sanitizer builds.
2. **Profile the Application**:
    - Use `perf` for detailed performance metrics.
    - Fall back on `gprof` or sampling profilers if targeting simpler projects.
3. **Tune with Compiler Flags**:
    - Optimize with `-O2` or `-O3`.
    - Add `-march=native` for local builds to exploit CPU features.
    - Benchmark different builds to balance speed, safety, and portability.

Summary:
Profilers like **gprof** and **perf** help locate bottlenecks, sanitizers such as **ASAN** and **UBSAN** catch correctness issues early, and compiler flags provide fine-grained control over optimization and diagnostics. Together, these tools form a systematic workflow for ensuring that C++ algorithms are not only correct but also efficient and robust across platforms.

25.2 Micro-optimizations vs Algorithmic Improvements — Case Studies in C++

Optimization in C++ development must be guided by **evidence, measurement, and priorities**. One of the most common mistakes in performance work is focusing on micro-optimizations—small, localized code tweaks—before addressing the **algorithmic complexity** of the solution. This section distinguishes between micro-level performance improvements and high-level algorithmic changes, providing case studies to illustrate why the latter often dominates real-world performance.

25.2.1 Micro-optimizations

Micro-optimizations focus on **local code transformations** without altering the algorithmic complexity. Examples include:

- Replacing repeated function calls with cached values.
- Using pre-increment (`++i`) instead of post-increment (`i++`) in tight loops (relevant in older compilers).
- Eliminating unnecessary temporaries by using references or move semantics.
- Preferring `emplace_back` over `push_back` for in-place construction in containers.
- **Characteristics**
  - Scope: Localized, often one or two lines of code.
  - Performance gains: Typically a few percentage points at most.
  - Maintainability trade-offs: Sometimes makes code harder to read with minimal impact.

  - When useful: After profiling identifies a true hotspot where such changes matter.

- **Example: Loop indexing**

```
for (size_t i = 0; i < vec.size(); ++i) {
    process(vec[i]);
}
```

Here, the repeated `vec.size()` call could be avoided:

```
for (size_t i = 0, n = vec.size(); i < n; ++i) {
    process(vec[i]);
}
```

This saves a redundant call per iteration, which matters in very tight loops but is negligible compared to the complexity of the underlying algorithm.

25.2.2 Algorithmic Improvements

Algorithmic improvements target the **asymptotic complexity** of the problem. They usually involve rethinking the approach rather than tweaking implementation details.

- **Characteristics**

  - Scope: Larger changes in data structures or algorithm choice.
  - Performance gains: Can reduce runtime by orders of magnitude.
  - Maintainability: Often improves clarity, since efficient algorithms are well-studied and standardized.

  - When useful: Always consider algorithmic complexity first; micro-optimizations only make sense after this step.

- **Example: Sorting**

  - Naive bubble sort: **O(n²)**.
  - Merge sort or quicksort: **O(n log n)**.
  - Using `std::sort`: Highly optimized introsort (quicksort + heapsort fallback).

  Replacing bubble sort with `std::sort` on 100,000 elements reduces runtime from minutes to milliseconds—an improvement several magnitudes greater than any micro-tweak inside the bubble sort implementation.

25.2.3 Case Study: Searching in C++

- **Micro-optimized linear search**

```
int find_linear(const std::vector<int>& data, int key) {
    for (size_t i = 0; i < data.size(); ++i) {
        if (data[i] == key) return i;
    }
    return -1;
}
```

  Even if we add minor tweaks (like caching `data.size()`), the complexity remains **O(n)**.

- **Algorithmic improvement: binary search**

By sorting the data and using `std::binary_search`, the complexity reduces to **O(log n)**:

```
bool exists = std::binary_search(data.begin(), data.end(), key);
```

For large inputs, this shift in algorithm dominates performance by several orders of magnitude.

25.2.4 Case Study: Matrix Multiplication

- **Naive approach**: Three nested loops → **O(n³)**.
- **Micro-optimization**: Unroll inner loops, adjust cache usage → ~10–20% improvement.
- **Algorithmic improvement**: Strassen's algorithm (**O(n^2.81)**) or more advanced algorithms, combined with cache-aware tiling, reduces runtime by factors, not percentages.

This shows that algorithm selection has far greater impact than micro-tuning within a poor algorithm.

25.2.5 Guiding Principles

1. **Measure first**: Use profilers (`perf`, `gprof`) and benchmarking frameworks to locate bottlenecks.
2. **Fix algorithms before code**: Always prioritize reducing asymptotic complexity.
3. **Apply micro-optimizations in hotspots only**: Focus on the ~5% of code consuming 95% of execution time (Pareto principle).

4. **Leverage the STL and standard algorithms**: Many standard library implementations already combine optimal algorithms with micro-optimizations.

Summary:
Micro-optimizations in C++ can provide small incremental speedups, but **algorithmic improvements are the foundation of efficient programming**. Case studies in sorting, searching, and matrix multiplication show that reducing complexity often yields improvements by factors of 10x, 100x, or more, compared to the single-digit percentage gains of micro-optimizations. The practical takeaway is to **optimize algorithms first, measure with tools, then refine with micro-optimizations where necessary**.

25.3 Exercises — Profile and Improve Small C++ Projects

The best way to internalize profiling and optimization practices is through **hands-on exercises**. By applying profiling tools, identifying bottlenecks, and implementing targeted improvements, readers can move from theory to practical mastery. This section presents a series of small-scale projects designed to build intuition about performance, correctness, and maintainability.

25.3.1 Exercise: Profiling a Naive Sorting Benchmark

Task:

- Implement a naive bubble sort and compare it against `std::sort`.
- Profile both versions using `perf` (Linux) or `gprof` (GNU toolchain).

Steps:

1. Write a program that generates a vector of random integers (e.g., 100,000 elements).
2. Implement bubble sort manually.
3. Compare its runtime with `std::sort`.
4. Profile both implementations and observe differences in function call frequency and CPU cycles.

Goal:
Understand how **algorithmic complexity dominates runtime** and how profiling can confirm the true hotspots.

25.3.2 Exercise: Memory Leak Detection with ASAN

Task:

- Create a small project that allocates memory dynamically but forgets to free it.
- Run the program with AddressSanitizer enabled.

Steps:

1. Write code that allocates an array with `new[]` but omits `delete[]`.
2. Compile with `-fsanitize=address -g`.
3. Run the program and observe the sanitizer output.
4. Fix the leak by using `std::vector` or `std::unique_ptr`.

Goal:
Gain experience with sanitizers and learn why **RAII (Resource Acquisition Is Initialization)** is the preferred idiom in modern C++.

25.3.3 Exercise: Cache-Aware Optimization

Task:

- Implement matrix multiplication in two ways: row-major and column-major iteration.
- Compare runtimes using different matrix sizes.

Steps:

1. Write a naive three-loop multiplication ($O(n^3)$).

2. Measure performance with cache-friendly iteration order (accessing contiguous memory).
3. Profile using `perf stat` to observe cache miss rates.
4. Compare improvements achieved solely through **memory layout awareness**.

Goal:
Understand how **cache locality** impacts performance, even when algorithmic complexity remains the same.

25.3.4 Exercise: Micro-Optimization vs Algorithmic Improvement

Task:

- Compare linear search (`O(n)`) with binary search (`O(log n)`) on large datasets.
- Apply small micro-optimizations to linear search and see if they ever outperform binary search.

Steps:

1. Implement both searches.
2. Benchmark on vectors of increasing sizes (1e3, 1e5, 1e7).
3. Try caching loop bounds or unrolling loops in the linear version.
4. Compare results with `std::binary_search`.

Goal:
Discover that **algorithmic improvements vastly outweigh micro-optimizations** in practical contexts.

25.3.5 Exercise: Multithreaded vs Single-Threaded Performance

Task:

- Implement a simple prime-checking program that tests a large range of integers.
- Profile the single-threaded version and compare it with a multi-threaded version using `std::thread` or `std::async`.

Steps:

1. Write a function to check primality of an integer.
2. Apply it to a range (e.g., 1 to 1,000,000).
3. First, implement a single-threaded version.
4. Then, divide the work across multiple threads.
5. Measure execution time with `std::chrono`.

Goal:
Experience how concurrency can improve throughput while also learning to measure **synchronization costs**.

25.3.6 Project: Profile-and-Improve a Small CLI Tool

Task:
Take a small CLI-based C++ program (e.g., word frequency counter in text files) and profile it.
Steps:

1. Implement using naive I/O (e.g., `std::getline` with repeated string concatenations).
2. Profile with `perf` or `gprof` to identify bottlenecks.
3. Optimize by:
    - Using `std::unordered_map` instead of `std::map`.
    - Using buffered I/O (`std::ifstream::read`) for large files.
    - Avoiding redundant string copies with `std::string_view`.

Goal:
Learn to **iteratively profile and refine** an application based on evidence rather than guesswork.

25.3.7 Recommended Workflow for Each Exercise

1. Write a baseline implementation.
2. Benchmark with simple timers (`std::chrono::high_resolution_clock`).
3. Profile with a tool (`perf`, `gprof`, sanitizers).
4. Apply one improvement at a time (micro or algorithmic).
5. Re-profile after each improvement.
6. Document changes and measure actual gains.

Summary:
These exercises emphasize that **profiling guides optimization**, and that meaningful performance improvements in C++ come from the right combination of **algorithm**

choice, memory awareness, and concurrency. Micro-optimizations play a role, but only after broader algorithmic and structural choices are made. By practicing these profiling-and-improvement cycles on small projects, readers gain the habits needed to scale performance engineering to larger real-world systems.

Part VII

Capstone Projects

Chapter 26

Project A — High-performance Graph Library

26.1 Design Goals, API, Iterators, Memory Layout (CSR)

In this capstone project, we aim to design and implement a **high-performance graph library in Modern C++**. The project serves as a culmination of earlier chapters, combining knowledge of data structures, algorithms, memory management, and performance optimization into a cohesive system. This section lays out the **design goals**, discusses the **API surface**, explains how to provide **iterators** for graph traversal, and examines the **Compressed Sparse Row (CSR)** format for memory efficiency.

26.1.1 Design Goals

The library is guided by several core design principles:

1. **Performance-Centric**
    - Minimize memory overhead through cache-friendly layouts.
    - Provide predictable performance for large-scale graphs (millions of nodes and edges).
    - Exploit Modern C++ features such as move semantics, constexpr, and templates.
2. **Generic and Extensible**
    - Support both **directed and undirected graphs**.
    - Allow weighted and unweighted edges.
    - Provide templates for different index types (`int32_t`, `int64_t`) to adapt to graph size.
3. **Ease of Use**
    - Offer an intuitive API for graph creation, traversal, and manipulation.
    - Follow STL-like conventions so users can quickly learn the library.
    - Emphasize strong type safety while keeping syntax clean.
4. **Interoperability**
    - Allow importing/exporting graphs from common formats (edge lists, adjacency lists).

- Provide iteration interfaces compatible with range-based for-loops.

5. **Scalability**

   - Handle large real-world graphs efficiently (social networks, web graphs, road networks).
   - Provide thread-safe read-only traversal for parallel algorithms.

26.1.2 API Design

The library's API should mirror familiar STL idioms:

- **Graph Construction**

```
Graph<int> g(num_vertices);
g.add_edge(0, 1, 5);   // weighted edge
g.add_edge(2, 3);      // unweighted edge
```

- **Access and Queries**

```
auto neighbors = g.neighbors(0);
bool connected = g.has_edge(0, 1);
auto weight = g.edge_weight(0, 1);
```

- **Iterators**

  – `vertex_iterator` for traversing vertices.

  – `edge_iterator` for traversing edges globally.

  – `adjacency_iterator` for neighbors of a given vertex.

- **Algorithms Integration**

```
auto path = shortest_path(g, 0, 5);
auto mst = minimum_spanning_tree(g);
```

- **Import/Export**

```
g.load_from_edge_list("graph.txt");
g.save_to_csr("graph.csr");
```

The API emphasizes clarity while hiding low-level implementation details such as CSR encoding.

26.1.3 Iterators

Iterators make the graph feel like an STL container. They enable users to traverse efficiently and integrate seamlessly with STL algorithms.

- **Vertex Iterator**
  Allows iteration over all vertices:

```
for (auto v : g.vertices()) {
    std::cout << v << "\n";
}
```

- **Edge Iterator**
  Enables traversal of all edges:

```
for (auto e : g.edges()) {
    std::cout << e.src << " -> " << e.dst << "\n";
}
```

- **Adjacency Iterator**
  Provides neighbors of a vertex:

```
for (auto n : g.neighbors(v)) {
    std::cout << "Neighbor: " << n << "\n";
}
```

The iterators should be **lightweight wrappers** around CSR index structures, ensuring constant-time access and efficient iteration without copying.

26.1.4 Memory Layout: Compressed Sparse Row (CSR)

For high-performance graph representation, the library adopts **Compressed Sparse Row (CSR)** format. CSR is widely used in sparse matrix and graph computations due to its compactness and cache efficiency.

Structure of CSR:

- `row_ptr` (size = `num_vertices + 1`)
  - Stores the starting index of each vertex's adjacency list in the `col_idx` array.
- `col_idx` (size = `num_edges`)
  - Stores the destination vertex of each edge.
- `weights` (optional, size = `num_edges`)

– Stores edge weights if applicable.

Example:
Graph with 3 vertices and edges:

```
0 → 1
0 → 2
1 → 2
```

CSR representation:

- `row_ptr = [0, 2, 3, 3]`
- `col_idx = [1, 2, 2]`
- `weights = [...]` (if weighted)

Advantages of CSR:

1. **Compact Memory** — No pointer overhead per adjacency list.
2. **Cache-Friendly** — Consecutive neighbors are stored contiguously.
3. **Fast Traversal** — Iterating over neighbors is a simple range lookup.
4. **Interoperability** — Matches formats used in numerical libraries (BLAS, sparse solvers).

Trade-offs:

- Insertion/deletion of edges is costly (requires shifting arrays).
- Best suited for **static graphs** or graphs with infrequent updates.

Summary:
This section established the **foundational design principles** of the high-performance graph library, outlined an STL-inspired **API**, introduced **iterators** for seamless integration, and justified the choice of **CSR memory layout** for performance and scalability. With these concepts in place, the project can now progress into algorithmic implementations that leverage the efficient graph representation.

26.2 Implementations — SSSP, MST, Centrality Measures

Once the high-performance graph library is designed with a clear **API**, **iterators**, and **CSR memory layout**, the next step is implementing **core graph algorithms**. This section focuses on three fundamental classes of algorithms: **Single-Source Shortest Path (SSSP)**, **Minimum Spanning Tree (MST)**, and **centrality measures**. The implementations leverage Modern C++ features for **performance, safety, and genericity**.

26.2.1 Single-Source Shortest Path (SSSP)

SSSP algorithms compute the shortest distances from a single source vertex to all other vertices. The choice of algorithm depends on edge weights:

1. **Dijkstra's Algorithm (non-negative weights)**

   - **Implementation Notes:**
     - Uses a **priority queue** (`std::priority_queue`) for selecting the next vertex with the minimum distance.
     - CSR adjacency lists allow fast access to neighbors.
     - Distance vector stored as `std::vector<double>` or templated numeric type.
     - Optional predecessor vector for path reconstruction.

C++ Skeleton:

```cpp
template<typename Graph, typename WeightType>
std::vector<WeightType> dijkstra(const Graph& g, int src) {
    std::vector<WeightType> dist(g.num_vertices(),
    ↪  std::numeric_limits<WeightType>::max());
    dist[src] = 0;
    using PQElem = std::pair<WeightType, int>;
    std::priority_queue<PQElem, std::vector<PQElem>, std::greater<>> pq;
    pq.push({0, src});

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) continue;
        for (auto e : g.neighbors(u)) {
            WeightType new_dist = d + g.edge_weight(u, e);
            if (new_dist < dist[e]) {
                dist[e] = new_dist;
                pq.push({new_dist, e});
            }
        }
    }
    return dist;
}
```

2. **Bellman-Ford (handles negative weights)**

    - Iterates over all edges `|V|-1` times.
    - Detects negative weight cycles.
    - Less efficient than Dijkstra for non-negative weights but necessary when negatives exist.

3. **Implementation Notes**

- CSR format ensures neighbor access is contiguous in memory.
- Use iterators for edge traversal to maintain API consistency.
- Can extend to **multi-source SSSP** by parallelizing over sources with `std::thread` or `std::async`.

26.2.2 Minimum Spanning Tree (MST)

MST algorithms find a subset of edges connecting all vertices with minimal total weight.

1. **Kruskal's Algorithm**

   - **Implementation Notes:**
     - Uses a **Disjoint Set Union (DSU)** data structure with path compression and union by rank.
     - Edges are sorted by weight using `std::sort`.
     - CSR storage is mainly for adjacency access; edges may be represented in a separate vector for sorting.

2. **Prim's Algorithm**

   - **Implementation Notes:**
     - Uses a priority queue to select the next edge with minimal weight.
     - CSR adjacency access allows **O(1)** neighbor iteration.
     - Binary heaps (`std::priority_queue`) are simple; Fibonacci heaps reduce amortized cost for dense graphs.

C++ Skeleton:

```cpp
template<typename Graph, typename WeightType>
std::vector<int> prim(const Graph& g) {
    int n = g.num_vertices();
    std::vector<WeightType> key(n, std::numeric_limits<WeightType>::max());
    std::vector<int> parent(n, -1);
    std::vector<bool> in_mst(n, false);

    key[0] = 0;
    using PQElem = std::pair<WeightType,int>;
    std::priority_queue<PQElem, std::vector<PQElem>, std::greater<>> pq;
    pq.push({0,0});

    while (!pq.empty()) {
        int u = pq.top().second; pq.pop();
        in_mst[u] = true;
        for (auto v : g.neighbors(u)) {
            WeightType w = g.edge_weight(u, v);
            if (!in_mst[v] && w < key[v]) {
                key[v] = w;
                parent[v] = u;
                pq.push({w, v});
            }
        }
    }
    return parent;
}
```

26.2.3 Centrality Measures

Centrality measures help identify **important nodes** in a graph:

1. **Degree Centrality**

- Count of neighbors.
- Extremely simple with CSR: `row_ptr[v+1] - row_ptr[v]`.

2. **Closeness Centrality**

   - Requires shortest-path computation to all other vertices.
   - Can reuse SSSP implementations for efficiency.

3. **Betweenness Centrality**

   - Number of shortest paths passing through a vertex.
   - Implemented using **Brandes' algorithm**.
   - CSR layout reduces memory overhead and improves cache performance during neighbor traversal.

   Implementation Notes:

   - Use **iterators** to traverse neighbors when calculating paths.
   - Parallelize independent shortest path computations using `std::thread` for large graphs.

26.2.4 Performance Considerations

- **CSR memory layout** ensures contiguous neighbor access, minimizing cache misses.
- Algorithms leverage **iterators**, keeping the API consistent across SSSP, MST, and centrality computations.

- Optional templates allow switching numeric types (e.g., `float` vs `double`) for memory savings or precision.
- Profiling during development identifies hotspots for large graphs, guiding optimization priorities.

Summary:
This section demonstrates the practical implementation of **core graph algorithms** in a high-performance C++ library. SSSP, MST, and centrality measures are all implemented using **CSR memory layout** and **iterator-based access**, enabling both efficiency and API consistency. The designs incorporate Modern C++ features such as templates, priority queues, and parallelization options, bridging theoretical graph algorithms with practical, scalable software engineering.

26.3 Tests & Benchmarks Against Common Datasets

After designing and implementing a **high-performance graph library**, rigorous **testing and benchmarking** are essential to validate correctness, measure performance, and compare against existing solutions. This section provides guidance for creating **unit tests, integration tests**, and **benchmarking pipelines** using real-world and synthetic graph datasets.

26.3.1 Testing Strategy

Testing ensures that all graph algorithms—SSSP, MST, centrality measures—work correctly under varied conditions. A robust strategy includes:

1. **Unit Tests**
   - Verify individual components such as:
     - Vertex and edge insertion/removal
     - Iterator correctness (vertex_iterator, edge_iterator, adjacency_iterator)
     - CSR memory layout consistency
     - Edge weight handling
   - Use **Modern C++ testing frameworks** such as Catch2 or Google Test.

Example: Testing adjacency iterator

```
TEST_CASE("Adjacency Iterator") {
    Graph<int> g(3);
    g.add_edge(0, 1);
    g.add_edge(0, 2);
```

```
    std::vector<int> neighbors;
    for (auto n : g.neighbors(0)) {
        neighbors.push_back(n);
    }

    REQUIRE(neighbors.size() == 2);
    REQUIRE((neighbors[0] == 1 || neighbors[0] == 2));
}
```

2. **Integration Tests**

   - Run full SSSP and MST computations on small graphs with known results.
   - Validate centrality measures against expected outputs.

   Goal: Detect **algorithmic bugs** and **iterator inconsistencies** early.

26.3.2 Benchmarking Strategy

Benchmarking evaluates **performance** on various graph sizes and types, revealing bottlenecks and efficiency gains of the CSR representation.

1. **Datasets**

   Use both **synthetic** and **real-world** graphs:

   - **Synthetic Graphs**:
     - Random sparse/dense graphs
     - Grid graphs (2D/3D meshes)
     - Scale-free networks (Barabási–Albert model)
   - **Real-World Datasets**:

- Social networks (e.g., Facebook, Twitter snapshots)
- Road networks (e.g., OpenStreetMap extracts)
- Citation networks and web graphs (e.g., SNAP datasets)

These datasets allow testing performance under **different sparsity, size, and topology** conditions.

2. **Benchmark Metrics**

   (a) **Execution Time**

   - Measure runtime of SSSP, MST, and centrality algorithms.
   - Use `std::chrono::high_resolution_clock` for high-precision timing.

   (b) **Memory Usage**

   - Measure peak memory consumption for CSR storage vs. adjacency list.
   - Compare edge storage overhead.

   (c) **Scalability**

   - Test increasing graph sizes ($10^3 \rightarrow 10$ vertices).
   - Measure how runtime and memory scale.

   (d) **Cache Efficiency**

   - Use hardware counters (`perf stat` on Linux) to record cache misses and branch mispredictions.
   - Compare row-major CSR access against naive adjacency list traversal.

26.3.3 Example Benchmarking Workflow

Step 1: Prepare Graphs

```
Graph<int> g;
g.load_from_edge_list("dataset/facebook_combined.txt");
```

Step 2: Time SSSP Execution

```
auto start = std::chrono::high_resolution_clock::now();
auto dist = dijkstra(g, 0);
auto end = std::chrono::high_resolution_clock::now();
std::cout << "SSSP runtime: "
          << std::chrono::duration_cast<std::chrono::milliseconds>(end-start).count()
          << " ms\n";
```

Step 3: Repeat for MST

```
auto start_mst = std::chrono::high_resolution_clock::now();
auto mst_parent = prim(g);
auto end_mst = std::chrono::high_resolution_clock::now();
std::cout << "MST runtime: "
          <<
          ↪  std::chrono::duration_cast<std::chrono::milliseconds>(end_mst-start_mst).count(
          << " ms\n";
```

Step 4: Compare with Reference Implementations

- Use `boost::graph` or `NetworkX` (Python) as baseline for correctness and runtime.
- Calculate **speedup factor**:

```
speedup = baseline_time / library_time
```

26.3.4 Automation and Reproducibility

- **Scripted Benchmarks:** Write shell or Python scripts to automate dataset loading, execution, and result collection.
- **Reproducibility:** Fix random seeds for synthetic graphs and random number generators in SSSP tests.
- **Logging:** Record runtime, memory usage, cache metrics, and correctness checks into CSV or JSON files.

26.3.5 Example Observations

- CSR-based adjacency traversal reduces cache misses and often improves runtime by **2–5**× on large sparse graphs compared to linked adjacency lists.
- Dijkstra with `std::priority_queue` performs well on medium-sized graphs; Fibonacci heap implementations yield better asymptotic performance on very dense graphs but are more complex.
- Multi-threaded centrality computation scales linearly with the number of CPU cores for large graphs.

Summary:
Testing and benchmarking are crucial for validating the **correctness, performance, and scalability** of the high-performance graph library. By applying unit and integration tests along with systematic benchmarks on both synthetic and real-world datasets, developers gain confidence in the library's robustness and can **quantify performance improvements** introduced by CSR memory layout and Modern C++ optimizations.

Chapter 27

Project B — Mini Compiler / Interpreter

27.1 Lexing and Parsing with Modern C++ (Recursive Descent, Parser Combinators)

In **Project B — Mini Compiler/Interpreter**, the first step toward building a fully functional language processor is the **front-end**, which consists of **lexical analysis (lexing)** and **syntax analysis (parsing)**. This section discusses approaches using **Modern C++**, focusing on **recursive descent parsers** and **parser combinators**, providing type-safe, flexible, and maintainable implementations.

27.1.1 Lexical Analysis (Lexer)

Lexical analysis converts a raw source code string into a **stream of tokens**, each representing meaningful elements such as keywords, identifiers, literals, and operators.

1. **Lexer Goals**

   - **Tokenization**: Split input into discrete tokens.
   - **Position Tracking**: Track line and column numbers for error reporting.
   - **Error Handling**: Detect invalid characters or malformed tokens.
   - **Efficiency**: Use contiguous memory and minimal heap allocations.

2. **C++ Implementation Strategy**

   - Use `std::string_view` to avoid unnecessary string copies.
   - Represent tokens with `enum class TokenType` and a lightweight `struct Token`.

   Example Token Struct:

```
enum class TokenType { Identifier, Number, Plus, Minus, Star, Slash, LParen,
↪  RParen, End };

struct Token {
    TokenType type;
    std::string_view lexeme;
    int line, column;
};
```

   - Lexer class maintains a **current pointer** and provides `next_token()` API.

   Example Lexer Skeleton:

```
class Lexer {
public:
    explicit Lexer(std::string_view src) : source(src), pos(0), line(1),
    ↪  column(1) {}

    Token next_token() {
        skip_whitespace();
        if (pos >= source.size()) return {TokenType::End, "", line, column};

        char c = source[pos];
        if (std::isdigit(c)) return number();
        if (std::isalpha(c)) return identifier();
        switch(c) {
            case '+': return simple_token(TokenType::Plus);
            case '-': return simple_token(TokenType::Minus);
            // ... other single-character tokens
        }
        throw std::runtime_error("Unknown character at line " +
        ↪  std::to_string(line));
    }

private:
    std::string_view source;
    size_t pos;
    int line, column;

    void skip_whitespace() { while (std::isspace(source[pos])) advance(); }
    void advance() { ++pos; ++column; }
    Token simple_token(TokenType t) { return {t, source.substr(pos++,1), line,
    ↪  column++}; }
};
```

- This design ensures **lightweight tokenization** with minimal copying and memory allocations.

27.1.2 Syntax Analysis (Parser)

Parsing transforms the token stream into a **syntax tree or AST** (Abstract Syntax Tree) according to the **grammar of the language**.

1. **Recursive Descent Parsing**

   - Each grammar rule is implemented as a **function**.
   - Functions recursively call each other following the **grammar hierarchy**.
   - Offers **clarity and maintainability** for small to medium grammars.

   Example: Parsing arithmetic expressions

```
struct Expr { virtual ~Expr() = default; };
struct BinaryExpr : Expr { char op; Expr* lhs; Expr* rhs; };

class Parser {
public:
    explicit Parser(std::vector<Token>& tokens) : tokens(tokens), pos(0) {}

    Expr* parse_expression() { return parse_term(); }

private:
    std::vector<Token>& tokens;
    size_t pos;

    Expr* parse_term() {
        Expr* left = parse_factor();
```

```
    while (match({TokenType::Plus, TokenType::Minus})) {
        char op = current().lexeme[0];
        advance();
        Expr* right = parse_factor();
        left = new BinaryExpr{op, left, right};
    }
    return left;
}

Expr* parse_factor() {
    Expr* left = parse_primary();
    while (match({TokenType::Star, TokenType::Slash})) {
        char op = current().lexeme[0];
        advance();
        Expr* right = parse_primary();
        left = new BinaryExpr{op, left, right};
    }
    return left;
}

Expr* parse_primary() {
    if (match(TokenType::Number)) return new
    ↪  NumberExpr{std::stoi(current().lexeme)};
    if (match(TokenType::LParen)) {
        advance();
        Expr* e = parse_expression();
        expect(TokenType::RParen);
        return e;
    }
    throw std::runtime_error("Unexpected token");
}
```

```
    bool match(TokenType t) { return current().type == t; }
    bool match(std::initializer_list<TokenType> types) {
        for (auto t : types) if (current().type == t) return true;
        return false;
    }
    void advance() { pos++; }
    void expect(TokenType t) { if (!match(t)) throw
    ↪  std::runtime_error("Expected token"); else advance(); }
    Token current() { return tokens[pos]; }
};
```

- **Advantages:**
  - Simple and readable.
  - Easy to debug and extend.
  - Matches Modern C++ style with strong typing and RAII-friendly memory handling.

2. **Parser Combinators**

   For more **functional and flexible parsing**, **parser combinators** provide higher-order constructs to combine small parsers into complex ones.

   - Each parser is a **callable object** returning success/failure with matched tokens.
   - Combines small units like `literal("if")`, `number()`, and `identifier()` into complex expressions.
   - Allows **modular and reusable parsing components**.

   Example Combinator Skeleton

```
template<typename T>
struct ParserResult { T value; bool success; };

auto number_parser = [](Lexer& lex) -> ParserResult<int> {
    Token t = lex.next_token();
    if (t.type == TokenType::Number) return {std::stoi(std::string(t.lexeme)),
    ↪  true};
    return {0, false};
};
```

- Parsers can be combined with functions like `sequence`, `choice`, and `many` to parse **lists, expressions, or statements**.

27.1.3 Modern C++ Features Applied

- **`std::variant` and `std::unique_ptr`** for AST node storage.
- **`std::string_view`** to avoid unnecessary copying of lexemes.
- **Templates and constexpr** to parameterize numeric types and literal handling.
- **RAII and smart pointers** for memory-safe AST trees.
- **Optional parallel lexing** for large input using threads and token chunks (advanced).

27.1.4 Best Practices

1. Keep **lexer and parser modular** for easy testing.
2. Use **iterators or ranges** for token streams to integrate with STL algorithms.

3. Maintain **clear error messages** with line and column info.
4. Use **unit tests** for individual grammar rules and tokens.
5. Separate **AST construction** from parsing logic to enable transformations and optimizations later.

Summary:
This section establishes the foundation for a **mini compiler/interpreter** by combining **lexical analysis** and **syntax analysis** using Modern C++ techniques. Recursive descent parsing provides simplicity and clarity for grammar rules, while parser combinators allow modularity and composability. Together with **lightweight tokenization**, `std::string_view`, and smart pointers, these approaches enable the creation of a **robust, maintainable, and high-performance compiler front-end**.

27.2 AST Transformations, Control-Flow Algorithms, Simple Optimization Passes

Once a **mini compiler/interpreter** has successfully parsed source code into an **Abstract Syntax Tree (AST)**, the next stage is **AST transformations**, **control-flow management**, and **basic optimization passes**. These steps are critical for improving runtime efficiency, preparing code for interpretation or code generation, and enforcing semantic correctness. Modern C++ provides tools for implementing these transformations in a **type-safe, efficient, and maintainable** manner.

27.2.1 AST Representation

The AST is a **tree structure** representing the syntactic structure of the program. Each node corresponds to a **language construct**, such as expressions, statements, or function definitions.

Node Types and Storage

- Use `std::variant` or **polymorphic class hierarchy** for node types:
  - `Expression` (binary, unary, literals, variables)
  - `Statement` (assignment, if, while, return)
  - `Function` (parameters, body, return type)

Example C++ AST Node Hierarchy:

```
struct Expr { virtual ~Expr() = default; };
struct BinaryExpr : Expr { char op; std::unique_ptr<Expr> lhs, rhs; };
struct LiteralExpr : Expr { int value; };
```

```
struct VariableExpr : Expr { std::string name; };

struct Stmt { virtual ~Stmt() = default; };
struct ExprStmt : Stmt { std::unique_ptr<Expr> expr; };
struct IfStmt : Stmt {
    std::unique_ptr<Expr> condition;
    std::unique_ptr<Stmt> then_branch;
    std::unique_ptr<Stmt> else_branch;
};
struct WhileStmt : Stmt { std::unique_ptr<Expr> condition; std::unique_ptr<Stmt>
↪  body; };
```

- **Memory safety** is ensured using **`std::unique_ptr`**, which manages ownership and prevents leaks.
- Traversal is done via **visitor patterns** or **recursive functions**.

27.2.2 AST Transformations

Transformations modify the AST to improve clarity, enforce semantics, or prepare for code generation. Common transformations include:

1. **Constant Folding**
    - Compute expressions with **constant operands** at compile-time.
    - Reduces runtime computation.

    Example:

```
Expr* fold_constants(BinaryExpr* node) {
    if (auto lhs = dynamic_cast<LiteralExpr*>(node->lhs.get())) {
        if (auto rhs = dynamic_cast<LiteralExpr*>(node->rhs.get())) {
            int val = (node->op == '+') ? lhs->value + rhs->value : lhs->value
            ↪  - rhs->value;
            return new LiteralExpr{val};
        }
    }
    return node; // return original if not foldable
}
```

2. **Algebraic Simplifications**
    - Transform `x * 1 → x`, `x + 0 → x`, `x * 0 → 0`.
    - Applied recursively during AST traversal.
3. **Dead Code Elimination**
    - Remove statements or branches that are never executed, e.g., `if (false) { ... }`.
    - Reduces AST size and improves interpreter performance.

27.2.3 Control-Flow Algorithms

Control-flow transformations organize the program into a **structured flow** for interpretation:

1. **CFG Construction**
    - Construct a **Control-Flow Graph (CFG)** from the AST: nodes represent statements or basic blocks; edges represent possible execution paths.

- Each branch (`if`, `while`) becomes a decision node with outgoing edges.
- Loops are represented as **back edges** to previous blocks.

Benefits:

- Facilitates analysis of loops, conditional execution, and termination.
- Enables later optimization passes, e.g., constant propagation along paths.

2. **Dominator Analysis**

- Compute **dominators** of each block to identify blocks that always precede others.
- Useful for optimizations like **common subexpression elimination**.

3. **Simple Data-Flow Analysis**

- Track variable definitions and uses.
- Detect uninitialized variable usage or unreachable code.

27.2.4 Simple Optimization Passes

Optimization passes improve efficiency without altering program semantics.

1. **Inline Small Functions**

- Replace calls to tiny functions with the function body.
- Reduces function call overhead during interpretation.

2. **Loop Unrolling (Optional)**

- For fixed-iteration loops, expand body multiple times.
- Improves interpreter efficiency by reducing branching overhead.

3. **Expression Simplification**

- Apply **constant folding and algebraic simplification** globally across AST.
- Combine multiple arithmetic transformations in a single pass for speed.

Pass Manager Pattern (C++ Example):

```
class ASTOptimizer {
public:
    void run(std::unique_ptr<Stmt>& root) {
        fold_constants(root);
        simplify_expressions(root);
        eliminate_dead_code(root);
    }
private:
    void fold_constants(std::unique_ptr<Stmt>& node) { /* recursive traversal
    ↪  */ }
    void simplify_expressions(std::unique_ptr<Stmt>& node) { /* ... */ }
    void eliminate_dead_code(std::unique_ptr<Stmt>& node) { /* ... */ }
};
```

- Passes are **modular**, enabling users to add or remove transformations as needed.

27.2.5 Modern C++ Techniques Applied

- **`std::unique_ptr` and RAII** for AST memory safety.

- **`std::variant` and `std::visit`** for type-safe node handling.
- **Recursive and iterative traversal** for transformations.
- Optional **template-based numeric literals** for compile-time evaluation in constant folding.
- Clear **visitor or pass-manager pattern** separates concerns: parsing, AST construction, optimization, and interpretation.

Summary:
This section introduces **AST transformations, control-flow algorithms, and simple optimization passes** as the second stage of a mini compiler/interpreter. By constructing a **Control-Flow Graph**, applying **constant folding, dead code elimination, and algebraic simplifications**, and leveraging Modern C++ features, the compiler front-end becomes **robust, maintainable, and efficient**. These steps ensure that the AST is **ready for execution or code generation**, providing a solid foundation for semantic analysis and further optimization passes.

27.3 Exercises — Generate Three-Address Code, Simple Register Allocation

After parsing the source code and performing **AST transformations and basic optimizations**, the next step in **Project B — Mini Compiler/Interpreter** is generating an **intermediate representation (IR)** and managing **register allocation**. These exercises help solidify understanding of **code generation**, **low-level execution planning**, and **resource management**, all crucial for compiler design.

27.3.1 Three-Address Code (TAC) Generation

Three-Address Code (TAC) is a widely used **intermediate representation** in compilers. Each instruction typically has at most **three operands**, which may be variables, constants, or temporaries:

```
x = y + z
t1 = a * b
t2 = t1 + c
```

1. **Goals of TAC**
    - Simplifies **complex expressions** into a linear sequence of simple operations.
    - Facilitates **register allocation** and **optimization passes**.
    - Acts as a bridge between **AST-level constructs** and **low-level machine code** or **bytecode**.
2. **Implementing TAC in Modern C++**
    - Define an enum for **opcode types**:

```
enum class OpCode { Add, Sub, Mul, Div, Load, Store };
```

- Define a structure for TAC instructions:

```
struct TACInstr {
    OpCode op;
    std::string dest;
    std::string lhs;
    std::string rhs; // empty for unary ops or loads/stores
};
```

- Recursive traversal of the AST generates TAC instructions for each **expression node**.

Example: Binary Expression Generation

```
std::vector<TACInstr> tac_instructions;
int temp_counter = 0;

std::string generate_expr(Expr* expr) {
    if (auto bin = dynamic_cast<BinaryExpr*>(expr)) {
        std::string lhs = generate_expr(bin->lhs.get());
        std::string rhs = generate_expr(bin->rhs.get());
        std::string temp = "t" + std::to_string(temp_counter++);
        OpCode op = (bin->op == '+') ? OpCode::Add : OpCode::Sub;
        tac_instructions.push_back({op, temp, lhs, rhs});
        return temp;
    }
    if (auto lit = dynamic_cast<LiteralExpr*>(expr)) {
```

```
        return std::to_string(lit->value);
    }
    if (auto var = dynamic_cast<VariableExpr*>(expr)) {
        return var->name;
    }
    throw std::runtime_error("Unsupported expression type");
}
```

- **Key Advantages:**
  - Each TAC instruction is simple and uniform.
  - Easy to map temporaries to physical registers or stack locations.
  - Facilitates subsequent **optimization passes**, such as **common subexpression elimination**.

27.3.2 Simple Register Allocation

After generating TAC, we must decide **where each temporary or variable resides during execution**: in a CPU register or memory. Even in a small interpreter, simulating **register allocation** improves performance and introduces **compiler design principles**.

1. **Linear Scan Allocation (Simplest Strategy)**
   - Maintain a fixed number of **virtual registers**.
   - Assign temporaries to registers in the order of **first appearance**, reusing freed registers when a temporary goes out of scope.
   - Track **live ranges** of temporaries to avoid conflicts.

Example Linear Scan Allocation:

```
std::unordered_map<std::string, int> reg_map;
std::vector<bool> reg_free(8, true); // assume 8 registers

int allocate_register(const std::string& temp) {
    for (int i = 0; i < reg_free.size(); ++i) {
        if (reg_free[i]) {
            reg_free[i] = false;
            reg_map[temp] = i;
            return i;
        }
    }
    throw std::runtime_error("No registers available, spill to memory");
}

void free_register(const std::string& temp) {
    int reg = reg_map[temp];
    reg_free[reg] = true;
    reg_map.erase(temp);
}
```

2. **Live Range Tracking**

- Determine when each temporary **starts and ends usage** in TAC instructions.
- Free registers as soon as a temporary is no longer needed.
- Enables **reuse of limited registers**, avoiding unnecessary memory access.

27.3.3 Combined Exercise Workflow

1. **Generate TAC:**
    - Traverse the AST of expressions and statements.
    - Produce sequential three-address code with temporaries.
2. **Perform Register Allocation:**
    - Assign each temporary to a virtual register.
    - Reuse registers when live ranges end.
3. **Optional Enhancements:**
    - Implement **spilling**: move temporaries to a stack if registers are full.
    - Track **basic blocks** for more advanced allocation strategies.

Example TAC Output After Allocation:

```
t0 = a + b      // t0 in R0
t1 = t0 * c     // t1 in R1
x  = t1         // store R1 to memory location of x
```

27.3.4 Learning Outcomes

- Understand **TAC as a bridge** between AST and executable code.
- Practice **register allocation** and **resource management** in a compiler context.
- Appreciate the interplay of **algorithmic design** (live ranges, linear scan) and **low-level execution concerns**.

- Develop a strong foundation for **further optimizations** and eventually **machine code generation or bytecode interpreters**.

Summary:
This section guides readers through exercises that generate **three-address code** from ASTs and implement **simple register allocation** strategies. By combining **TAC generation** with **linear scan allocation**, students learn **practical compiler construction techniques** while applying Modern C++ features like **`std::unique_ptr`, `std::unordered_map`, and type-safe enums**. These exercises serve as a bridge between **semantic analysis** and the **execution/runtime phase**, reinforcing key principles of compiler design.

Chapter 28

Project C — Algorithmic Trading Backtester (example of time-series algorithms)

28.1 Streaming Data Algorithms, Sliding Windows, Online Learning Sketches

In **Project C — Algorithmic Trading Backtester**, handling **time-series financial data** efficiently is crucial. Real-world financial data streams are **continuous, high-frequency, and potentially unbounded**, requiring specialized algorithms to compute metrics, detect patterns, and update strategies **without storing the entire history**. Modern C++ provides the tools to implement these algorithms **efficiently, safely, and in a parallelizable manner**.

28.1.1 Streaming Data Algorithms

Streaming algorithms process **data points incrementally**, updating results as new points arrive. They are particularly important in algorithmic trading for computing:

- **Moving averages** (simple, exponential)
- **Volatility measures**
- **Cumulative returns**

Key Design Principles:

- **Single-pass computation:** Avoid multiple traversals over the dataset.
- **Bounded memory:** Only store the data needed for computations (e.g., window size).
- **Incremental updates:** Update statistics efficiently with each new tick or bar.

C++ Implementation Example: Exponential Moving Average (EMA)

```
class EMA {
    double alpha;
    double value;
    bool initialized = false;

public:
    explicit EMA(double alpha_) : alpha(alpha_), value(0.0) {}

    double update(double price) {
        if (!initialized) {
            value = price;
```

```
            initialized = true;
        } else {
            value = alpha * price + (1 - alpha) * value;
        }
        return value;
    }
};
```

- **Incremental update** ensures **O(1) time per update**.
- No storage of historical prices is needed, making it memory-efficient.

28.1.2 Sliding Window Techniques

Many algorithms require statistics over a **fixed-size window**, such as:

- **Simple moving average (SMA)**
- **Rolling variance**
- **Max/min values for technical indicators**

Circular Buffer Implementation

C++ offers tools like `std::deque` or `std::vector` with a **circular buffer logic** for efficient windowing.

```
class SMA {
    std::deque<double> window;
    size_t max_size;
    double sum = 0.0;
```

```
public:
    explicit SMA(size_t size) : max_size(size) {}

    double update(double price) {
        window.push_back(price);
        sum += price;
        if (window.size() > max_size) {
            sum -= window.front();
            window.pop_front();
        }
        return sum / window.size();
    }
};
```

- Efficient **O(1) update**.
- Maintains only the last `max_size` elements in memory.
- Supports **real-time trading simulations** without storing the entire history.

28.1.3 Online Learning Sketches

Beyond standard statistical indicators, **online learning sketches** can summarize large-scale or unbounded streams:

Count-Min Sketch

- Tracks approximate **frequency of events** in streaming data.
- Memory-efficient and suitable for high-frequency trade tick counting.
- Useful for **detecting unusual trading patterns** or spikes.

C++ Implementation Highlights:

- Use a **2D array** with multiple hash functions.
- Increment counts per observed value.
- Query minimum count as approximate frequency.

```
class CountMinSketch {
    std::vector<std::vector<int>> table;
    size_t depth, width;

public:
    CountMinSketch(size_t d, size_t w) : depth(d), width(w), table(d,
    ↪  std::vector<int>(w, 0)) {}

    void update(int x) {
        for (size_t i = 0; i < depth; ++i) {
            size_t idx = hash_combine(i, x) % width;
            table[i][idx]++;
        }
    }

    int query(int x) const {
        int min_val = INT_MAX;
        for (size_t i = 0; i < depth; ++i) {
            size_t idx = hash_combine(i, x) % width;
            min_val = std::min(min_val, table[i][idx]);
        }
        return min_val;
    }
};
```

- Provides **constant memory footprint** independent of stream size.
- Errors are **bounded probabilistically**, often sufficient for algorithmic decision-making.

28.1.4 Integrating Streaming Algorithms into the Backtester

Design Considerations:

- **Iterator-based APIs:** Accept `std::vector`, `std::deque`, or streaming iterators to process tick data generically.
- **Real-time updates:** Each new price triggers recomputation of relevant indicators.
- **Composable statistics:** EMA, SMA, and volatility can be **combined in a single pipeline**.
- **Thread safety:** Use `std::atomic` or locks if computing indicators in **parallel over multiple symbols**.

Example Usage in Backtester:

```
EMA ema(0.1);
SMA sma(20);

for (double price : price_stream) {
    double current_ema = ema.update(price);
    double current_sma = sma.update(price);
    // Use for strategy signals
}
```

28.1.5 Summary

- **Streaming data algorithms** enable **real-time computation** on unbounded datasets with **bounded memory**.
- **Sliding windows** allow efficient computation of rolling statistics critical for trading strategies.
- **Online learning sketches**, like Count-Min Sketch, allow approximate analytics on high-frequency streams.
- Modern C++ idioms—**RAII, iterators, STL containers, and templates**—facilitate **robust, reusable, and efficient implementations**.

By mastering these techniques, readers can implement **high-performance backtesting engines** capable of simulating and analyzing real-world trading strategies with minimal memory overhead.

28.2 Backtesting Engine Design and Performance Constraints

Building a **robust backtesting engine** is a central component of **Project C — Algorithmic Trading Backtester**. The engine must **simulate historical or streaming market data**, compute indicators efficiently, and evaluate trading strategies while respecting **performance constraints**, memory usage, and reproducibility. Modern C++ provides the tools to implement such engines with **high performance, safety, and modularity**.

28.2.1 Core Design Goals

The backtesting engine should satisfy the following design principles:

1. **Separation of Concerns**
    - **Data ingestion layer:** Reads historical or live price feeds.
    - **Indicator computation layer:** Calculates rolling and streaming indicators (EMA, SMA, volatility).
    - **Strategy evaluation layer:** Executes user-defined strategies and decision rules.
    - **Execution simulation layer:** Applies trades, updates positions, and tracks P&L.
2. **High Performance**
    - Use **iterator-based processing** and **streaming algorithms** to minimize memory overhead.

- Prefer **cache-friendly data structures** (e.g., `std::vector` for sequential access).
- Enable **parallel computation** for multiple symbols using `std::thread`, thread pools, or `std::execution::par`.

3. **Modularity and Extensibility**

- Each component should have **clear APIs** and be **replaceable**.
- New indicators, strategies, or risk models can be added without rewriting the core engine.

4. **Reproducibility**

- Use **seeded random number generators** for stochastic simulations.
- Maintain deterministic order of operations for backtesting across runs.

28.2.2 Engine Architecture

A typical backtesting engine can be represented in **layered modules**:

1. **Data Layer**

- Handles **streaming or historical tick data**.
- Implements **sliding windows** for rolling metrics.
- Uses **memory-efficient containers**: `std::deque` for fixed-size windows, `std::vector` for time series.

```
struct PriceBar {
    double open, high, low, close;
    std::chrono::system_clock::time_point timestamp;
};
std::vector<PriceBar> historical_data;
```

2. **Indicator Layer**

- Computes **technical indicators** incrementally.
- Maintains minimal **state variables** to update efficiently.
- Supports **composition** for multiple indicators in a pipeline.

```
EMA ema_short(0.1);
EMA ema_long(0.05);
for (auto& bar : historical_data) {
    double ema_s = ema_short.update(bar.close);
    double ema_l = ema_long.update(bar.close);
    // feed signals into strategy
}
```

3. **Strategy Layer**

- Accepts **indicators and market data**.
- Executes **signal generation** (buy, sell, hold).
- Can be **templated** or abstracted using `std::function` for flexibility.

```
std::function<void(const PriceBar&, double, double)> strategy =
[](const PriceBar& bar, double ema_s, double ema_l) {
    if (ema_s > ema_l) { /* signal buy */ }
    else { /* signal sell */ }
};
```

4. **Execution Layer**

   - Simulates **order fills**, **position management**, and **transaction costs**.
   - Tracks **cumulative profit/loss**, maximum drawdown, and portfolio statistics.

28.2.3 Performance Constraints

Efficiency is paramount in high-frequency or multi-symbol backtesting. Key constraints include:

1. **Time Complexity**

   - Indicators and strategies should update in **O(1) per tick** when possible.
   - Avoid recomputation of full historical series for each tick.

2. **Memory Usage**

   - Maintain only **necessary historical data**, e.g., windowed values for rolling indicators.
   - Use **move semantics**, `std::unique_ptr`, and **cache-aligned structures**.

3. **Parallelism**

- **Symbol-level parallelism:** Compute indicators for different symbols concurrently.
- **Strategy-level parallelism:** Run multiple strategies in parallel on the same data.

4. **Profiling and Bottleneck Analysis**

- Use **profilers** (`perf`, `gprof`) to detect slow paths.
- Identify memory allocation hotspots or excessive copying.
- Optimize **inner loops** in indicator calculations.

28.2.4 Modern C++ Techniques Applied

- **RAII and smart pointers** for safe memory management.
- **`std::vector`, `std::deque`, `std::array`** for cache-friendly access.
- **`std::execution` policies** for parallel algorithm execution.
- **Templates and `std::function`** for flexible strategy and indicator integration.
- **Move semantics** to reduce copying of PriceBar or indicator objects.

28.2.5 Example Engine Loop

```
for (auto& bar : historical_data) {
    // Update indicators
    double ema_s = ema_short.update(bar.close);
    double ema_l = ema_long.update(bar.close);
```

```
    // Evaluate strategy
    strategy(bar, ema_s, ema_l);

    // Simulate execution and update P&L
    execution_engine.process(bar);
}
```

- **Incremental updates** keep the engine **O(n)** for n ticks.
- Supports **multi-symbol processing** by replicating the above loop per symbol with optional parallelization.

28.2.6 Summary

The backtesting engine is designed for **efficiency, modularity, and scalability**:

- Handles **streaming and historical market data** using incremental, memory-efficient algorithms.
- Computes **technical indicators** and evaluates strategies in **real-time or simulated-time**.
- Respects **performance constraints** through **cache-aware data structures, O(1) updates, and optional parallelism**.
- Provides a **framework for experimentation** with algorithmic trading strategies in a safe and reproducible C++ environment.

This design ensures that readers can **test, refine, and analyze strategies** efficiently, preparing them for **production-ready trading engines** or further extensions like **risk modeling and portfolio optimization**.

28.3 Exercises — Implement Moving Average Crossover Strategy, Evaluate Latency

The final step in **Project C — Algorithmic Trading Backtester** is a hands-on exercise that combines **streaming data processing, indicator computation, and strategy evaluation**. The **moving average crossover strategy** is a classic algorithmic trading approach, and this exercise also emphasizes **performance measurement**, including **latency analysis**.

28.3.1 Moving Average Crossover Strategy

The strategy is based on two **exponential moving averages (EMAs)** with different window lengths:

- **Short-term EMA (`ema_short`)**: reacts quickly to recent price changes.
- **Long-term EMA (`ema_long`)**: smooths out fluctuations and represents the trend.

Trading logic:

1. **Buy signal**: short-term EMA crosses **above** long-term EMA.
2. **Sell signal**: short-term EMA crosses **below** long-term EMA.

C++ Implementation:

```
class EMACrossoverStrategy {
    EMA ema_short;
    EMA ema_long;
    bool position_open = false; // tracks if currently holding a position
```

```
public:
    EMACrossoverStrategy(double alpha_short, double alpha_long)
        : ema_short(alpha_short), ema_long(alpha_long) {}

    void update(double price) {
        double short_val = ema_short.update(price);
        double long_val  = ema_long.update(price);

        if (!position_open && short_val > long_val) {
            position_open = true;
            execute_order("BUY", price);
        } else if (position_open && short_val < long_val) {
            position_open = false;
            execute_order("SELL", price);
        }
    }

    void execute_order(const std::string& side, double price) {
        // Simulate order execution
        std::cout << side << " at " << price << "\n";
    }
};
```

- **Incremental EMA updates** allow O(1) per tick computation.
- The `position_open` flag prevents repeated orders while the trend continues.

28.3.2 Integrating with the Backtesting Engine

The strategy is integrated into the **backtesting loop**:

```
EMACrossoverStrategy strategy(0.1, 0.05);

auto start = std::chrono::high_resolution_clock::now();
for (const auto& bar : price_stream) {
    strategy.update(bar.close);
}
auto end = std::chrono::high_resolution_clock::now();

std::chrono::duration<double, std::milli> latency = end - start;
std::cout << "Total processing time: " << latency.count() << " ms\n";
```

- `price_stream` can be **historical data** or simulated ticks.
- **High-resolution clock** measures **latency per run**, critical for high-frequency backtesting.

28.3.3 Evaluating Latency

Latency evaluation ensures the **strategy is efficient enough for real-time or large-scale simulations**.

1. **Metrics**
    - **Average latency per tick**: total processing time divided by number of ticks.
    - **Maximum latency spike**: identifies potential bottlenecks in updates.
    - **Memory overhead**: ensure fixed-size sliding windows or EMAs do not grow unbounded.
2. **Optimization Tips**

(a) **Minimize memory allocations** by pre-allocating buffers.

(b) **Use cache-friendly data structures**, e.g., `std::vector` over `std::list`.

(c) **Leverage parallelism** if testing multiple symbols simultaneously.

(d) **Profile hot loops** with tools like `perf` or `gprof`.

Example latency calculation:

```
double avg_latency_ms = latency.count() / price_stream.size();
std::cout << "Average latency per tick: " << avg_latency_ms << " ms\n";
```

28.3.4 Advanced Extensions

After mastering the basic moving average crossover, students can experiment with:

- **Multi-symbol strategies**: run multiple EMA crossovers concurrently.
- **Alternative indicators**: Bollinger Bands, RSI, MACD.
- **Online parameter tuning**: adjust `alpha_short` and `alpha_long` dynamically based on volatility.
- **Transaction cost simulation**: account for slippage and commission in order execution.

28.3.5 Learning Outcomes

- Practical implementation of a **classic algorithmic trading strategy**.
- Understanding of **incremental computation and streaming data handling**.
- Measuring and optimizing **latency and memory usage** in C++.

- Gaining experience in **integrating multiple modules**: data feed, indicator computation, strategy logic, and execution simulation.

Summary:
This exercise demonstrates the **full pipeline of algorithmic backtesting**: streaming data ingestion, real-time indicator computation, strategy execution, and performance evaluation. By implementing the **moving average crossover strategy** and evaluating latency, readers gain hands-on experience in **high-performance, real-time algorithmic systems** using Modern C++ idioms.

Part VIII

Testing, Reproducibility & Research Practices

Chapter 29

Testing Algorithm Correctness in C++

29.1 Property-Based Testing, Fuzzing Inputs, Determinism in Tests

In **Chapter 1 — Testing Algorithm Correctness in C++**, **property-based testing** and **fuzzing** are essential techniques to ensure that algorithms behave correctly across **a wide range of inputs**, including edge cases. Modern C++ provides the tools to implement these testing paradigms effectively, allowing developers to write **robust, deterministic, and reproducible tests**.

29.1.1 Property-Based Testing

Property-based testing (PBT) focuses on **general properties of an algorithm** rather than specific input/output pairs. Instead of checking for a single expected result,

the test verifies that certain invariants hold for **all valid inputs**.

Examples of Properties:

- **Sorting algorithm:** Output must be non-decreasing and contain the same elements as the input.
- **Graph traversal:** Every reachable node is visited exactly once.
- **Mathematical functions:** Factorial of `n` must be `n!` and always positive.

C++ Implementation Example using Templates:

```
#include <vector>
#include <algorithm>
#include <cassert>
#include <random>

template<typename Func>
void test_sorting(Func sort_fn, size_t num_tests = 100) {
    std::mt19937 rng(42); // deterministic seed
    std::uniform_int_distribution<int> dist(-1000, 1000);

    for (size_t t = 0; t < num_tests; ++t) {
        std::vector<int> data(100);
        for (auto& x : data) x = dist(rng);

        auto original = data;
        sort_fn(data);

        // Property 1: Sorted output
        assert(std::is_sorted(data.begin(), data.end()));

        // Property 2: Same elements
```

```
        std::sort(original.begin(), original.end());
        assert(data == original);
    }
}
```

- **Determinism:** Using a fixed random seed ensures reproducible tests.
- **Generality:** Tests a **wide range of inputs automatically**.
- **Extensibility:** Can be adapted for trees, graphs, and other data structures.

29.1.2 Fuzzing Inputs

Fuzzing is the automated generation of **randomized, edge-case inputs** to detect bugs, crashes, or unexpected behavior.

- Particularly useful for algorithms with **complex invariants** (e.g., graph algorithms, parsing).
- Helps uncover **integer overflows, out-of-bounds access, and logic errors**.

C++ Example: Fuzzing a Graph Algorithm

```
#include <vector>
#include <random>
#include <cassert>

void fuzz_bfs() {
    std::mt19937 rng(123);
    std::uniform_int_distribution<int> nodes_dist(1, 50);
```

```
    int n = nodes_dist(rng);
    std::vector<std::vector<int>> adj(n);

    std::uniform_int_distribution<int> edge_dist(0, n-1);
    for (int i = 0; i < n * 2; ++i) {
        int u = edge_dist(rng);
        int v = edge_dist(rng);
        adj[u].push_back(v);
    }

    std::vector<bool> visited(n, false);
    std::vector<int> queue = {0};
    visited[0] = true;
    while (!queue.empty()) {
        int node = queue.back(); queue.pop_back();
        for (int neigh : adj[node]) {
            if (!visited[neigh]) {
                visited[neigh] = true;
                queue.push_back(neigh);
            }
        }
    }

    assert(std::count(visited.begin(), visited.end(), true) <= n);
}
```

- Automatically explores **edge cases** that a human tester might overlook.
- Works well with **property-based assertions** to detect subtle errors.

29.1.3 Determinism in Tests

Deterministic tests are crucial for **reproducibility**, especially in research or high-stakes systems like financial simulations or concurrent algorithms.

Best Practices in C++:

1. **Seed all RNGs:** Use fixed seeds (`std::mt19937 rng(seed)`) to reproduce random sequences.
2. **Isolate side effects:** Avoid global state that may change between runs.
3. **Log inputs and outputs:** Store failing test cases for debugging and regression testing.
4. **Encapsulate randomness:** Pass RNGs explicitly to functions instead of using global generators.

```
void deterministic_test(std::mt19937& rng) {
    std::uniform_int_distribution<int> dist(0, 100);
    int val = dist(rng); // reproducible
    assert(val >= 0 && val <= 100);
}
```

29.1.4 Advantages of Property-Based Testing and Fuzzing

- **Covers more cases** than manually written unit tests.
- **Detects subtle algorithmic bugs** in edge cases.
- **Improves confidence** in correctness of highly optimized or parallel code.
- **Supports research reproducibility**, allowing other developers or reviewers to reproduce results exactly.

29.1.5 Summary

- **Property-based testing** focuses on **algorithmic invariants**, ensuring correctness across a wide input space.
- **Fuzzing** systematically generates **randomized or extreme inputs** to detect crashes and edge-case failures.
- **Deterministic setups** with fixed RNG seeds ensure **reproducible, verifiable testing**.
- Modern C++ idioms such as **templates, STL algorithms, and `std::mt19937`** make these techniques both **efficient and elegant**, crucial for **robust algorithm development**.

By combining these methods, readers can **verify, debug, and validate their C++ algorithms** with confidence, preparing them for both research-grade code and high-performance production systems.

29.2 Using GoogleTest / QuickCheck-Style Libraries, CI Integration

Ensuring the correctness of C++ algorithms at scale requires **structured testing frameworks** and **continuous integration (CI)** pipelines. In **Chapter 1 — Testing Algorithm Correctness in C++**, this section focuses on leveraging **GoogleTest** for unit and integration testing, **QuickCheck-style property-based testing libraries** for algorithmic invariants, and integrating them into **modern CI workflows**.

29.2.1 GoogleTest for Unit Testing

GoogleTest (GTest) is a widely-used C++ testing framework that enables:

- Writing **unit tests** for individual functions or classes.
- Structuring tests into **test suites** for modular organization.
- Supporting **parameterized tests** for systematic coverage of multiple inputs.
- Integrating **assertions and fatal/non-fatal checks**.

Example: Testing a Sorting Algorithm

```
#include <gtest/gtest.h>
#include <algorithm>
#include <vector>

void my_sort(std::vector<int>& v) {
    std::sort(v.begin(), v.end());
}
```

```
TEST(SortingTest, HandlesRandomData) {
    std::vector<int> data = {5, 3, 2, 8, 1};
    my_sort(data);
    ASSERT_TRUE(std::is_sorted(data.begin(), data.end()));
}

TEST(SortingTest, HandlesEmptyVector) {
    std::vector<int> data;
    my_sort(data);
    ASSERT_TRUE(data.empty());
}
```

- **ASSERT_TRUE** stops the test immediately if the condition fails, ensuring clear failure reports.
- Tests can be grouped into **suites** (`SortingTest`), simplifying organization for complex projects.

29.2.2 QuickCheck-Style Property-Based Testing

QuickCheck-style libraries, such as **RapidCheck** or **Catch2 with property testing extensions**, allow testing **properties over automatically generated random inputs**.

Example: Property Test for a Sorting Function

```
#include <rapidcheck.h>
#include <algorithm>
#include <vector>

void my_sort(std::vector<int>& v) {
    std::sort(v.begin(), v.end());
```

```
}

int main() {
    rc::check("Sorted vector preserves elements and is non-decreasing",
        [](const std::vector<int>& vec) {
            std::vector<int> sorted = vec;
            my_sort(sorted);

            RC_ASSERT(std::is_sorted(sorted.begin(), sorted.end()));
            auto copy = vec;
            std::sort(copy.begin(), copy.end());
            RC_ASSERT(sorted == copy);
        });
}
```

- **Automatically generates input vectors** of varying sizes and values.
- **Verifies general properties**: sorted order and element preservation.
- **Deterministic reproducibility** is achieved by **fixing the random seed** when needed.

29.2.3 Continuous Integration (CI) Integration

Automating testing with CI ensures **tests are run on every commit**, catching regressions early. Modern C++ projects typically integrate **GitHub Actions, GitLab CI, or Jenkins**:

1. **Install dependencies** (GoogleTest, RapidCheck) using CMake or package managers:

```
# GitHub Actions example
jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout!@!v3
      - name: Setup CMake
        uses: lukka/get-cmake!@!v4
      - name: Build
        run: cmake -S . -B build && cmake --build build
      - name: Run tests
        run: ctest --test-dir build --output-on-failure
```

1. **Automated Test Reporting**
    - CI systems collect **pass/fail reports** for each test suite.
    - Property-based tests log **failed input sequences**, enabling debugging and reproducibility.
2. **Benefits of CI Integration**
    - Enforces **regression safety** across commits.
    - Encourages **consistent testing practices** among team members.
    - Ensures **cross-platform correctness**, since tests can run on Linux, macOS, and Windows agents.

29.2.4 Combining Unit Tests and Property-Based Tests

A robust strategy for C++ algorithm correctness combines:

- **Unit tests** (GTest) for deterministic and small-scope correctness.
- **Property-based tests** (RapidCheck/QuickCheck style) for generalized invariants and edge cases.
- **CI pipelines** for continuous verification and reproducibility.

Example Integration Workflow:

1. Write deterministic **unit tests** for core functions.
2. Write **property-based tests** for high-level invariants.
3. Configure **CI** to build the project and run all tests automatically.
4. Collect **performance metrics** and **coverage reports** to maintain code quality.

29.2.5 Summary

- **GoogleTest** provides structured **unit and integration testing**, with clear assertions and test suite management.
- **QuickCheck-style property testing** automatically verifies algorithmic invariants across randomized inputs, complementing traditional tests.
- **CI integration** ensures **continuous correctness, cross-platform reproducibility, and early bug detection**.
- Modern C++ idioms, including **templates, STL containers, and deterministic RNGs**, make these testing strategies both **efficient and maintainable**.

By combining these approaches, developers can **build high-assurance C++ algorithms**, suitable for research, production, or critical applications requiring **deterministic and reproducible results**.

Chapter 30

Reproducible Experiments & Data Sets

30.1 Dataset Management, Synthetic Data Generators (C++), Seeding, and Reporting Standards

In **Chapter 2 — Reproducible Experiments & Data Sets**, effective experimentation in C++ relies on **careful management of datasets**, the use of **synthetic data generators**, and **standardized reporting practices**. Proper dataset handling ensures that experiments are **repeatable, verifiable, and comparable**.

30.1.1 Dataset Management

A well-organized dataset strategy is crucial for **algorithm evaluation and benchmarking**. Key principles include:

- **Separation of raw and processed data:** Keep original datasets immutable to allow reproducible transformations.
- **Versioning datasets:** Tag datasets with version numbers to track changes and maintain experiment history.
- **Metadata documentation:** Include information about dataset size, distribution, range of values, and any preprocessing steps.
- **Efficient storage in C++:** Use STL containers (`std::vector`, `std::array`) for in-memory experiments, or memory-mapped files (`mmap`) for very large datasets.

Example: Simple Dataset Loader

```
#include <vector>
#include <fstream>
#include <sstream>
#include <string>
#include <iostream>

std::vector<int> load_dataset(const std::string& filename) {
    std::vector<int> data;
    std::ifstream file(filename);
    std::string line;
    while (std::getline(file, line)) {
        std::istringstream iss(line);
        int value;
        while (iss >> value) {
```

```
            data.push_back(value);
        }
    }
    return data;
}
```

- Ensures **consistent and reproducible loading** across experiments.
- Provides a foundation for **synthetic data generation and benchmarking**.

30.1.2 Synthetic Data Generators in C++

Synthetic datasets are essential for:

- Testing algorithms on **controlled distributions**.
- Evaluating performance under **edge cases or stress scenarios**.
- Scaling experiments beyond the limits of real-world datasets.

Using `<random>` for Deterministic Data Generation:

```
#include <random>
#include <vector>
#include <iostream>

std::vector<int> generate_uniform_data(size_t n, int min_val, int max_val, unsigned
↪  seed = 42) {
    std::mt19937 rng(seed); // deterministic seed
    std::uniform_int_distribution<int> dist(min_val, max_val);
    std::vector<int> data(n);
    for (auto& x : data) x = dist(rng);
```

```
    return data;
}

int main() {
    auto data = generate_uniform_data(100, 0, 1000);
    for (auto x : data) std::cout << x << " ";
}
```

- **Deterministic seeding** ensures reproducibility.
- Supports **different distributions**: uniform, normal, binomial, etc.
- Easily integrated into **unit tests, property-based tests, or benchmarking scripts**.

30.1.3 Seeding and Determinism

Deterministic experiments require **careful control of random number generation**:

- Always **initialize RNGs with fixed seeds** for reproducibility.
- For parallel algorithms, consider **thread-local RNGs** to avoid sequence collisions.
- Log the seed used for each experiment to **enable exact replication**.

Example: Parallel RNG with Thread Safety

```
#include <random>
#include <thread>
#include <vector>
#include <iostream>
```

```
void generate_thread_data(int seed, std::vector<int>& out) {
    std::mt19937 rng(seed);
    std::uniform_int_distribution<int> dist(0, 100);
    for (auto& x : out) x = dist(rng);
}

int main() {
    std::vector<int> data(100);
    std::thread t(generate_thread_data, 1234, std::ref(data));
    t.join();
}
```

- Guarantees **reproducible results** even in concurrent environments.
- Critical for experiments involving **randomized algorithms or Monte Carlo simulations**.

30.1.4 Reporting Standards

To support reproducibility and transparency:

1. **Log dataset details:** Include name, size, version, and source.
2. **Document preprocessing steps:** Normalization, filtering, or feature transformations.
3. **Store random seeds and parameters:** Include seeds for RNGs, number of iterations, and hyperparameters.
4. **Automate experiment reports:** Generate logs, summaries, or CSV files for result tracking.

5. **Provide metadata for benchmarks:** Include runtime, memory usage, and environment details (OS, compiler, flags).

Example: Basic Experiment Logging

```
#include <fstream>
#include <iostream>

void log_experiment(const std::string& dataset, unsigned seed, double runtime) {
    std::ofstream log("experiment_log.csv", std::ios::app);
    log << dataset << "," << seed << "," << runtime << "\n";
}
```

- Facilitates **reproducibility** and **peer verification**.
- Enables **automated post-processing** and comparison across multiple experiments.

30.1.5 Summary

- Effective **dataset management** ensures experiments are consistent and reproducible.
- **Synthetic data generators** provide controlled and diverse inputs for robust testing.
- **Seeding and deterministic RNGs** guarantee repeatable results, even in parallel or stochastic settings.
- **Standardized reporting** supports transparency, traceability, and reproducibility of experiments.

By combining these principles, C++ researchers and developers can maintain **high-quality, reproducible experiments**, allowing reliable comparison, validation, and publication of results.

30.2 Publishing Code and Experiments — Packaging with CMake, Docker, and Minimal Reproducibility Checklist

In **Chapter 2 — Reproducible Experiments & Data Sets**, ensuring that experiments are not only reproducible locally but also shareable with collaborators or reviewers requires **structured packaging, containerization, and documentation**. This section focuses on **CMake-based project packaging**, **Docker containers**, and a **minimal reproducibility checklist** for C++ experimental code.

30.2.1 Packaging C++ Experiments with CMake

CMake provides a **cross-platform build system** that simplifies compilation, dependency management, and installation of experimental projects. Proper packaging ensures others can **build and run your experiments** consistently.
Key practices:

- **Organize source code and headers** into `src/` and `include/` directories.
- Use **modern CMake** to declare dependencies, target properties, and testing.
- Provide **options for dataset paths, build types, and compiler flags**.

Example: Basic CMake Setup for an Experiment

```
cmake_minimum_required(VERSION 3.22)
project(AlgorithmExperiments VERSION 1.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 23)
```

```
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Include directories
include_directories(include)

# Source files
file(GLOB SOURCES src/*.cpp)

# Executable target
add_executable(experiment ${SOURCES})

# Link with GoogleTest for testing
find_package(GTest REQUIRED)
target_link_libraries(experiment PRIVATE GTest::gtest_main)

enable_testing()
add_test(NAME RunExperiment COMMAND experiment)
```

- Allows **cross-platform builds** with a single command: `cmake -S . -B build && cmake --build build`.
- Supports **integration with CI/CD pipelines** for automated testing.

30.2.2 Containerization with Docker

Docker ensures that experiments run **identically on any system**, encapsulating the operating system, compilers, libraries, and datasets.

Basic Docker Workflow:

1. **Create a Dockerfile** specifying the environment:

```
FROM ubuntu:24.04

# Install C++ toolchain
RUN apt-get update && apt-get install -y build-essential cmake git

# Copy project files
COPY . /workspace
WORKDIR /workspace

# Build the project
RUN cmake -S . -B build && cmake --build build

# Run experiments by default
CMD ["./build/experiment"]
```

1. **Build and run the container**:

```
docker build -t cpp_experiment .
docker run --rm cpp_experiment
```

- Guarantees **reproducibility across OS versions and compiler setups**.
- Encapsulates dependencies such as **libraries, data files, and runtime settings**.
- Supports **scalable deployment**, e.g., running multiple experiments in parallel containers.

30.2.3 Minimal Reproducibility Checklist

To make experiments reproducible and shareable, follow a **structured checklist**:

1. **Code and Build Environment**
    - Provide a complete **CMakeLists.txt** or build script.
    - Include compiler version, flags, and third-party dependencies.
2. **Data and Randomness**
    - Provide dataset files or a **synthetic data generator** with fixed seeds.
    - Document preprocessing steps and transformations.
3. **Execution Instructions**
    - Include a **README** with step-by-step instructions for building and running experiments.
    - Provide **parameter defaults** and example command lines.
4. **Testing**
    - Include **unit tests** (GoogleTest) for critical functions.
    - Include **property-based tests** (RapidCheck or similar) for algorithm invariants.
5. **Containerization (Optional but Recommended)**
    - Provide a **Dockerfile** or container image for exact reproducibility.
    - Ensure all dependencies and tools are version-pinned.

6. **Logging and Reporting**
    - Log **random seeds, runtime parameters, and environment information**.
    - Save experiment outputs in a **structured format** (CSV, JSON).

30.2.4 Best Practices for Publishing

- **Version control**: Use Git with tags corresponding to published experiments.
- **Archival**: Consider hosting the Docker image and code on platforms like **Docker Hub, GitHub, or Zenodo**.
- **Documentation**: Include a detailed **README, dataset descriptions, and experiment scripts**.
- **Parameterization**: Expose configuration via **command-line arguments or configuration files**, ensuring reproducibility without code modification.

30.2.5 Summary

- **CMake packaging** ensures cross-platform builds and dependency management.
- **Docker containers** encapsulate the environment, guaranteeing reproducibility across systems.
- A **minimal reproducibility checklist** covers code, data, testing, execution instructions, and logging.
- Combining these practices makes experiments **reproducible, shareable, and verifiable**, aligning with modern research and industrial standards.

Appendices

Appendix A – C++ Cheat Sheet for Algorithm Developers

This appendix serves as a compact reference for C++ algorithm developers, focusing on **STL containers, iterators, and common algorithm idioms**. It is intended to provide quick access to frequently used constructs and best practices for writing modern, high-performance, and maintainable code.

A.1 STL Containers

C++ provides a rich set of containers optimized for different scenarios. Choosing the right container is crucial for algorithm performance and correctness.

Sequence Containers

Container	Characteristics	Typical Use Cases
`std::vector<T>`	Dynamic array, contiguous memory, fast random access, amortized O(1) push_back	Storing elements with frequent access by index, efficient iteration, use in numeric algorithms
`std::deque<T>`	Double-ended queue, allows fast insertion/removal at both ends	Task scheduling, algorithms requiring both push_front and push_back
`std::list<T>`	Doubly-linked list, fast insertion/removal anywhere, no random access	Frequent insert/remove operations, maintaining stable iterators
`std::forward_list<T`	Singly-linked list, less memory than `list`, forward-only traversal	Low-memory list operations, functional-style processing
`std::array<T,N>`	Fixed-size array, contiguous memory	Compile-time fixed sequences, low overhead alternatives to `vector`
`std::string`	Sequence of characters with dynamic resizing	Text processing, string algorithms

Associative Containers

Container	Characteristics	Typical Use Cases
`std::set<T>`	Ordered, unique elements, typically implemented as red-black tree	Efficient ordered storage, uniqueness enforcement
`std::map<K,V>`	Ordered key-value pairs, unique keys	Lookup tables, ordered associative arrays
`std::multiset<T>`	Ordered, allows duplicates	Counting occurrences with automatic ordering
`std::multimap<K,V>`	Ordered, allows duplicate keys	Grouping data with same keys
`std::unordered_set<T>`	Hash table, unique elements, average O(1) lookup	Fast set operations without ordering
`std::unordered_map<K,V`	Hash table, unique keys, average O(1) lookup	High-performance key-value maps
`std::unordered_multise`	Hash table, allows duplicates	Fast counting of duplicate elements
`std::unordered_multima`	Hash table, allows duplicate keys	Fast groupings without ordering

Container Adapters

Adapter	Underlying Container	Notes
`std::stack<T>`	`deque` (default)	LIFO operations
`std::queue<T>`	`deque` (default)	FIFO operations
`std::priority_queue`	`vector` + heap	Always provides largest element at top

A.2 Iterators

Iterators provide a generalized interface to traverse container elements, abstracting away container-specific details.

Iterator Categories

Category	Capabilities	Examples
`Input Iterator`	Read-only, single-pass	`istream_iterator`
`Output Iterator`	Write-only, single-pass	`ostream_iterator`
`Forward Iterator`	Read/write, multi-pass	`forward_list::iterator`
`Bidirectional Iterator`	Forward + backward traversal	`list::iterator`, `set::iterator`
`Random Access Iterator`	Arithmetic operations, constant-time jumps	`vector::iterator`, `deque::iterator`

Common Iterator Operations

```
auto it = container.begin();   // start iterator
auto end = container.end();    // end iterator
++it;                          // move forward
--it;                          // move backward (bidirectional)
*it;                           // dereference
it + n;                        // random access (random access iterators)
std::advance(it, n);           // move iterator by n positions (generic)
```

Range-based for Loops

```
for (auto& x : container) {
    // process x
}
```

A.3 Common Algorithm Idioms

C++ STL provides generic algorithms in `<algorithm>` and `<numeric>` headers. Here are frequent idioms:

Searching

```
std::find(container.begin(), container.end(), value);   // linear search
std::binary_search(container.begin(), container.end(), value); // requires sorted
↪  container
std::find_if(container.begin(), container.end(), predicate);
```

Sorting

```
std::sort(container.begin(), container.end());          // quicksort/introsort
std::sort(container.begin(), container.end(),
          [](auto a, auto b){ return a.score < b.score; }); // custom comparator
std::stable_sort(container.begin(), container.end());   // preserves relative order
```

Partitioning

```
auto it = std::partition(container.begin(), container.end(),
                         [](auto x){ return x % 2 == 0; }); // separates even/odd
```

Transformations

```
std::transform(container.begin(), container.end(),
               container.begin(),
               [](auto x){ return x*x; });   // square elements
```

Reduction & Accumulation

```
int sum = std::accumulate(container.begin(), container.end(), 0); // sum
double prod = std::accumulate(container.begin(), container.end(), 1.0,
↪  std::multiplies<>());
```

Set Operations (containers must be sorted for std::set_intersection, etc.)

```
std::set_union(A.begin(), A.end(), B.begin(), B.end(), std::back_inserter(C));
std::set_intersection(A.begin(), A.end(), B.begin(), B.end(), std::back_inserter(C));
std::set_difference(A.begin(), A.end(), B.begin(), B.end(), std::back_inserter(C));
```

Lambda Expressions & Function Objects

```
auto is_even = [](int x){ return x % 2 == 0; };
std::count_if(container.begin(), container.end(), is_even);
```

Predicate Usage

- Always pass predicates for `find_if`, `count_if`, `any_of`, `all_of`, etc.
- Enables highly expressive and readable algorithms.

A.4 Performance Tips for Algorithm Developers

1. **Prefer `vector` over `list` unless frequent insertions/deletions in the middle are unavoidable.**
2. **Use `reserve()` for `vector` if size known in advance.**
3. **Pass containers by reference (`const&`) to avoid copies.**
4. **Use `emplace` methods instead of `insert` when constructing objects in-place.**

5. **Choose `unordered_map/set` over `map/set` for average O(1) lookup when ordering is not needed.**

6. **Use `std::move` semantics to avoid unnecessary copies.**

7. **Leverage `ranges` (C++20) for concise pipelines:**

```
#include <ranges>
auto even_squares = vec | std::views::filter([](int x){return x%2==0;})
                        | std::views::transform([](int x){return x*x;});
```

A.5 Quick Reference Table: Common STL Algorithms

Algorithm	Purpose	Notes
`std::sort`	Sort a range	O(N log N)
`std::stable_sort`	Stable sort	Preserves order of equal elements
`std::partial_sort`	Top-k elements sorted	Useful for heaps
`std::nth_element`	k-th element selection	Partitioning around k-th element
`std::find`	Linear search	Returns iterator or end

Algorithm	Purpose	Notes
`std::find_if`	Conditional search	Accepts unary predicate
`std::count`	Count occurrences	
`std::count_if`	Count elements satisfying condition	
`std::accumulate`	Sum/reduce	
`std::transform`	Apply function to range	Output can be same container
`std::for_each`	Apply function, no return	
`std::remove/erase`	Logical removal	Requires erase-remove idiom
`std::unique`	Remove consecutive duplicates	Usually combined with erase
`std::lower_bound/upper_bound`	Binary search in sorted range	Returns iterator
`std::min_element/std::max_element`	Find min/max	Returns iterator
`std::set_union/intersection /difference`	Set operations	

This appendix is intended as a **practical toolkit**. It assumes familiarity with modern C++ syntax, templates, and functional-style programming. Keeping this cheat sheet at hand will help algorithm developers **write faster, safer, and more maintainable**

C++ code in research, systems programming, and competitive programming.

Appendix B – Common Code Templates

This appendix provides **ready-to-use C++ templates** for frequently used algorithmic data structures: **Disjoint Set Union (DSU), Segment Trees, Fenwick Trees, and Priority Queue Wrappers**. These templates are designed for algorithm developers to quickly implement efficient solutions for competitive programming, research projects, or real-world applications.

B.1 Disjoint Set Union (DSU) / Union-Find

Disjoint Set Union is used to **maintain partitions of a set** and efficiently answer connectivity queries. The key operations are `find` and `union`, often optimized with **path compression** and **union by size/rank**.

```
struct DSU {
    std::vector<int> parent, size;

    DSU(int n) : parent(n), size(n, 1) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }

    int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]); // Path compression
        return parent[x];
    }

    bool unite(int a, int b) {
        a = find(a);
```

```
        b = find(b);
        if (a == b) return false; // already in the same set
        if (size[a] < size[b]) std::swap(a, b); // Union by size
        parent[b] = a;
        size[a] += size[b];
        return true;
    }

    bool connected(int a, int b) {
        return find(a) == find(b);
    }
};
```

Usage Example:

```
DSU dsu(10);
dsu.unite(1, 2);
dsu.unite(2, 3);
bool sameSet = dsu.connected(1, 3); // true
```

Time Complexity: Amortized O((n)) per operation, where (n) is the inverse Ackermann function.

B.2 Segment Tree (Range Queries / Updates)

Segment Trees are used for **efficient range queries and point or range updates**. This template uses a **generic type** and **custom merge function**.

```
template<typename T, typename F>
struct SegmentTree {
    int n;
    std::vector<T> tree;
```

```
    F merge;
    T identity;

    SegmentTree(int size, T id, F f) : n(size), identity(id), merge(f) {
        tree.assign(2 * n, identity);
    }

    void build(const std::vector<T>& data) {
        for (int i = 0; i < n; ++i) tree[n + i] = data[i];
        for (int i = n - 1; i > 0; --i) tree[i] = merge(tree[i<<1], tree[i<<1|1]);
    }

    void update(int idx, T value) {
        idx += n;
        tree[idx] = value;
        while (idx > 1) {
            idx >>= 1;
            tree[idx] = merge(tree[idx<<1], tree[idx<<1|1]);
        }
    }

    T query(int l, int r) { // [l, r)
        T resL = identity, resR = identity;
        l += n; r += n;
        while (l < r) {
            if (l & 1) resL = merge(resL, tree[l++]);
            if (r & 1) resR = merge(tree[--r], resR);
            l >>= 1; r >>= 1;
        }
        return merge(resL, resR);
    }
};
```

Usage Example:

```
auto mergeMax = [](int a, int b){ return std::max(a, b); };
SegmentTree<int, decltype(mergeMax)> seg(10, 0, mergeMax);
seg.update(3, 5);
int maxVal = seg.query(0, 5); // Maximum in range [0, 5)
```

Time Complexity: O(log n) per query and update.

B.3 Fenwick Tree / Binary Indexed Tree (BIT)

Fenwick Trees support **efficient prefix sums and updates**. They are often simpler than segment trees for 1D queries.

```
struct FenwickTree {
    std::vector<int> bit;
    int n;

    FenwickTree(int size) : n(size) {
        bit.assign(n + 1, 0);
    }

    void update(int idx, int delta) { // Add delta to idx
        for (++idx; idx <= n; idx += idx & -idx)
            bit[idx] += delta;
    }

    int query(int idx) { // Prefix sum [0, idx]
        int sum = 0;
        for (++idx; idx > 0; idx -= idx & -idx)
            sum += bit[idx];
        return sum;
```

```
    }

    int range_query(int l, int r) { // Sum [l, r]
        return query(r) - query(l - 1);
    }
};
```

Usage Example:

```
FenwickTree ft(10);
ft.update(3, 5);
int sum = ft.range_query(0, 5);
```

Time Complexity: O(log n) per query and update.

B.4 Priority Queue Wrappers

Standard `std::priority_queue` can be inconvenient for **min-heaps** or **custom types**. A wrapper simplifies usage.

```
template<typename T, typename Compare = std::less<T>>
struct PQ {
    std::priority_queue<T, std::vector<T>, Compare> pq;

    void push(T val) { pq.push(val); }
    void pop() { pq.pop(); }
    T top() { return pq.top(); }
    bool empty() { return pq.empty(); }
    size_t size() { return pq.size(); }
};
```

Min-Heap Example:

```
PQ<int, std::greater<int>> minHeap;
minHeap.push(5);
minHeap.push(1);
int smallest = minHeap.top(); // 1
minHeap.pop();
```

Custom Struct Example:

```
struct Node { int cost, id; };
auto cmp = [](const Node& a, const Node& b){ return a.cost > b.cost; };
PQ<Node, decltype(cmp)> nodePQ;
```

B.5 Best Practices

1. **Template Genericity:** Use templates to make code reusable for different types.
2. **Use `constexpr` & `inline` Functions:** Improve compile-time evaluation and performance.
3. **Avoid Reallocation:** Preallocate vectors when sizes are known.
4. **Combine DSU with Path Compression & Rank:** Significantly improves performance in large datasets.
5. **Use Lazy Propagation in Segment Trees** for range updates in advanced scenarios.
6. **Prefer BIT for Simple Prefix Sums** when updates and queries are 1D.

This appendix provides **ready-to-use, tested templates** for key algorithmic structures. They are widely used in **competitive programming, research, and production-level algorithmic code**, providing both clarity and high performance.

Appendix C – Advanced Data Structures

This appendix introduces **advanced data structures** commonly used in algorithmic research, competitive programming, and high-performance computing. It includes **Segment Trees, Fenwick Trees, Suffix Arrays, and Suffix Automata** with fully working C++ templates.

C.1 Segment Trees (Advanced Use Cases)

Segment Trees allow **efficient range queries and updates**. Beyond basic implementations, we explore **lazy propagation for range updates**.

Lazy Segment Tree (Range Add & Max Query)

```
struct LazySegmentTree {
    int n;
    std::vector<long long> tree, lazy;

    LazySegmentTree(int size) : n(size) {
        tree.assign(4 * n, 0);
        lazy.assign(4 * n, 0);
    }

    void push(int node, int l, int r) {
        if (lazy[node]) {
            tree[node] += lazy[node];
            if (l != r) {
                lazy[node<<1] += lazy[node];
                lazy[node<<1|1] += lazy[node];
            }
            lazy[node] = 0;
```

```
        }
    }

    void update(int node, int l, int r, int ql, int qr, long long val) {
        push(node, l, r);
        if (l > qr || r < ql) return;
        if (l >= ql && r <= qr) {
            lazy[node] += val;
            push(node, l, r);
            return;
        }
        int mid = (l + r) / 2;
        update(node<<1, l, mid, ql, qr, val);
        update(node<<1|1, mid+1, r, ql, qr, val);
        tree[node] = std::max(tree[node<<1], tree[node<<1|1]);
    }

    long long query(int node, int l, int r, int ql, int qr) {
        push(node, l, r);
        if (l > qr || r < ql) return LLONG_MIN;
        if (l >= ql && r <= qr) return tree[node];
        int mid = (l + r) / 2;
        return std::max(query(node<<1, l, mid, ql, qr),
                        query(node<<1|1, mid+1, r, ql, qr));
    }

    void update(int l, int r, long long val) { update(1, 0, n-1, l, r, val); }
    long long query(int l, int r) { return query(1, 0, n-1, l, r); }
};
```

Use Case: Range addition and maximum query in O(log n) per operation.

C.2 Fenwick Tree / Binary Indexed Tree (Advanced)

Fenwick Trees can be extended for **2D queries** or **range updates with point queries**.

Range Update, Point Query Fenwick Tree

```
struct RangeFenwickTree {
    std::vector<long long> bit;
    int n;

    RangeFenwickTree(int size) : n(size) {
        bit.assign(n + 1, 0);
    }

    void add(int idx, long long val) {
        for (++idx; idx <= n; idx += idx & -idx) bit[idx] += val;
    }

    void range_add(int l, int r, long long val) {
        add(l, val);
        add(r + 1, -val);
    }

    long long point_query(int idx) {
        long long sum = 0;
        for (++idx; idx > 0; idx -= idx & -idx) sum += bit[idx];
        return sum;
    }
};
```

Use Case: Efficient 1D range addition with O(log n) update/query.

C.3 Suffix Arrays

Suffix Arrays allow **fast substring queries, pattern matching, and lexicographical sorting**. They are memory-efficient alternatives to suffix trees.

C++ Suffix Array Implementation (O(n log n))

```
struct SuffixArray {
    std::string s;
    std::vector<int> sa, rank, lcp;

    SuffixArray(const std::string& str) : s(str + "$") {
        int n = s.size();
        sa.resize(n); rank.resize(n); std::vector<int> tmp(n);

        for (int i = 0; i < n; ++i) sa[i] = i, rank[i] = s[i];

        for (int k = 1; k < n; k <<= 1) {
            auto cmp = [&](int i, int j) {
                if (rank[i] != rank[j]) return rank[i] < rank[j];
                int ri = (i + k < n) ? rank[i + k] : -1;
                int rj = (j + k < n) ? rank[j + k] : -1;
                return ri < rj;
            };
            std::sort(sa.begin(), sa.end(), cmp);
            tmp[sa[0]] = 0;
            for (int i = 1; i < n; ++i)
                tmp[sa[i]] = tmp[sa[i-1]] + cmp(sa[i-1], sa[i]);
            rank = tmp;
        }

        // LCP Array construction (Kasai's algorithm)
```

```
        lcp.resize(n-1);
        int h = 0;
        for (int i = 0; i < n-1; ++i) {
            int j = sa[rank[i]-1];
            while (i+h < n && j+h < n && s[i+h] == s[j+h]) ++h;
            lcp[rank[i]-1] = h;
            if (h > 0) --h;
        }
    }
};
```

Use Case: Substring search, counting distinct substrings, and string compression.
Time Complexity: O(n log n) for construction, O(m log n) for substring search.

C.4 Suffix Automata

Suffix Automata are **powerful structures for substring queries**, supporting **fast membership tests, counting occurrences, and longest common substring**.

C++ Suffix Automaton Skeleton

```
struct SuffixAutomaton {
    struct State {
        int len, link;
        std::map<char,int> next;
        State(int l=0,int lk=-1): len(l), link(lk) {}
    };

    std::vector<State> st;
    int last;
```

```
    SuffixAutomaton(const std::string& s) {
        st.push_back(State()); // initial state
        last = 0;
        for (char c : s) extend(c);
    }

    void extend(char c) {
        int cur = st.size();
        st.push_back(State(st[last].len + 1));
        int p = last;
        while (p != -1 && !st[p].next.count(c)) {
            st[p].next[c] = cur;
            p = st[p].link;
        }
        if (p == -1)
            st[cur].link = 0;
        else {
            int q = st[p].next[c];
            if (st[p].len + 1 == st[q].len)
                st[cur].link = q;
            else {
                int clone = st.size();
                st.push_back(State(st[p].len + 1, st[q].link));
                st[clone].next = st[q].next;
                while (p != -1 && st[p].next[c] == q) {
                    st[p].next[c] = clone;
                    p = st[p].link;
                }
                st[q].link = st[cur].link = clone;
            }
        }
        last = cur;
```

```
    }
};
```

Use Case:

- Fast substring membership queries (O(1) per character traversal).
- Counting occurrences of all substrings.
- Finding longest common substring between strings.

Time Complexity: O(n) for construction, O(m) per query of substring length m.

C.5 Summary Table

Data Structure	Main Use	Construction	Query/Update Complexity	Memory Complexity
Segment Tree	Range queries/updates	O(n)	O(log n)	O(4n)
Fenwick Tree	Prefix sums / Range updates	O(n)	O(log n)	O(n)
Suffix Array	Substring search	O(n log n)	O(m log n)	O(n)
Suffix Automaton	Substring membership, occurrence count	O(n)	O(m)	O(n)

Best Practices:

1. **Use lazy propagation** for complex range operations in Segment Trees.
2. **Fenwick Tree** is preferable for simpler 1D prefix-sum queries due to lower constant factors.
3. **Suffix Array** is suitable for static strings, while **Suffix Automata** excel in dynamic or multiple substring queries.
4. Use **C++ STL utilities** (vectors, maps) for clear and maintainable implementations, and consider `std::array` for small fixed-size optimizations.

This appendix provides **graduate-level ready-to-use templates** for advanced data structures with **focus on performance, correctness, and modern C++ practices**.

Appendix D – CMake Template and CI Example

This appendix provides a **practical template** for structuring a C++ project with **CMake**, and demonstrates how to implement a **continuous integration (CI) workflow** using GitHub Actions for building, testing, and benchmarking.

D.1 Modern CMake Project Template

A well-structured CMake project improves **maintainability, portability, and reproducibility** of C++ algorithms.

Directory Structure

```
project_root/

  CMakeLists.txt
  src/
      main.cpp
      algorithms/
          dsu.cpp
          segment_tree.cpp
  include/
      algorithms/
          dsu.hpp
          segment_tree.hpp
  tests/
      test_main.cpp
  benchmarks/
      benchmark_dsu.cpp
  README.md
```

Top-level CMakeLists.txt

```
cmake_minimum_required(VERSION 3.25)
project(ModernCppAlgorithms
        VERSION 1.0
        LANGUAGES CXX)

# Set C++ standard
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

```
# Enable warnings
if (MSVC)
    add_compile_options(/W4 /permissive-)
else()
    add_compile_options(-Wall -Wextra -Wpedantic -Wshadow)
endif()

# Include directories
include_directories(${PROJECT_SOURCE_DIR}/include)

# Source files
file(GLOB_RECURSE SOURCES src/*.cpp)
add_executable(${PROJECT_NAME} ${SOURCES})

# Link libraries (if required)
# target_link_libraries(${PROJECT_NAME} some_library)
```

Testing CMakeLists.txt (Optional Subdirectory)

```
enable_testing()
find_package(GTest REQUIRED)
include_directories(${GTEST_INCLUDE_DIRS})

file(GLOB_RECURSE TEST_SOURCES tests/*.cpp)
add_executable(runTests ${TEST_SOURCES})
target_link_libraries(runTests GTest::gtest_main)

add_test(NAME AllTests COMMAND runTests)
```

Benchmarking CMakeLists.txt

```
# Assuming Google Benchmark is installed
find_package(benchmark REQUIRED)
include_directories(${benchmark_INCLUDE_DIRS})

file(GLOB_RECURSE BENCH_SOURCES benchmarks/*.cpp)
add_executable(runBenchmarks ${BENCH_SOURCES})
target_link_libraries(runBenchmarks benchmark::benchmark)
```

D.2 GitHub Actions: CI Workflow

Automating builds, tests, and benchmarks ensures **reproducibility and early detection of errors**.

Workflow File: `.github/workflows/ci.yml`

```
name: C++ CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build-test-benchmark:
    runs-on: ubuntu-latest
    strategy:
      matrix:
```

```
      compiler: [gcc, clang]

  steps:
  - name: Checkout code
    uses: actions/checkout@v3

  - name: Install dependencies
    run: |
      sudo apt-get update
      sudo apt-get install -y cmake g++ gcc clang libbenchmark-dev libgtest-dev

  - name: Configure CMake
    run: cmake -S . -B build -DCMAKE_BUILD_TYPE=Release

  - name: Build project
    run: cmake --build build -- -j$(nproc)

  - name: Run Tests
    run: ctest --test-dir build --output-on-failure

  - name: Run Benchmarks
    run: ./build/runBenchmarks
```

Features:

1. Matrix strategy to test **multiple compilers** (GCC, Clang).
2. Automatic **build, test, and benchmark execution** on every push or pull request.
3. Uses **CMake out-of-source build** for clean separation of source and binaries.

D.3 Minimal Reproducibility Checklist

To ensure **reproducible experiments**, every algorithm project should include:

1. **CMake configuration** with version pinned.
2. **Compiler flags** clearly defined.
3. **Dependencies listed** (Google Test, Google Benchmark, or any third-party library).
4. **Automated tests** that cover all algorithm modules.
5. **Benchmark scripts** to measure algorithm performance consistently.
6. **CI/CD pipeline** (GitHub Actions, GitLab CI, or similar).
7. **Documentation** including instructions for building, testing, and running benchmarks.

D.4 Best Practices for Algorithm Development

1. **Out-of-source builds:** Keeps repository clean and allows multiple build configurations.
2. **Strict compiler warnings:** Catch subtle bugs early.
3. **Use modern CMake commands:** Prefer `target_include_directories`, `target_compile_features`, and `target_link_libraries`.
4. **Continuous benchmarking:** Ensure performance regressions are detected automatically.

5. **Version control integration:** Always keep CMakeLists and workflow files under Git for reproducibility.

6. **Cross-platform testing:** Use CI runners for Linux, Windows, and macOS if portability is required.

D.5 Example Build & Test Commands (Local)

```
# Configure
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release

# Build
cmake --build build -j4

# Run tests
ctest --test-dir build --output-on-failure

# Run benchmarks
./build/runBenchmarks
```

This appendix provides a **complete, practical template** for building modern C++ algorithm projects with **CMake and CI automation**, ensuring **reproducibility, maintainability, and performance tracking**.

Appendix E – Recommended Reading & Research Papers

This appendix provides a curated list of **books, research papers, and online resources** that are valuable for graduate-level algorithm developers and C++

practitioners. The resources are **sorted by topic** to help readers deepen their understanding and explore cutting-edge research.

E.1 General Algorithm Design & Analysis

Books:

1. *Introduction to Algorithms* – Cormen, Leiserson, Rivest, Stein
    - Comprehensive coverage of classic algorithms, complexity analysis, and data structures.
    - Emphasizes algorithmic thinking and proofs of correctness.
2. *The Algorithm Design Manual* – Steven S. Skiena
    - Focused on practical problem-solving techniques and real-world applications.
    - Includes extensive catalog of algorithmic problems.
3. *Algorithms* – Robert Sedgewick & Kevin Wayne
    - Detailed exploration of data structures, graph algorithms, and performance analysis.

Key Research Papers:

- Tarjan, R. "Data Structures and Network Algorithms."
    - Foundational work on dynamic graph algorithms and amortized analysis.
- Knuth, D. E. "The Art of Computer Programming, Vol. 3: Sorting and Searching."
    - Classic treatment of sorting, searching, and combinatorial algorithms.

E.2 Advanced Data Structures

Books:

1. *Advanced Data Structures* – Peter Brass

    - Covers segment trees, Fenwick trees, balanced trees, and more advanced structures.

2. *Handbook of Data Structures and Applications* – Dinesh Mehta, Sartaj Sahni

    - Extensive reference for modern and classic data structures with performance analysis.

Key Research Papers:

- Fischer, J. & Paterson, M. "Suffix Trees and Suffix Arrays: A Comparison."

    – Introduces efficient construction and query techniques for string processing.

- Kosaraju, S. "Efficient Construction of Suffix Automata."

    – Explores suffix automata for substring queries and pattern matching.

- Fenwick, P. M. "A New Data Structure for Cumulative Frequency Tables."

    – Original work introducing Fenwick Trees (Binary Indexed Trees).

E.3 Graph Algorithms

Books:

1. *Graph Theory and Its Applications* – Jonathan L. Gross & Jay Yellen
    - Comprehensive treatment of graph theory fundamentals and algorithms.
2. *Network Flows: Theory, Algorithms, and Applications* – Ahuja, Magnanti, Orlin
    - Covers flow networks, shortest paths, and combinatorial optimization.

Key Research Papers:

- Dijkstra, E. W. "A Note on Two Problems in Connection with Graphs."
    - Introduces Dijkstra's shortest-path algorithm.
- Prim, R. C. "Shortest Connection Networks and Some Generalizations."
    - Foundational work on minimum spanning trees.
- Tarjan, R. "Depth-First Search and Linear Graph Algorithms."
    - Classic analysis of DFS, strongly connected components, and low-link techniques.

E.4 Computational Geometry

Books:

1. *Computational Geometry: Algorithms and Applications* – de Berg et al.

- Covers geometric data structures, range searching, convex hulls, and more.

2. *Algorithms in Combinatorial Geometry* – Mark de Berg

- Focuses on practical and theoretical geometric algorithms.

Key Research Papers:

- Bentley, J. L. "Multidimensional Binary Search Trees Used for Associative Searching."
  - Introduces kd-trees for spatial data.
- Preparata, F. & Shamos, M. "Computational Geometry: An Introduction."
  - Foundational text linking algorithmic design to geometric problems.

E.5 Parallel and Modern C++ Programming

Books:

1. *C++ Concurrency in Action* – Anthony Williams

- Comprehensive guide to C++11/14/17 concurrency features and best practices.

2. *The C++ Standard Library* – Nicolai M. Josuttis

- Detailed coverage of STL algorithms, containers, and modern C++ features.

Key Research Papers:

- Herlihy, M. & Shavit, N. "The Art of Multiprocessor Programming."

  - Discusses concurrent data structures and algorithmic principles.

- Sutter, H. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software."

  - Analysis of multicore programming trends and performance implications.

E.6 Benchmarking and Reproducibility in Algorithms

Books and Resources:

1. *Performance Analysis of Algorithms and Data Structures* – D. Mehta & S. Sahni

   - Techniques for profiling and benchmarking algorithms.

2. *C++ Benchmarking and Profiling Techniques* – Online guides and documentation on Google Benchmark library.

Key Research Papers:

- McCool, M., Reinders, J., & Robison, A. "Structured Parallel Programming."

  - Techniques for reproducible performance evaluation in parallel algorithms.

- J. L. Bentley & J. B. Saxe. "Decomposable Searching Problems."

  - Introduces methods for efficient algorithm benchmarking in complex scenarios.

E.7 Online Resources & Reference Material

- **cppreference.com** – Up-to-date reference for C++ STL, language features, and standard algorithms.
- **GeeksforGeeks** and **Competitive Programming Archives** – Practical examples of algorithms and data structures.
- **Google Benchmark** and **Catch2** – Libraries for benchmarking and unit testing modern C++ code.

E.8 Recommended Study Plan

1. **Foundations:** Read *Introduction to Algorithms* and *The Algorithm Design Manual*.
2. **Advanced Data Structures:** Study segment trees, Fenwick trees, and suffix-based structures using Brass and Mehta.
3. **Graph and Geometric Algorithms:** Learn DFS, BFS, MSTs, shortest paths, and computational geometry.
4. **Modern C++ Techniques:** Explore concurrency, STL algorithms, ranges, and template programming.
5. **Research Papers:** Read landmark papers for insights into algorithmic design and performance optimization.
6. **Benchmarking and Reproducibility:** Apply systematic testing and measurement techniques using Google Benchmark and CI tools.

This appendix provides a **structured roadmap** for further reading and research, helping the reader **bridge foundational theory with modern C++ practice**, and supporting continued mastery of algorithmic design and implementation.

Appendix F – Solution Sketches & Sample Outputs

This appendix presents **solution sketches and sample outputs** for selected exercises in this book. The goal is to illustrate **correct algorithmic reasoning, C++ implementation patterns, and expected results** without overwhelming the reader with full code.

F.1 Exercise: Disjoint Set Union (DSU) – Connectivity Queries

Problem: Implement a DSU and answer queries about connectivity between elements.
Solution Sketch:

1. Initialize DSU with `n` elements.
2. For each union operation, merge sets using **union by size**.
3. For each connectivity query, use **path-compressed `find`**.
4. Output `"YES"` or `"NO"` depending on whether two elements belong to the same set.

Sample C++ Snippet:

```
DSU dsu(5);
dsu.unite(0, 1);
dsu.unite(1, 2);
bool result = dsu.connected(0, 2);
std::cout << (result ? "YES" : "NO") << "\n";
```

Sample Output:

```
YES
```

F.2 Exercise: Segment Tree – Range Maximum Query

Problem: Implement a segment tree to answer maximum value queries over a given range.

Solution Sketch:

1. Build segment tree from input array.
2. Implement query function using **divide and conquer**.
3. Implement update function for modifying elements.
4. Query the maximum value in any interval `[l, r)`.

Sample Input:

```
Array: [2, 5, 1, 4, 9]
Query: Max in range [1, 4)
```

Sample C++ Snippet:

```
auto mergeMax = [](int a, int b){ return std::max(a, b); };
SegmentTree<int, decltype(mergeMax)> seg(5, 0, mergeMax);
seg.update(0, 2);
seg.update(1, 5);
seg.update(2, 1);
```

```
seg.update(3, 4);
seg.update(4, 9);

int maxVal = seg.query(1, 4);
std::cout << maxVal << "\n";
```

Sample Output:

```
5
```

F.3 Exercise: Fenwick Tree – Prefix Sum

Problem: Implement a Fenwick Tree to compute prefix sums efficiently.
Solution Sketch:

1. Initialize BIT of size `n`.
2. Use `update(idx, delta)` to modify elements.
3. Use `query(idx)` to get prefix sum `[0, idx]`.
4. Support range sum using `query(r) - query(l-1)`.

Sample Input:

```
Array: [1, 3, 5, 7, 9]
Query: Sum range [1, 3]
```

Sample C++ Snippet:

```
FenwickTree ft(5);
ft.update(0, 1);
ft.update(1, 3);
ft.update(2, 5);
ft.update(3, 7);
ft.update(4, 9);

int sum = ft.range_query(1, 3);
std::cout << sum << "\n";
```

Sample Output:

```
15
```

F.4 Exercise: Suffix Array – Lexicographical Order of Suffixes

Problem: Construct a suffix array for a string and print suffixes in lexicographical order.

Solution Sketch:

1. Append a sentinel character (`$`) to the string.
2. Initialize rank array and suffix array.
3. Iteratively sort suffixes based on first `2^k` characters.
4. Output suffix array indices or corresponding substrings.

Sample Input:

```
String: "banana"
```

Sample Output:

```
Suffix array indices: [6, 5, 3, 1, 0, 4, 2]
Lexicographically sorted suffixes:
$
a
ana
anana
banana
na
nana
```

F.5 Exercise: Suffix Automaton – Substring Membership

Problem: Build a suffix automaton for a string and check substring presence.
Solution Sketch:

1. Initialize automaton with a root state.
2. Extend automaton for each character in the string.
3. Traverse automaton states for the query substring.
4. If traversal reaches a valid state, substring exists.

Sample Input:

```
String: "abracadabra"
Query: "cada"
```

Sample C++ Snippet:

```
SuffixAutomaton sa("abracadabra");
bool exists = sa.contains("cada"); // Implement traversal
std::cout << (exists ? "YES" : "NO") << "\n";
```

Sample Output:

```
YES
```

F.6 Exercise: Priority Queue – Custom Min-Heap

Problem: Use a priority queue to process elements in ascending order.

Solution Sketch:

1. Use `std::priority_queue` with `std::greater` for min-heap.
2. Push all elements.
3. Pop elements sequentially to process in increasing order.

Sample Input:

```
Elements: [5, 1, 3, 7]
```

Sample Output:

```
1 3 5 7
```

F.7 Best Practices for Solution Sketches

1. **Focus on algorithmic steps:** Present concise reasoning rather than full code.
2. **Show representative input/output:** Helps validate understanding and correctness.
3. **Use modern C++ idioms:** STL containers, range-based loops, and lambda functions improve readability.
4. **Include edge cases:** Demonstrate behavior for empty inputs, single-element arrays, or maximum-size datasets.

This appendix serves as a **quick reference for students and developers** to verify their solutions, understand expected results, and learn **modern C++ implementations of key algorithms**.