CP Algorithms & Techniques

Mensi Mohamed Amine

Abstract

Competitive programming requires fast and efficient problem-solving using key algorithms. This paper highlights essential techniques like sorting, graphs, and dynamic programming to boost performance.

1 Introduction

Competitive programming tests algorithm skills and speed. Knowing core algorithms helps solve problems effectively. This article covers fundamental techniques for success.

Contents

2 Competitive Programming: Essential Algorithms and Techniques

Competitive programming requires a strong grasp of algorithms, data structures, and problem-solving techniques. Below is a structured guide to essential algorithms and techniques used in competitive programming.

2.1 Basic Data Structures

- **Arrays & Strings**: Sliding window, Prefix sum, Kadane's algorithm.
- **Linked Lists**: Cycle detection, Reversal, Fast & slow pointers.
- **Stacks & Queues**: Monotonic stacks, Queue using stacks, Deque.
- **Hash Maps / Sets**: Frequency counting, Two-sum problem, Hashing tricks.
- **Heaps (Priority Queues)**: K-th smallest/largest element, Dijkstra's algorithm.

2.2 Sorting & Searching

- **Sorting Algorithms**:
  - QuickSort, MergeSort, HeapSort.
  - Counting Sort, Radix Sort (for linear time sorting).
- **Binary Search**:
  - Standard binary search.
  - Lower/upper bound, Binary search on answer.
  - Ternary search (for unimodal functions).

2.3 Graph Algorithms

- **Traversal**:
  - **BFS**: Shortest path in unweighted graphs, Flood fill.
  - **DFS**: Cycle detection, Topological sorting.
- **Shortest Path**:
  - **Dijkstra's**: For non-negative weights.
  - **Bellman-Ford**: For negative weights.
  - **Floyd-Warshall**: All-pairs shortest path.
- **Minimum Spanning Tree (MST)**:
  - **Kruskal's** (with DSU).
  - **Prim's**.
- **Strongly Connected Components (SCC)**:
  - Kosaraju's, Tarjan's algorithm.
- **Network Flow**:
  - **Ford-Fulkerson**, **Edmonds-Karp** (Max flow).
  - **Bipartite Matching** (Kuhn's algorithm).

2.4 Dynamic Programming (DP)

- **Classic Problems**:
  - Fibonacci, Knapsack, LCS (Longest Common Subsequence), LIS (Longest Increasing Subsequence).
- **DP Techniques**:
  - Memoization vs Tabulation.
  - Bitmask DP (TSP - Traveling Salesman Problem).
  - Digit DP (Counting numbers with properties).
  - DP on Trees (Independent Set, Diameter).
  - DP with Matrix Exponentiation (Fast recurrence solving).
- **Optimizations**:
  - Sliding Window (Space optimization).
  - Convex Hull Trick (CHT) for DP speedup.

2.5 Number Theory & Math

- **Prime Numbers**:
  - Sieve of Eratosthenes, Miller-Rabin primality test.
- **Modular Arithmetic**:
  - Fermat's Little Theorem, Modular Inverse.
  - Chinese Remainder Theorem (CRT).
- **Combinatorics**:
  - Permutations, Combinations (nCr), Catalan numbers.
- **Fast Exponentiation**:
  - Binary exponentiation (PowMod).
- **GCD & LCM**:
  - Euclidean algorithm, Extended Euclidean algorithm.

2.6 String Algorithms

- **Pattern Matching**:
  - KMP, Z-Algorithm, Rabin-Karp.
- **Trie (Prefix Tree)**:
  - Word search, XOR problems.
- **Suffix Structures**:
  - Suffix Array, LCP (Kasai's algorithm).
  - Suffix Automaton (Advanced).
- **Manacher's Algorithm**: Longest palindromic substring.

2.7 Advanced Techniques

- **Greedy Algorithms**:
  - Interval scheduling, Huffman coding.
- **Divide & Conquer**:
  - Merge Sort, Fast Fourier Transform (FFT).
- **Bit Manipulation**:
  - Bitmasking, XOR tricks, Fenwick Tree (BIT).
- **Mo's Algorithm**:
  - Offline range queries (sqrt decomposition).
- **Segment Trees & Lazy Propagation**:
  - Range queries and updates.
- **Persistent Data Structures**:
  - Persistent segment trees.

2.8 Game Theory

- Grundy Numbers (Nim Game).
- Sprague-Grundy Theorem.
- Minimax Algorithm (Alpha-Beta Pruning).

2.9 Geometry

- Convex Hull (Graham Scan, Andrew's Algorithm).
- Line Intersection & Sweep Line Algorithm.
- Closest Pair of Points (Divide & Conquer).

2.10 Miscellaneous Tricks

- Two Pointers Technique.
- Meet-in-the-Middle (for NP problems).
- Heuristics & Randomized Algorithms (Monte Carlo).

Final Tips for Competitive Programming

1. Master Time Complexity Analysis (Big-O notation).
2. Practice on Online Judges (Codeforces, LeetCode, AtCoder, SPOJ).
3. Learn Standard Template Library (STL) (C++ `std::set`, `std::map`, `std::priority_queue`).
4. Participate in Contests (ACM ICPC, Google Code Jam, CodeChef Long Challenge).
5. Upsolve Problems (Review unsolved problems after contests).

Would you like a deeper dive into any specific topic?

3 Sliding Window Technique in C++ (Arrays & Strings)

Sliding window is an optimization technique that reduces the time complexity of subarray/substring problems from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Below, we cover **fixed-size** and **variable-size** sliding windows with C++ examples.

3.1 Fixed-Size Sliding Window

Problem: Maximum Sum Subarray of Size k
Given an array `arr` and integer k, find the maximum sum of any contiguous subarray of size k.

Example:
Input: `arr` $= [2, 1, 5, 1, 3, 2]$, $k = 3$
Output: 9 (Subarray $[5, 1, 3]$)

```cpp
#include <iostream>
#include <vector>
#include <climits> // INT_MIN

using namespace std;

int maxSubarraySum(const vector<int>& arr, int k) {
    int windowSum = 0, maxSum = INT_MIN;
    int windowStart = 0;

    for (int windowEnd = 0; windowEnd < arr.size(); ++windowEnd) {
        windowSum += arr[windowEnd]; // Add next element

        if (windowEnd >= k - 1) {
            maxSum = max(maxSum, windowSum);
            windowSum -= arr[windowStart]; // Remove leftmost element
            windowStart++;                 // Slide window forward
        }
    }
    return maxSum;
}

int main() {
    vector<int> arr = {2, 1, 5, 1, 3, 2};
    int k = 3;
    cout << "Maximum sum: " << maxSubarraySum(arr, k) << endl; // Output: 9
    return 0;
}
```

Explanation:

- `windowSum` tracks the current sum of the window.
- `windowEnd` expands the window.
- When window size reaches k, update `maxSum`, subtract the leftmost element, and move `windowStart` forward.

3.2 Variable-Size Sliding Window

Problem: Smallest Subarray with Sum $\geq$ Target
Find the length of the smallest contiguous subarray whose sum is at least `target`.

Example:
Input: `arr` $= [2, 1, 5, 2, 3, 2]$, `target` $= 7$
Output: 2 (Subarray $[5, 2]$)

```cpp
#include <iostream>
#include <vector>
#include <climits> // INT_MAX

using namespace std;
```

```cpp
int smallestSubarrayWithSum(const vector<int>& arr, int target) {
    int minLength = INT_MAX, windowSum = 0, windowStart = 0;

    for (int windowEnd = 0; windowEnd < arr.size(); ++windowEnd) {
        windowSum += arr[windowEnd]; // Expand window

        while (windowSum >= target) {
            minLength = min(minLength, windowEnd - windowStart + 1);
            windowSum -= arr[windowStart]; // Shrink window
            windowStart++;
        }
    }
    return (minLength == INT_MAX) ? 0 : minLength;
}

int main() {
    vector<int> arr = {2, 1, 5, 2, 3, 2};
    int target = 7;
    cout << "Smallest subarray length: " << smallestSubarrayWithSum(arr, target) << endl; // Output: 2
    return 0;
}
```

Explanation:

- `windowEnd` expands the window by adding elements.
- While `windowSum` is $\geq$ target, update minimum length and shrink the window from the left.

3.3 Sliding Window on Strings

Problem: Longest Substring with k Distinct Characters
Find the longest substring with at most k distinct characters.

Example:
Input: s = "araaci", $k = 2$
Output: 4 (Substring "araa")

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

using namespace std;

int longestSubstringKDistinct(const string& s, int k) {
    unordered_map<char, int> freqMap;
    int maxLen = 0, windowStart = 0;

    for (int windowEnd = 0; windowEnd < s.size(); ++windowEnd) {
        char rightChar = s[windowEnd];
        freqMap[rightChar]++;

        while (freqMap.size() > k) {
            char leftChar = s[windowStart];
            freqMap[leftChar]--;
            if (freqMap[leftChar] == 0)
                freqMap.erase(leftChar);
            windowStart++;
        }
        maxLen = max(maxLen, windowEnd - windowStart + 1);
    }
    return maxLen;
}

int main() {
    string s = "araaci";
    int k = 2;
    cout << "Longest substring length: " << longestSubstringKDistinct(s, k) << endl; // Output: 4
    return 0;
}
```

Explanation:

- `freqMap` tracks character frequencies in the current window.
- Expand `windowEnd` and add characters.
- Shrink window if number of distinct chars exceeds k.

3.4 Common Sliding Window Problems

Problem	**Type**	**Approach**
Maximum sum subarray of size k	Fixed	Track window sum
Smallest subarray with sum $\geq$ target	Variable	Shrink when sum $\geq$ target
Longest substring with $\leq k$ distinct chars	Variable	Frequency map (`unordered_map`)
Find all anagrams of a string	Fixed	Compare frequency maps
Minimum window substring (LeetCode Hard)	Variable	Expand and shrink with two pointers

3.5 Key Takeaways

- **Fixed-size window**: Maintain constant window size ($\texttt{windowEnd} - \texttt{windowStart} + 1 = k$).
- **Variable-size window**: Expand (`windowEnd++`) and shrink (`windowStart++`) dynamically.
- Use `unordered_map` for frequency tracking in string problems.
- Optimizes brute-force $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.

Would you like more problems or optimization techniques?

4 Prefix Sum Technique (Arrays & Strings)

Prefix sum (also called cumulative sum or scan) is a powerful technique that precomputes cumulative sums to answer range sum queries in constant time. It's widely used in competitive programming for efficient range operations.

4.1 Basic Concept

- Precompute an array `prefix` where `prefix[i]` is the sum of elements from `arr[0]` to `arr[i]`.
- Allows calculating any subarray sum in $\mathcal{O}(1)$ time.

4.1.1 Formula

```
prefix[i] = prefix[i-1] + arr[i]  // for i > 0
prefix[0] = arr[0]                // base case

// Sum from index a to b:
sum(a, b) = prefix[b] - prefix[a-1]  // when a > 0
sum(0, b) = prefix[b]                // when a = 0
```

4.2 C++ Implementation

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

using namespace std;

vector<int> buildPrefixSum(const vector<int>& arr) {
    vector<int> prefix(arr.size());
    prefix[0] = arr[0];
    for (int i = 1; i < arr.size(); i++) {
        prefix[i] = prefix[i-1] + arr[i];
    }
    return prefix;
}

int rangeSum(const vector<int>& prefix, int l, int r) {
    if (l == 0) return prefix[r];
    return prefix[r] - prefix[l-1];
}

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    vector<int> prefix = buildPrefixSum(arr);

    cout << "Sum from index 1 to 3: " << rangeSum(prefix, 1, 3) << endl; // 2+3+4 = 9
    cout << "Sum from index 0 to 4: " << rangeSum(prefix, 0, 4) << endl; // 1+2+3+4+5 = 15
    return 0;
}
```

4.3 Common Applications

- **Range Sum Queries** – Find sum of any subarray in $\mathcal{O}(1)$ time.
- **Equilibrium Index** – Find index where sum before = sum after.
- **Counting Subarrays** – Count subarrays with a given sum.
- **Range Updates** – Efficiently apply multiple range additions.

4.4 Example Problems

4.4.1 Count Subarrays with Sum K

```
int countSubarrays(vector<int>& nums, int k) {
    unordered_map<int, int> prefix_counts;
    prefix_counts[0] = 1;
    int sum = 0, count = 0;

    for (int num : nums) {
        sum += num;
        if (prefix_counts.find(sum - k) != prefix_counts.end()) {
            count += prefix_counts[sum - k];
        }
        prefix_counts[sum]++;
    }
    return count;
}
```

4.4.2 Find Pivot Index

```
int pivotIndex(vector<int>& nums) {
    int total = accumulate(nums.begin(), nums.end(), 0);
    int left_sum = 0;

    for (int i = 0; i < nums.size(); i++) {
        if (left_sum == total - left_sum - nums[i]) {
            return i;
        }
        left_sum += nums[i];
    }
    return -1;
}
```

4.5 Variations

- 2D Prefix Sum – For matrix range queries.
- Circular Prefix Sum – For circular array problems.
- Prefix Product – Same principle but with multiplication instead of addition.

4.6 Time Complexity

Operation	**Time**
Preprocessing	$\mathcal{O}(n)$
Range Query	$\mathcal{O}(1)$
Space Complexity	$\mathcal{O}(n)$

Prefix sum is particularly useful when you need to answer many range queries on a static array. The initial $\mathcal{O}(n)$ preprocessing cost gets amortized over multiple $\mathcal{O}(1)$ queries.

5 Kadane's Algorithm (Maximum Subarray Sum)

Kadane's Algorithm is an efficient $\mathcal{O}(n)$ solution to find the **maximum sum of a contiguous subarray** in a given array of numbers. It's one of the most important algorithms for array processing in competitive programming.

5.1 Key Idea

- Maintain a running sum of the current subarray.
- Reset the running sum if it becomes negative (since a negative sum will only decrease future sums).
- Track the maximum sum encountered at each step.

5.2 Standard Implementation (C++)

```cpp
#include <vector>
#include <climits> // for INT_MIN

using namespace std;

int kadane(vector<int>& nums) {
    int max_current = nums[0];
    int max_global = nums[0];

    for (int i = 1; i < nums.size(); i++) {
        // Choose between extending the subarray or starting fresh
        max_current = max(nums[i], max_current + nums[i]);

        // Update global maximum if current subarray is better
        if (max_current > max_global) {
            max_global = max_current;
        }
    }

    return max_global;
}

// Example usage:
int main() {
    vector<int> arr = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    cout << "Maximum subarray sum: " << kadane(arr) << endl; // Output: 6 (from [4,-1,2,1])
    return 0;
}
```

5.3 Variations and Extensions

5.3.1 A. Handling All-Negative Arrays

```cpp
int kadane(vector<int>& nums) {
    int max_current = nums[0];
    int max_global = nums[0];

    for (int i = 1; i < nums.size(); i++) {
        max_current = max(nums[i], max_current + nums[i]);
        max_global = max(max_global, max_current);
    }

    return max_global;
}
```

5.3.2 B. Finding Subarray Indices

```cpp
vector<int> kadaneWithIndices(vector<int>& nums) {
    int max_current = nums[0];
    int max_global = nums[0];
    int start = 0, end = 0;
    int temp_start = 0;

    for (int i = 1; i < nums.size(); i++) {
        if (nums[i] > max_current + nums[i]) {
            max_current = nums[i];
            temp_start = i;
        } else {
            max_current += nums[i];
        }

        if (max_current > max_global) {
            max_global = max_current;
            start = temp_start;
            end = i;
        }
    }

    return {max_global, start, end};
}
```

5.3.3 C. Maximum Circular Subarray Sum

```
int maxCircularSum(vector<int>& nums) {
    // Case 1: Maximum sum using standard Kadane's
    int max_kadane = kadane(nums);

    // Case 2: Maximum sum wraps around (total - min_subarray)
    int total_sum = 0;
    for (int num : nums) total_sum += num;

    // Invert array and find minimum subarray
    vector<int> inverted = nums;
    for (int& num : inverted) num = -num;
    int min_subarray = -kadane(inverted);

    int max_wrap = total_sum - min_subarray;

    // Handle all negative case
    if (max_wrap == 0) return max_kadane;

    return max(max_kadane, max_wrap);
}
```

5.4 Common Problems Solved by Kadane's

Problem	**Solution Approach**
Maximum Subarray Sum	Standard Kadane's
Maximum Product Subarray	Modified Kadane's tracking min/max
Longest Alternating Subarray	Kadane's with state tracking
Maximum Sum Circular Subarray	Combination of Kadane's and inverted Kadane's

5.5 Time and Space Complexity

- **Time Complexity:** $\mathcal{O}(n)$ – Single pass through the array.
- **Space Complexity:** $\mathcal{O}(1)$ – Uses constant extra space.

5.6 Practice Problems

1. Maximum Subarray (LeetCode 53)
2. Maximum Product Subarray (LeetCode 152)
3. Maximum Sum Circular Subarray (LeetCode 918)
4. Best Time to Buy and Sell Stock (LeetCode 121)

Kadane's algorithm is a fundamental technique that every competitive programmer should master due to its wide applicability in array processing problems.

6 Cycle Detection in Linked Lists (Floyd's Algorithm)

Cycle detection is a classic problem in linked list manipulation, and Floyd's Tortoise and Hare algorithm provides an elegant $\mathcal{O}(n)$ time, $\mathcal{O}(1)$ space solution.

6.1 Floyd's Cycle-Finding Algorithm

6.1.1 Key Idea

- Use two pointers: **slow** (tortoise) moves 1 step, **fast** (hare) moves 2 steps.
- If there's a cycle, they will eventually meet.
- If fast reaches `NULL`, no cycle exists.

6.1.2 C++ Implementation

```cpp
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};

bool hasCycle(ListNode *head) {
    if (!head || !head->next) return false;

    ListNode *slow = head;
    ListNode *fast = head->next;

    while (slow != fast) {
        if (!fast || !fast->next) return false;
        slow = slow->next;
        fast = fast->next->next;
    }

    return true;
}
```

6.2 Finding the Cycle Start Node

6.2.1 Mathematical Insight

When slow and fast meet:

- Reset slow to head.
- Move both pointers at 1 step until they meet again.
- The meeting point is the cycle start.

6.2.2 Implementation

```cpp
ListNode *detectCycle(ListNode *head) {
    if (!head || !head->next) return nullptr;

    ListNode *slow = head;
    ListNode *fast = head;

    // First phase: find meeting point
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) break;
    }

    // No cycle
    if (slow != fast) return nullptr;
    // Second phase: find cycle start
    slow = head;
    while (slow != fast) {
        slow = slow->next;
        fast = fast->next;
    }

    return slow;
}
```

6.3 Time & Space Complexity

Operation	Complexity
Detection	$\mathcal{O}(n)$ time, $\mathcal{O}(1)$ space
Finding Start	$\mathcal{O}(n)$ time, $\mathcal{O}(1)$ space

6.4 Common Variations

6.4.1 Cycle Length Calculation

```
int cycleLength(ListNode *meet) {
    int length = 1;
    ListNode *temp = meet->next;
    while (temp != meet) {
        length++;
        temp = temp->next;
    }
    return length;
}
```

6.4.2 Check if Linked List is Palindrome

Using slow/fast pointers to find the middle of the list is a key technique when solving palindrome-related problems.

6.5 Why This Matters in Interviews

- Tests understanding of pointer manipulation.
- Demonstrates space optimization.
- Serves as a foundation for more complex linked list problems.

6.6 Practice Problems

1. Linked List Cycle (LeetCode 141)
2. Linked List Cycle II (LeetCode 142)
3. Happy Number (LeetCode 202) – Cycle detection in number sequences

Floyd's algorithm is a must-know for any technical interview involving linked lists, combining elegant mathematics with practical implementation.

7 Linked List Reversal Techniques

Reversing a linked list is a fundamental operation that tests your understanding of pointer manipulation. Here are the key methods to reverse a linked list in C++:

7.1 Iterative Reversal (Most Efficient)

Time Complexity: $\mathcal{O}(n)$
Space Complexity: $\mathcal{O}(1)$

```cpp
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};

ListNode* reverseList(ListNode* head) {
    ListNode *prev = nullptr;
    ListNode *curr = head;

    while (curr) {
        ListNode *nextTemp = curr->next; // Store next node
        curr->next = prev;              // Reverse link
        prev = curr;                    // Move prev forward
        curr = nextTemp;                // Move curr forward
    }

    return prev; // New head
}
```

Visualization:

```
Original: 1 → 2 → 3 → 4 → 5 → NULL
Step-by-step:
NULL ← 1  2 → 3 → 4 → 5
NULL ← 1 ← 2  3 → 4 → 5
...
NULL ← 1 ← 2 ← 3 ← 4 ← 5 (final)
```

7.2 Recursive Reversal

Time Complexity: $\mathcal{O}(n)$
Space Complexity: $\mathcal{O}(n)$ (call stack)

```cpp
ListNode* reverseListRecursive(ListNode* head) {
    if (!head || !head->next) return head;

    ListNode* newHead = reverseListRecursive(head->next);
    head->next->next = head; // Reverse link
    head->next = nullptr;     // Break old link

    return newHead;
}
```

Recursion Stack Explanation:

1. Recurses to end of list (newHead becomes 5)
2. On the way back: makes each node point to its predecessor

7.3 Partial Reversal (Between Positions m and n)

```cpp
ListNode* reverseBetween(ListNode* head, int m, int n) {
    if (!head || m == n) return head;
    ListNode dummy(0);
    dummy.next = head;
    ListNode *pre = &dummy;

    // Move pre to m-1 position
    for (int i = 0; i < m-1; ++i)
        pre = pre->next;

    ListNode *curr = pre->next;

    // Reverse segment between m and n
    for (int i = 0; i < n-m; ++i) {
        ListNode *temp = curr->next;
        curr->next = temp->next;
        temp->next = pre->next;
        pre->next = temp;
    }

    return dummy.next;
}
```

7.4 Common Variations

7.4.1 Reverse in Groups of Size k

```
ListNode* reverseKGroup(ListNode* head, int k) {
    ListNode *curr = head;
    int count = 0;

    // First check if at least k nodes remain
    while (curr && count < k) {
        curr = curr->next;
        count++;
    }

    if (count == k) {
        curr = reverseKGroup(curr, k); // Reverse remaining
        // Reverse current group
        while (count-- > 0) {
            ListNode *temp = head->next;
            head->next = curr;
            curr = head;
            head = temp;
        }
        head = curr;
    }
    return head;
}
```

7.4.2 Reverse Alternating Groups

Implementation involves toggling the reversal for each k-sized group.

7.4.3 Palindrome Check (Using Reversal)

Reverse second half of list and compare with first.

7.5 Key Insights

- **Iterative Method** is preferred in production (no stack overflow risk).
- **Dummy Nodes** help handle edge cases (e.g., empty list, single node).
- **Partial Reversal** requires careful pointer bookkeeping.

7.6 Practice Problems

1. Reverse Linked List (LeetCode 206)
2. Reverse Linked List II (LeetCode 92)
3. Reverse Nodes in k-Group (LeetCode 25)

Mastering list reversal is crucial for technical interviews and forms the basis for more complex linked list operations.

8 Fast & Slow Pointers Technique in Linked Lists

The fast & slow pointers technique (also known as the "tortoise and hare" approach) is a powerful method for solving linked list problems with optimal efficiency. This pattern uses two pointers moving at different speeds to solve problems in $\mathcal{O}(n)$ time with $\mathcal{O}(1)$ space.

8.1 Core Concept

- **Slow pointer**: Moves 1 node at a time (`slow = slow->next`)
- **Fast pointer**: Moves 2 nodes at a time (`fast = fast->next->next`)
- Terminates when fast reaches the end (detects cycles, finds middle, etc.)

8.2 Key Applications

8.2.1 A. Cycle Detection (Floyd's Algorithm)

```
bool hasCycle(ListNode *head) {
    if (!head) return false;

    ListNode *slow = head;
    ListNode *fast = head->next;

    while (fast && fast->next) {
        if (slow == fast) return true;
        slow = slow->next;
        fast = fast->next->next;
    }

    return false;
}
```

Time: $\mathcal{O}(n)$ **Space:** $\mathcal{O}(1)$

8.2.2 B. Finding Middle Node

```
ListNode* findMiddle(ListNode *head) {
    ListNode *slow = head, *fast = head;

    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow; // For even lists, returns second middle
}
```

Example:

- Input: `1 → 2 → 3 → 4 → 5` $\rightarrow$ Middle = 3
- Input: `1 → 2 → 3 → 4` $\rightarrow$ Middle = 3 (second middle)

8.2.3 C. Finding Cycle Start Node

```
ListNode *detectCycleStart(ListNode *head) {
    ListNode *slow = head, *fast = head;

    // Find meeting point
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) break;
    }
    if (!fast || !fast->next) return nullptr; // No cycle

    // Reset slow to head
    slow = head;
    while (slow != fast) {
        slow = slow->next;
        fast = fast->next;
    }

    return slow; // Cycle start node
}
```

8.3 Advanced Variations

8.3.1 A. Palindrome Check

```
bool isPalindrome(ListNode* head) {
    if (!head || !head->next) return true;

    // Find middle and reverse second half
    ListNode *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }

    ListNode *secondHalf = reverseList(slow);

    // Compare both halves
    while (secondHalf) {
        if (head->val != secondHalf->val) return false;
        head = head->next;
        secondHalf = secondHalf->next;
    }

    return true;
}
```

8.3.2 B. Find Nth Node from End

```
ListNode* findNthFromEnd(ListNode* head, int n) {
    ListNode *fast = head, *slow = head;

    // Move fast n nodes ahead
    for (int i = 0; i < n; ++i) {
        if (!fast) return nullptr; // List shorter than n
        fast = fast->next;
    }

    // Move both until fast reaches end
    while (fast) {
        slow = slow->next;
        fast = fast->next;
    }

    return slow;
}
```

8.4 Why This Technique Matters

- **Optimal Efficiency**: Solves problems in a single pass ($\mathcal{O}(n)$ time)
- **Space Efficiency**: Uses only two pointers ($\mathcal{O}(1)$ space)
- **Versatile**: Applies to cycle detection, middle finding, palindrome checks, etc.

8.5 Practice Problems

1. Linked List Cycle (LeetCode 141)
2. Middle of Linked List (LeetCode 876)
3. Palindrome Linked List (LeetCode 234)
4. Remove Nth Node From End (LeetCode 19)

Mastering fast & slow pointers will give you an edge in technical interviews and competitive programming challenges involving linked lists!

9 Monotonic Stacks: A Powerful Data Structure Technique

Monotonic stacks are a specialized stack variant that maintain elements in either strictly increasing or decreasing order. They excel at solving problems requiring finding the next/previous greater/smaller element efficiently.

9.1 Core Concept

A monotonic stack is a stack where elements are:

- **Monotonically increasing**: Each new element $\geq$ top element
- **Monotonically decreasing**: Each new element $\leq$ top element

Key Property: When pushing a new element, we pop elements that violate the monotonic property before pushing.

9.2 Common Applications

9.2.1 A. Next Greater Element (NGE)

Find the first element to the right that's greater than the current element.

```cpp
vector<int> nextGreaterElements(vector<int>& nums) {
    stack<int> s;
    vector<int> res(nums.size(), -1);

    for (int i = 0; i < nums.size(); i++) {
        while (!s.empty() && nums[s.top()] < nums[i]) {
            res[s.top()] = nums[i];
            s.pop();
        }
        s.push(i);
    }

    return res;
}
```

9.2.2 B. Previous Smaller Element

Find the first element to the left that's smaller than the current element.

```cpp
vector<int> prevSmallerElements(vector<int>& nums) {
    stack<int> s;
    vector<int> res(nums.size(), -1);

    for (int i = nums.size()-1; i >= 0; i--) {
        while (!s.empty() && nums[s.top()] > nums[i]) {
            res[s.top()] = nums[i];
            s.pop();
        }
        s.push(i);
    }

    return res;
}
```

9.3 Implementation Patterns

9.3.1 Increasing vs Decreasing Stacks

Type	Condition	Typical Use
Increasing	`nums[i] >= stack.top()`	Next smaller element
Decreasing	`nums[i] <= stack.top()`	Next greater element

9.3.2 Template for Monotonic Stack Problems

```cpp
vector<int> monotonicStackSolution(vector<int>& nums) {
    stack<int> s;
    vector<int> res(nums.size());

    for (int i = 0; i < nums.size(); i++) {
        // Maintain monotonic property
        while (!s.empty() && nums[s.top()] COMPARISON nums[i]) {
            // Process the element being popped
            res[s.top()] = ANSWER;
            s.pop();
        }
        s.push(i);
    }

    // Process remaining elements in stack
    while (!s.empty()) {
        res[s.top()] = DEFAULT_ANSWER;
        s.pop();
    }

    return res;
}
```

9.4 Advanced Applications

9.4.1 A. Largest Rectangle in Histogram

```cpp
int largestRectangleArea(vector<int>& heights) {
    stack<int> s;
    int max_area = 0;
    heights.push_back(0); // Sentinel value

    for (int i = 0; i < heights.size(); ) {
        if (s.empty() || heights[i] > heights[s.top()]) {
            s.push(i++);
        } else {
            int h = heights[s.top()]; s.pop();
            int w = s.empty() ? i : i - s.top() - 1;
            max_area = max(max_area, h * w);
        }
    }

    return max_area;
}
```

9.4.2 B. Trapping Rain Water

```cpp
int trap(vector<int>& height) {
    stack<int> s;
    int water = 0;

    for (int i = 0; i < height.size(); i++) {
        while (!s.empty() && height[i] > height[s.top()]) {
            int top = s.top(); s.pop();
            if (s.empty()) break;
            int distance = i - s.top() - 1;
            int bounded_height = min(height[i], height[s.top()]) - height[top];
            water += distance * bounded_height;
        }
        s.push(i);
    }

    return water;
}
```

9.5 Time Complexity Analysis

- **Time Complexity:** $\mathcal{O}(n)$ — Each element is pushed and popped at most once
- **Space Complexity:** $\mathcal{O}(n)$ — Worst case stack size

9.6 Practice Problems

1. Next Greater Element I (LeetCode 496)
2. Daily Temperatures (LeetCode 739)
3. Largest Rectangle in Histogram (LeetCode 84)
4. Trapping Rain Water (LeetCode 42)

Monotonic stacks provide elegant solutions to problems involving element comparisons in sequences, making them essential for technical interviews and competitive programming.

10 Implementing a Queue Using Stacks

Queues (FIFO) can be implemented using stacks (LIFO) via two common approaches. Both methods preserve the first-in-first-out order, despite the inherent last-in-first-out nature of stacks.

10.1 Two-Stack Approach (Amortized $\mathcal{O}(1)$ Operations)

10.1.1 Concept

- **Input stack**: Handles all push operations.
- **Output stack**: Handles all pop/peek operations.
- When the output stack is empty, transfer all elements from the input stack to it.

```
class MyQueue {
private:
    stack<int> input, output;

public:
    void push(int x) {
        input.push(x);
    }

    int pop() {
        if (output.empty()) {
            while (!input.empty()) {
                output.push(input.top());
                input.pop();
            }
        }
        int val = output.top();
        output.pop();
        return val;
    }

    int peek() {
        if (output.empty()) {
            while (!input.empty()) {
                output.push(input.top());
                input.pop();
            }
        }
        return output.top();
    }

    bool empty() {
        return input.empty() && output.empty();
    }
};
```

10.1.2 Complexity Analysis

Operation	Time Complexity
Push	$\mathcal{O}(1)$
Pop	Amortized $\mathcal{O}(1)$
Peek	Amortized $\mathcal{O}(1)$
Empty	$\mathcal{O}(1)$

10.2 Single Stack (Recursive) Approach

10.2.1 Concept

- Uses the call stack to reverse element order.
- Suitable only for small queues due to recursion depth limits.

```
class MyQueue {
private:
    stack<int> s;

public:
    void push(int x) {
        s.push(x);
    }

    int pop() {
        int top = s.top();
        s.pop();
        if (s.empty()) {
            return top;
        }
        int item = pop();
        s.push(top);
        return item;
    }

    int peek() {
        int top = s.top();
        s.pop();
        if (s.empty()) {
            s.push(top);
            return top;
        }
        int item = peek();
        s.push(top);
        return item;
    }

    bool empty() {
        return s.empty();
    }
};
```

10.2.2 Complexity Analysis

Operation	Time Complexity
Push	$\mathcal{O}(1)$
Pop	$\mathcal{O}(n)$
Peek	$\mathcal{O}(n)$
Empty	$\mathcal{O}(1)$

10.3 Key Differences

Approach	Pros	Cons
Two-Stack	Amortized $\mathcal{O}(1)$ operations	Requires additional space
Recursive	Simple implementation	Risk of stack overflow

10.4 Practical Applications

- Implementing queues where only stack structures are available.
- Algorithmic problems such as level-order tree traversal.
- Interview questions testing understanding of stack and queue behavior.

10.5 Common Interview Variations

1. Implement a queue using one stack and recursion.
2. Implement a queue where the middle element can be deleted in $\mathcal{O}(1)$.
3. Implement a queue with $\mathcal{O}(1)$ time find-min or find-max operations.

The two-stack approach is typically preferred in practice due to its optimal time complexity and avoidance of recursion stack overflow.

11 Deque (Double-Ended Queue) Implementation

A **deque** (pronounced "deck") is a linear data structure that supports insertion and deletion at both ends with $\mathcal{O}(1)$ time complexity. It combines the capabilities of both stacks and queues.

11.1 Core Operations

Operation	Description	Time Complexity
`push_front(x)`	Insert at front	$\mathcal{O}(1)$
`push_back(x)`	Insert at end	$\mathcal{O}(1)$
`pop_front()`	Remove from front	$\mathcal{O}(1)$
`pop_back()`	Remove from end	$\mathcal{O}(1)$
`front()`	Get front element	$\mathcal{O}(1)$
`back()`	Get back element	$\mathcal{O}(1)$
`empty()`	Check if empty	$\mathcal{O}(1)$
`size()`	Get current size	$\mathcal{O}(1)$

11.2 Implementation Approaches

A. Using Doubly Linked List (Optimal)

```cpp
#include <list>
template <typename T>
class Deque {
private:
    std::list<T> container;

public:
    void push_front(T x) { container.push_front(x); }
    void push_back(T x)  { container.push_back(x); }
    void pop_front()     { container.pop_front(); }
    void pop_back()      { container.pop_back(); }
    T front()            { return container.front(); }
    T back()             { return container.back(); }
    bool empty()         { return container.empty(); }
    size_t size()        { return container.size(); }
};
```

B. Using Circular Array (Fixed Capacity)

```cpp
template <typename T>
class ArrayDeque {
private:
    T* arr;
    int capacity;
    int front_idx;
    int back_idx;
    int current_size;

public:
    ArrayDeque(int cap) : capacity(cap), front_idx(0),
                          back_idx(cap-1), current_size(0) {
        arr = new T[capacity];
    }

    ~ArrayDeque() { delete[] arr; }

    void push_front(T x) {
        if (full()) throw std::overflow_error("Deque full");
        front_idx = (front_idx - 1 + capacity) % capacity;
        arr[front_idx] = x;
        current_size++;
    }

    void push_back(T x) {
        if (full()) throw std::overflow_error("Deque full");
        back_idx = (back_idx + 1) % capacity;
        arr[back_idx] = x;
        current_size++;
    }

    // Other methods follow similar pattern...
};
```

11.3 STL Implementation

C++ provides `std::deque` in the standard library:

```
#include <deque>

std::deque<int> dq;

// Basic operations
dq.push_front(10);
dq.push_back(20);
dq.pop_front();
dq.pop_back();

// Access elements
int first = dq.front();
int last = dq.back();

// Other useful methods
dq.size();
dq.empty();
dq.clear();
```

11.4 Key Applications

- Sliding Window Problems
- Undo/Redo Functionality
- Work Balancing in Parallel Systems
- Palindrome Checking
- Breadth-First Search (BFS) with Level Tracking

11.5 Advanced Techniques

A. Monotonic Deque for Sliding Window Maximum

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

    for (int i = 0; i < nums.size(); i++) {
        // Remove elements outside window
        if (!dq.empty() && dq.front() == i - k)
            dq.pop_front();

        // Maintain decreasing order
        while (!dq.empty() && nums[dq.back()] < nums[i])
            dq.pop_back();

        dq.push_back(i);

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

B. Deque vs Other Structures

Structure	Insert Front	Insert Back	Random Access	Memory
Deque	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	Higher
Vector	$\mathcal{O}(n)$	Amort. $\mathcal{O}(1)$	$\mathcal{O}(1)$	Medium
List	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Highest

11.6 Practice Problems

1. Sliding Window Maximum (LeetCode 239)
2. Design Circular Deque (LeetCode 641)
3. Palindrome Linked List (LeetCode 234)
4. Maximum of All Subarrays of Size K (GFG)

Deques are powerful and flexible containers, essential in algorithmic design for problems requiring efficient two-ended access and modifications.

12 Frequency Counting with Hash Maps/Sets

Frequency counting is a fundamental technique that uses hash-based structures to count occurrences of elements, enabling efficient lookups and computations. This section provides a comprehensive guide to mastering this essential skill.

12.1 Core Techniques

A. Basic Frequency Map (C++)

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

void countFrequencies(const vector<int>& nums) {
    unordered_map<int, int> freq;

    // Count frequencies
    for (int num : nums) {
        freq[num]++;
    }

    // Print frequencies
    for (auto& [num, count] : freq) {
        cout << num << ": " << count << endl;
    }
}
```

B. Character Frequency in Strings

```cpp
unordered_map<char, int> countChars(const string& s) {
    unordered_map<char, int> freq;
    for (char c : s) {
        freq[c]++;
    }
    return freq;
}
```

12.2 Common Applications

A. Finding Duplicates

```cpp
bool containsDuplicate(vector<int>& nums) {
    unordered_set<int> seen;
    for (int num : nums) {
        if (seen.count(num)) return true;
        seen.insert(num);
    }
    return false;
}
```

B. Two Sum Problem

```cpp
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> num_map; // value -> index

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

C. Anagram Detection

```
bool isAnagram(string s, string t) {
    if (s.length() != t.length()) return false;

    int count[26] = {0};

    for (char c : s) count[c - 'a']++;
    for (char c : t) count[c - 'a']--;

    for (int i = 0; i < 26; i++) {
        if (count[i] != 0) return false;
    }
    return true;
}
```

12.3 Advanced Patterns

A. Top K Frequent Elements

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    unordered_map<int, int> freq;
    for (int num : nums) freq[num]++;

    priority_queue<pair<int, int>> pq;
    for (auto& [num, count] : freq) {
        pq.push({-count, num});
        if (pq.size() > k) pq.pop();
    }

    vector<int> result;
    while (!pq.empty()) {
        result.push_back(pq.top().second);
        pq.pop();
    }
    return result;
}
```

B. Frequency Counting with Constraints

```
int maxOperations(vector<int>& nums, int k) {
    unordered_map<int, int> freq;
    int operations = 0;

    for (int num : nums) {
        int complement = k - num;
        if (freq[complement] > 0) {
            operations++;
            freq[complement]--;
        } else {
            freq[num]++;
        }
    }

    return operations;
}
```

12.4 Optimizations & Tradeoffs

Technique	When to Use	Time	Space
Hash Map	General case	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Array Count	Limited value range	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Bitmask	Boolean flags	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Multi-set	Need sorted frequencies	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$

12.5 Practice Problems

1. Valid Anagram (LeetCode 242)
2. Group Anagrams (LeetCode 49)
3. Intersection of Two Arrays (LeetCode 349)
4. First Unique Character (LeetCode 387)
5. Subarray Sum Equals K (LeetCode 560)

12.6 Pro Tips

- Use arrays instead of hash maps when dealing with limited character sets (e.g., lowercase English letters)
- Consider sorted structures when you need ordered frequencies
- Combine with sliding window techniques for substring or subarray problems
- Prefer `unordered_map` over `map` unless ordering is necessary ($\mathcal{O}(1)$ vs $\mathcal{O}(\log n)$ operations)

Frequency counting is a versatile and foundational technique in algorithm design, essential for efficient solutions in array and string problems.

13 Two Sum Problem: Comprehensive Guide

The Two Sum problem is a classic algorithmic challenge that tests your ability to use hash maps efficiently. Here's a deep dive into multiple approaches with optimizations.

13.1 Problem Statement

Given an array of integers `nums` and an integer `target`, return indices of the two numbers that add up to the target.

Example:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1] (because nums[0] + nums[1] = 9)
```

13.2 Brute Force Approach

Time Complexity: $\mathcal{O}(n^2)$
Space Complexity: $\mathcal{O}(1)$

```
vector<int> twoSum(vector<int>& nums, int target) {
    for (int i = 0; i < nums.size(); i++) {
        for (int j = i + 1; j < nums.size(); j++) {
            if (nums[i] + nums[j] == target) {
                return {i, j};
            }
        }
    }
    return {};
}
```

13.3 Optimal Hash Map Solution

Time Complexity: $\mathcal{O}(n)$
Space Complexity: $\mathcal{O}(n)$

```
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> num_map; // Stores value -> index

    for (int i = 0; i < nums.size(); i++) {
        int complement = target - nums[i];
        if (num_map.find(complement) != num_map.end()) {
            return {num_map[complement], i};
        }
        num_map[nums[i]] = i; // Store after checking to avoid self-match
    }
    return {};
}
```

Key Insights:

1. Single pass through the array
2. Stores each number's index only after checking its complement
3. Handles duplicate values correctly

13.4 Variations and Edge Cases

A. Return Values Instead of Indices

```
vector<int> twoSumValues(vector<int>& nums, int target) {
    unordered_set<int> seen;
    for (int num : nums) {
        int complement = target - num;
        if (seen.count(complement)) {
            return {complement, num};
        }
        seen.insert(num);
    }
    return {};
}
```

B. Sorted Input Array (Two Pointer Approach)

Time: $\mathcal{O}(n)$
Space: $\mathcal{O}(1)$

```
vector<int> twoSumSorted(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) {
            return {left, right};
        } else if (sum < target) {
            left++;
        } else {
            right--;
        }
    }
    return {};
}
```

13.5 Comparative Analysis

Method	**Time**	**Space**	**When to Use**
Brute Force	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Never in interviews
Hash Map	$\mathcal{O}(n)$	$\mathcal{O}(n)$	General case
Two Pointer	$\mathcal{O}(n)$	$\mathcal{O}(1)$	Sorted input

13.6 Common Mistakes to Avoid

- **Returning the same index twice:** Always check the complement before inserting the current number into the map.
- **Assuming the input is sorted:** Clarify constraints before applying two-pointer logic.
- **Using `map` instead of `unordered_map`:** The latter provides $\mathcal{O}(1)$ average-case lookups.
- **Not handling negative numbers:** Hash map works for negative values too.

13.7 Practice Problems

1. Two Sum (LeetCode 1)
2. Two Sum II - Sorted Array (LeetCode 167)
3. Two Sum Less Than K (LeetCode 1099)
4. 3Sum (LeetCode 15) (Extension)

13.8 Pro Tips

- Clarify input constraints (duplicates allowed? negative numbers?)
- Consider space-time tradeoffs when selecting between hash map and two-pointer
- Test edge cases:
  - Empty array

  - No solution exists
  - Multiple valid solutions
  - Target = 0 with zero elements in array

The Two Sum problem is foundational—mastering it prepares you for more complex problems like 3Sum and 4Sum.

14 Advanced Hashing Tricks for Competitive Programming

Hashing is a powerful technique that goes far beyond simple frequency counting. Here are some sophisticated hashing tricks that can give you an edge in competitive programming.

14.1 Rolling Hash for String Matching

Rabin-Karp Algorithm (Pattern Search)

```
vector<int> rabinKarp(string text, string pattern) {
    const int p = 31;
    const int mod = 1e9+9;
    int T = text.size(), P = pattern.size();

    // Precompute powers of p
    vector<long long> p_pow(max(T, P));
    p_pow[0] = 1;
    for (int i = 1; i < p_pow.size(); i++)
        p_pow[i] = (p_pow[i-1] * p) % mod;

    // Compute hash of pattern
    long long h_pattern = 0;
    for (int i = 0; i < P; i++)
        h_pattern = (h_pattern + (pattern[i] - 'a' + 1) * p_pow[i]) % mod;

    // Compute prefix hashes of text
    vector<long long> h_text(T+1, 0);
    for (int i = 0; i < T; i++)
        h_text[i+1] = (h_text[i] + (text[i] - 'a' + 1) * p_pow[i]) % mod;

    // Find matches
    vector<int> occurrences;
    for (int i = 0; i <= T - P; i++) {
        long long curr_hash = (h_text[i+P] - h_text[i] + mod) % mod;
        if (curr_hash == h_pattern * p_pow[i] % mod)
            occurrences.push_back(i);
    }
    return occurrences;
}
```

14.2 Custom Hash for Unordered Containers

Preventing Hash Collision Attacks

```
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};

// Usage:
unordered_map<long long, int, custom_hash> safe_map;
unordered_set<long long, custom_hash> safe_set;
```

14.3 Multi-Dimensional Hashing

Hashing Pairs and Tuples

```
struct pair_hash {
    template <class T1, class T2>
    size_t operator()(const pair<T1, T2>& p) const {
        auto hash1 = hash<T1>{}(p.first);
        auto hash2 = hash<T2>{}(p.second);
        return hash1 ^ (hash2 << 1);
    }
};

// Usage:
unordered_map<pair<int, int>, int, pair_hash> pair_map;
```

14.4 Bloom Filter (Probabilistic Hashing)

Space-Efficient Membership Test

```
class BloomFilter {
    vector<bool> bits;
    vector<function<size_t(string)>> hash_functions;

public:
    BloomFilter(int size, int num_hashes) : bits(size) {
        for (int i = 0; i < num_hashes; i++) {
            hash_functions.push_back([i](string s) {
                hash<string> hasher;
                return hasher(s + to_string(i)) % size;
            });
        }
    }

    void add(string item) {
        for (auto& hash_fn : hash_functions) {
            bits[hash_fn(item)] = true;
        }
    }

    bool possiblyContains(string item) {
        for (auto& hash_fn : hash_functions) {
            if (!bits[hash_fn(item)]) return false;
        }
        return true;
    }
};
```

14.5 Count-Min Sketch (Frequency Estimation)

Streaming Data Frequency Tracking

```
class CountMinSketch {
    vector<vector<int>> counts;
    vector<function<size_t(string)>> hash_functions;

public:
    CountMinSketch(int width, int depth) :
        counts(depth, vector<int>(width, 0)) {
        for (int i = 0; i < depth; i++) {
            hash_functions.push_back([i, width](string s) {
                hash<string> hasher;
                return hasher(s + to_string(i)) % width;
            });
        }
    }

    void update(string item, int count = 1) {
        for (int i = 0; i < hash_functions.size(); i++) {
            counts[i][hash_functions[i](item)] += count;
        }
    }

    int estimate(string item) {
        int min_count = INT_MAX;
        for (int i = 0; i < hash_functions.size(); i++) {
            min_count = min(min_count, counts[i][hash_functions[i](item)]);
        }
        return min_count;
    }
};
```

14.6 XOR Hashing Tricks

Finding Unique Elements

```
int findUnique(const vector<int>& nums) {
    int result = 0;
    for (int num : nums) result ^= num;
    return result;
}

// Finds two unique numbers where others appear twice
pair<int, int> findTwoUniques(const vector<int>& nums) {
    int xor_sum = 0;
    for (int num : nums) xor_sum ^= num;

    int set_bit = xor_sum & -xor_sum;
    int a = 0, b = 0;

    for (int num : nums) {
        if (num & set_bit) a ^= num;
        else b ^= num;
    }

    return {a, b};
}
```

14.7 Applications in Problem Solving

- **String Matching** (Rabin-Karp)
- **Geometric Hashing** (Point set matching)
- **Data Stream Analysis** (Count-Min Sketch)
- **Memoization** (Custom hash for complex keys)
- **Cryptography** (Simple hashing applications)

14.8 Performance Considerations

- **Load Factor:** Keep `unordered_map`'s load factor below 0.75 for efficiency
- **Prime Moduli:** Use prime numbers for mod operations to ensure better hash distribution
- **Hash Quality:** Ensure custom hash functions have a strong avalanche effect to reduce collisions

These advanced hashing techniques can significantly optimize solutions for problems involving:

- Large string processing
- Streaming data analysis
- Geometric algorithms
- Memory-constrained environments
- Probability-based computations

15 Finding K-th Smallest / Largest Element Using Heaps

15.1 Overview

Heaps (priority queues) are highly effective for finding k-th order statistics such as the k-th smallest or largest element. They offer flexible performance for static arrays, streaming data, and dynamic updates.

15.2 Solution Approaches

A. K-th Smallest Element

1. Max-Heap Approach (Best when $k \ll n$) **Use case:** Small k with large data, one-time queries

```
int findKthSmallest(vector<int>& nums, int k) {
    priority_queue<int> max_heap;

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

    return max_heap.top();
}
```

Time Complexity: $O(n \log k)$
Space Complexity: $O(k)$

2. Min-Heap In-Place Approach **Use case:** Can modify input array

```
int findKthSmallest(vector<int>& nums, int k) {
    make_heap(nums.begin(), nums.end(), greater<int>());
    for (int i = 0; i < k - 1; i++) {
        pop_heap(nums.begin(), nums.end(), greater<int>());
        nums.pop_back();
    }
    return nums.front();
}
```

Time Complexity: $O(n + k \log n)$
Space Complexity: $O(1)$

B. K-th Largest Element

1. Min-Heap Approach (Best when $k \ll n$) **Use case:** Find top-k largest efficiently

```
int findKthLargest(vector<int>& nums, int k) {
    priority_queue<int, vector<int>, greater<int>> min_heap;

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

    return min_heap.top();
}
```

Time Complexity: $O(n \log k)$
Space Complexity: $O(k)$

2. Max-Heap In-Place Approach **Use case:** Input array can be modified

```
int findKthLargest(vector<int>& nums, int k) {
    make_heap(nums.begin(), nums.end());
    for (int i = 0; i < k - 1; i++) {
        pop_heap(nums.begin(), nums.end());
        nums.pop_back();
    }
    return nums.front();
}
```

Time Complexity: $O(n + k \log n)$
Space Complexity: $O(1)$

15.3 Comparison of Approaches

Approach	Time Complexity	Space	Best For
Max-Heap (Smallest)	$O(n \log k)$	$O(k)$	Small k, streaming
Min-Heap (Smallest)	$O(n + k \log n)$	$O(1)$	In-place, large k
Min-Heap (Largest)	$O(n \log k)$	$O(k)$	Top-k queries
Max-Heap (Largest)	$O(n + k \log n)$	$O(1)$	In-place, large k

15.4 Alternative Methods

Quickselect (Average $O(n)$ Time)

Best when: One-time query, mutable data

```
int quickselect(vector<int>& nums, int l, int r, int k) {
    int pivot = nums[r], i = l;
    for (int j = l; j < r; j++) {
        if (nums[j] <= pivot) swap(nums[i++], nums[j]);
    }
    swap(nums[i], nums[r]);

    int count = i - l + 1;
    if (count == k) return nums[i];
    return (count > k) ? quickselect(nums, l, i-1, k)
                       : quickselect(nums, i+1, r, k - count);
}
```

Median of Medians (Worst-case $O(n)$)

Used in strict time-bound algorithms (e.g., introselect)
Omitted for brevity — requires multiple helpers.

15.5 Edge Cases to Consider

- $k >$ size of array
- $k = 0$ or $k = 1$
- Duplicates in array
- All elements equal
- Empty array

15.6 Practice Problems

- Kth Largest Element in an Array (LeetCode 215)
- Top K Frequent Elements (LeetCode 347)
- K Closest Points to Origin (LeetCode 973)
- Find Median from Data Stream (LeetCode 295)

15.7 Pro Tips

1. For multiple queries, sort array once and access in $O(1)$
2. Max-heap is more efficient for small k
3. Min-heap better when k is large
4. Use heap for streaming; Quickselect for single query
5. In C++, `priority_queue` is max-heap by default
6. Use `priority_queue<T, vector<T>, greater<T>>` for min-heap

16 Dijkstra's Algorithm Using Priority Queues

16.1 Overview

Dijkstra's algorithm computes the shortest paths from a single source node to all other nodes in a weighted graph with non-negative edge weights. It leverages a min-heap (priority queue) to efficiently process nodes based on the smallest tentative distances.

16.2 Core Algorithm

Key Components

- **Priority Queue (Min-Heap)**: Selects node with minimum current distance
- **Distance Array**: Stores shortest distance to each node
- **Visited Set**: Optional for skipping already processed nodes

C++ Implementation

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

using namespace std;

vector<int> dijkstra(const vector<vector<pair<int, int>>>& graph, int source) {
    int n = graph.size();
    vector<int> dist(n, INT_MAX);
    dist[source] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.push({0, source});

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

        if (current_dist > dist[u]) continue;

        for (auto& [v, weight] : graph[u]) {
            if (dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

16.3 Complexity Analysis

Operation	**Time Complexity**	**Notes**
Initialization	$O(V)$	Distance array setup
Heap Operations	$O(\log V)$	Insert/extract-min
Processing Edges	$O(E \log V)$	Each edge relaxes once
Total Time	$\boldsymbol{O((V + E) \log V)}$	Dominated by heap ops

Space Complexity: $O(V + E)$ for graph + $O(V)$ for distances and heap

16.4 Practical Optimizations

A. Early Termination

```cpp
if (u == target) break; // Stop once target is reached
```

B. Path Reconstruction

```
vector<int> parent(n, -1); // Track predecessors

if (dist[v] > dist[u] + weight) {
    dist[v] = dist[u] + weight;
    parent[v] = u;
    pq.push({dist[v], v});
}

vector<int> getPath(int target, const vector<int>& parent) {
    vector<int> path;
    for (int v = target; v != -1; v = parent[v])
        path.push_back(v);
    reverse(path.begin(), path.end());
    return path;
}
```

16.5 Common Variations

A. Maximum Probability Path (LeetCode 1514)

```
vector<double> maxProbability(int n, vector<vector<int>>& edges,
                            vector<double>& succProb, int start) {
    vector<vector<pair<int, double>>> graph(n);
    for (int i = 0; i < edges.size(); i++) {
        int u = edges[i][0], v = edges[i][1];
        double prob = succProb[i];
        graph[u].emplace_back(v, prob);
        graph[v].emplace_back(u, prob);
    }

    vector<double> probs(n, 0.0);
    probs[start] = 1.0;

    priority_queue<pair<double, int>> pq;
    pq.push({1.0, start});

    while (!pq.empty()) {
        auto [current_prob, u] = pq.top(); pq.pop();
        if (current_prob < probs[u]) continue;

        for (auto& [v, prob] : graph[u]) {
            if (probs[v] < probs[u] * prob) {
                probs[v] = probs[u] * prob;
                pq.push({probs[v], v});
            }
        }
    }

    return probs;
}
```

B. Multi-Criteria Dijkstra

```
struct State {
    int node, distance, cost;
    bool operator>(const State& other) const {
        return distance > other.distance;
    }
};

vector<int> multiDijkstra(...) {
    priority_queue<State, vector<State>, greater<>> pq;
    // Additional handling of multiple attributes (e.g., cost constraints)
}
```

16.6 Edge Cases and Validation

- Negative edge weights $\rightarrow$ Use Bellman-Ford
- Disconnected nodes remain at INT_MAX
- Multiple edges $\rightarrow$ Automatically selects minimum
- Self-loops or zero-weight edges are allowed

16.7 Comparison With Other Algorithms

Algorithm	**Time Complexity**	**Handles Negative Weights?**	**Best For**
Dijkstra	$O((V+E)\log V)$	No	Single-source, positive weights
Bellman-Ford	$O(VE)$	Yes	Negative edges or detection
Floyd-Warshall	$O(V^3)$	Yes	All-pairs shortest paths
A* Search	$O((V+E)\log V)$	No	Heuristic-guided shortest path

16.8 Practice Problems

- Network Delay Time (LeetCode 743)
- Cheapest Flights Within K Stops (LeetCode 787)
- Path With Maximum Probability (LeetCode 1514)
- Minimum Cost to Reach Destination (LeetCode 1928)

16.9 Pro Tips

1. Always initialize distances to INT_MAX
2. For grid-based problems, adapt neighbors accordingly
3. Use a max-heap with inverted costs for max path problems
4. Prefer adjacency lists over matrices for sparse graphs
5. Use visited set or check condition `if (cur > dist[u])` to skip stale paths

17 QuickSort Algorithm: Comprehensive Guide

QuickSort is a highly efficient, comparison-based, divide-and-conquer sorting algorithm. It has an average-case time complexity of $O(n \log n)$ and is widely used due to its excellent performance in practice.

17.1 Core Algorithm

Key Components

- **Partitioning**: Select a pivot and rearrange elements so that:
  - Elements less than the pivot come before it
  - Elements greater than the pivot come after it
- **Recursion**: Apply the same process to sub-arrays

Standard Implementation (Lomuto Partition)

```
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            swap(arr[i], arr[j]);
            i++;
        }
    }
    swap(arr[i], arr[high]);
    return i;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void quickSort(vector<int>& arr) {
    quickSort(arr, 0, arr.size() - 1);
}
```

17.2 Optimizations

A. Hoare's Partition Scheme

```
int partitionHoare(vector<int>& arr, int low, int high) {
    int pivot = arr[low + (high - low)/2];
    int i = low - 1, j = high + 1;
    while (true) {
        do { i++; } while (arr[i] < pivot);
        do { j--; } while (arr[j] > pivot);
        if (i >= j) return j;
        swap(arr[i], arr[j]);
    }
}
```

B. Three-Way Partitioning (for duplicates)

```
void quickSort3Way(vector<int>& arr, int low, int high) {
    if (high <= low) return;
    int lt = low, gt = high;
    int pivot = arr[low], i = low;
    while (i <= gt) {
        if (arr[i] < pivot)       swap(arr[lt++], arr[i++]);
        else if (arr[i] > pivot)  swap(arr[i], arr[gt--]);
        else                      i++;
    }
    quickSort3Way(arr, low, lt - 1);
    quickSort3Way(arr, gt + 1, high);
}
```

C. Hybrid with Insertion Sort (for small arrays)

```
void hybridQuickSort(vector<int>& arr, int low, int high) {
    while (low < high) {
        if (high - low < 16) {
            insertionSort(arr, low, high); // Assume defined elsewhere
            break;
        }
        int pi = partition(arr, low, high);
        if (pi - low < high - pi) {
            hybridQuickSort(arr, low, pi - 1);
            low = pi + 1;
        } else {
            hybridQuickSort(arr, pi + 1, high);
            high = pi - 1;
        }
    }
}
```

17.3 Complexity Analysis

Case	Time	Space	Notes
Best	$O(n \log n)$	$O(\log n)$	Balanced partitions
Average	$O(n \log n)$	$O(\log n)$	On random input
Worst	$O(n^2)$	$O(n)$	Already sorted input, bad pivot
Optimized Worst	$O(n \log n)$	$O(\log n)$	With good pivot strategy

17.4 Pivot Selection Strategies

- **First or Last Element**: Simple, but risky for sorted input
- **Middle Element**: Safer for nearly sorted arrays
- **Random Element**: Reduces chance of worst-case
- **Median-of-Three**:

```
int choosePivot(vector<int>& arr, int low, int high) {
    int mid = low + (high - low)/2;
    if (arr[high] < arr[low]) swap(arr[low], arr[high]);
    if (arr[mid] < arr[low]) swap(arr[mid], arr[low]);
    if (arr[high] < arr[mid]) swap(arr[mid], arr[high]);
    return mid;
}
```

17.5 Comparison with Other Sorting Algorithms

Algorithm	Best	Average	Worst	Space	Stable
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
IntroSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	No

17.6 Practical Considerations

1. QuickSort has excellent cache performance.

2. Difficult to parallelize due to data dependencies.
3. Not stable unless extra space is used.
4. Used in C++ STL's `std::sort()` via IntroSort.

17.7 Common Mistakes

- Forgetting the base case `low < high`.
- Poor pivot selection leading to $O(n^2)$ time.
- Stack overflow from deep recursion.
- Index errors during partitioning.

17.8 Practice Problems

- Sort an Array (LeetCode 912)
- Kth Largest Element (LeetCode 215)
- Wiggle Sort II (LeetCode 324)
- Sort Colors (LeetCode 75)

17.9 Pro Tips

1. Use three-way partitioning to handle many duplicates efficiently.
2. Use insertion sort for very small subarrays (e.g., size < 16).
3. For performance and safety, prefer random or median-of-three pivots.
4. Consider IntroSort in production (used in `std::sort`).

18 MergeSort Algorithm: Comprehensive Guide

MergeSort is a stable, comparison-based sorting algorithm that uses the divide-and-conquer approach, guaranteeing $O(n \log n)$ time complexity in all cases.

18.1 Core Algorithm

Key Components

1. **Divide**: Split the array into two halves
2. **Conquer**: Recursively sort each half
3. **Merge**: Combine the sorted halves into a single sorted array

Standard Implementation

```
void merge(vector<int>& arr, int left, int mid, int right) {
    vector<int> temp(right - left + 1);
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) temp[k++] = arr[i++];
        else temp[k++] = arr[j++];
    }
    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    for (int p = 0; p < k; p++) {
        arr[left + p] = temp[p];
    }
}

void mergeSort(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);
}

void mergeSort(vector<int>& arr) {
    mergeSort(arr, 0, arr.size() - 1);
}
```

18.2 Optimizations

A. Hybrid MergeSort with Insertion Sort

```
const int INSERTION_THRESHOLD = 16;

void insertionSort(vector<int>& arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void optimizedMergeSort(vector<int>& arr, int left, int right) {
    if (right - left <= INSERTION_THRESHOLD) {
        insertionSort(arr, left, right);
        return;
    }
```

```cpp
    int mid = left + (right - left) / 2;
    optimizedMergeSort(arr, left, mid);
    optimizedMergeSort(arr, mid + 1, right);

    if (arr[mid] <= arr[mid + 1]) return; // Skip merge if already sorted

    merge(arr, left, mid, right);
}
```

B. Bottom-Up (Iterative) MergeSort

```cpp
void bottomUpMergeSort(vector<int>& arr) {
    int n = arr.size();
    vector<int> temp(n);

    for (int width = 1; width < n; width *= 2) {
        for (int i = 0; i < n; i += 2 * width) {
            int left = i;
            int mid = min(i + width, n);
            int right = min(i + 2 * width, n);

            int i1 = left, i2 = mid, k = left;
            while (i1 < mid && i2 < right) {
                if (arr[i1] <= arr[i2]) temp[k++] = arr[i1++];
                else temp[k++] = arr[i2++];
            }
            while (i1 < mid) temp[k++] = arr[i1++];
            while (i2 < right) temp[k++] = arr[i2++];

            for (int j = left; j < right; j++) {
                arr[j] = temp[j];
            }
        }
    }
}
```

18.3 Complexity Analysis

Case	Time	Space	Notes
Best	$O(n \log n)$	$O(n)$	Same for all cases
Average	$O(n \log n)$	$O(n)$	Consistent performance
Worst	$O(n \log n)$	$O(n)$	Guaranteed upper bound

18.4 Key Properties

- **Stable**: Maintains relative order of equal elements
- **Not in-place**: Requires $O(n)$ extra memory
- **Parallelizable**: Suitable for distributed systems
- **Consistent**: Performance does not degrade on specific inputs

18.5 Comparison with Other Sorting Algorithms

Algorithm	Time	Space	Stable	Best for
MergeSort	$O(n \log n)$	$O(n)$	Yes	Large datasets, stability
QuickSort	$O(n \log n)$ avg	$O(\log n)$	No	General purpose, cache-friendly
HeapSort	$O(n \log n)$	$O(1)$	No	Space-limited environments
TimSort	$O(n \log n)$	$O(n)$	Yes	Real-world data (Python, Java default)

18.6 Practical Considerations

1. Requires extra $O(n)$ space, which can be costly for huge data.
2. Less cache-friendly than QuickSort due to access patterns.

3. Excellent choice for sorting linked lists (uses $O(1)$ extra space).
4. Basis for external sorting of massive datasets.

18.7 Common Mistakes

- Incorrect midpoint calculation (use `left + (right - left) / 2` to avoid overflow).
- Forgetting base case: always return when `left >= right`.
- Insufficient temporary array size during merging.
- Violating stability by using strict less-than (`<`) instead of less-or-equal (`<=`) during merge comparisons.

18.8 Real-World Applications

- Database sorting
- External sorting of large files
- Counting inversions in arrays
- Distributed sorting (e.g., MapReduce frameworks)
- Any scenario requiring stable sorting

18.9 Practice Problems

- Sort an Array (LeetCode 912)
- Count of Smaller Numbers After Self (LeetCode 315)
- Reverse Pairs (LeetCode 493)
- Sort List (LeetCode 148)

18.10 Summary

MergeSort provides predictable $O(n \log n)$ performance and stability, making it ideal for applications where these properties outweigh its $O(n)$ extra space requirement.

19 HeapSort Algorithm: Comprehensive Guide

HeapSort is an efficient, in-place, comparison-based sorting algorithm with guaranteed $O(n \log n)$ time complexity in all cases, making it particularly useful when predictable performance is required.

19.1 Core Algorithm

Key Components

1. **Heap Construction**: Build a max-heap from the input array
2. **Sorting**: Repeatedly extract the maximum element and maintain heap property

Standard Implementation

```cpp
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;         // Initialize largest as root
    int left = 2 * i + 1;    // Left child
    int right = 2 * i + 2;   // Right child

    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int>& arr) {
    int n = arr.size();

    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

19.2 Optimizations

A. Bottom-Up Heap Construction

```cpp
void buildHeap(vector<int>& arr) {
    int n = arr.size();
    for (int i = (n / 2) - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
}
```

B. Iterative Heapify

```cpp
void heapifyIterative(vector<int>& arr, int n, int i) {
    while (true) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && arr[left] > arr[largest])
            largest = left;
        if (right < n && arr[right] > arr[largest])
            largest = right;
        if (largest == i) break;
        swap(arr[i], arr[largest]);
        i = largest;
    }
}
```

19.3 Complexity Analysis

Case	**Time**	**Space**	**Notes**
Best	$O(n \log n)$	$O(1)$	All cases same
Average	$O(n \log n)$	$O(1)$	Consistent performance
Worst	$O(n \log n)$	$O(1)$	Guaranteed bound

19.4 Key Properties

- **In-Place Sort**: Requires only $O(1)$ extra space
- **Not Stable**: May change relative order of equal elements
- **Cache Performance**: Poor compared to QuickSort due to non-sequential memory access
- **Guaranteed Performance**: No worst-case $O(n^2)$ scenario (unlike QuickSort)

19.5 Comparison with Other Sorting Algorithms

Algorithm	**Time**	**Space**	**Stable**	**Best For**
HeapSort	$O(n \log n)$	$O(1)$	No	Predictable performance, space constraints
QuickSort	$O(n \log n)$ avg	$O(\log n)$	No	General purpose, cache friendly
MergeSort	$O(n \log n)$	$O(n)$	Yes	Stable sorting, large datasets
IntroSort	$O(n \log n)$	$O(\log n)$	No	C++ std::sort implementation

19.6 Practical Considerations

1. HeapSort operations correspond to priority queue manipulations.
2. Less suitable than MergeSort for external sorting on very large datasets.
3. Common in embedded systems due to minimal space requirements.
4. Difficult to parallelize effectively.

19.7 Common Mistakes

- Incorrect heapify direction: start heapify from leaves (not root) when building.
- Off-by-one errors in child index calculation (using $2i$ instead of $2i + 1$).
- Forgetting to reduce heap size during extraction phase.
- Assuming HeapSort is stable (it is not).

19.8 Real-World Applications

- Memory-constrained environments.
- Real-time systems requiring guaranteed performance.
- Implementing priority queues.
- Systems where worst-case $O(n^2)$ performance is unacceptable.

19.9 Practice Problems

- Sort an Array (LeetCode 912)
- Kth Largest Element in an Array (LeetCode 215)
- Find Median from Data Stream (LeetCode 295)
- Top K Frequent Elements (LeetCode 347)

19.10 Summary

HeapSort's combination of guaranteed $O(n \log n)$ performance and space efficiency makes it valuable in specific domains, though in practice it is often outperformed by hybrid algorithms like IntroSort (used in C++'s `std::sort`).

20 Counting Sort Algorithm: Comprehensive Guide

Counting Sort is a non-comparison based integer sorting algorithm that operates in $O(n+k)$ time, where k is the range of input values, making it extremely efficient for sorting integers within a limited range.

20.1 Core Algorithm

Key Characteristics

- **Integer Sorting**: Only works with integer keys
- **Stable Sort**: Preserves original order of equal elements
- **Linear Time**: $O(n + k)$ time complexity
- **Non-Comparative**: Doesn't compare elements during sorting

Standard Implementation

```cpp
void countingSort(vector<int>& arr) {
    if (arr.empty()) return;

    int max_val = *max_element(arr.begin(), arr.end());
    int min_val = *min_element(arr.begin(), arr.end());
    int range = max_val - min_val + 1;

    vector<int> count(range, 0);
    vector<int> output(arr.size());

    for (int num : arr) {
        count[num - min_val]++;
    }

    for (int i = 1; i < range; i++) {
        count[i] += count[i - 1];
    }

    for (int i = arr.size() - 1; i >= 0; i--) {
        output[count[arr[i] - min_val] - 1] = arr[i];
        count[arr[i] - min_val]--;
    }

    arr = output;
}
```

20.2 Optimizations

A. Positive Integers Only (Simplified)

```cpp
void countingSortPositive(vector<int>& arr) {
    int max_val = *max_element(arr.begin(), arr.end());
    vector<int> count(max_val + 1, 0);

    for (int num : arr) count[num]++;

    int index = 0;
    for (int i = 0; i <= max_val; i++) {
        while (count[i]-- > 0) {
            arr[index++] = i;
        }
    }
}
```

B. In-Place Counting Sort

```cpp
void inPlaceCountingSort(vector<int>& arr) {
    int min_val = *min_element(arr.begin(), arr.end());
    int max_val = *max_element(arr.begin(), arr.end());
    int range = max_val - min_val + 1;

    vector<int> count(range, 0);
    for (int num : arr) count[num - min_val]++;

    int index = 0;
    for (int i = 0; i < range; i++) {
        while (count[i] > 0) {
            arr[index++] = i + min_val;
            count[i]--;
        }
    }
}
```

20.3 Complexity Analysis

Case	Time Complexity	Space Complexity	Notes
Best	$O(n+k)$	$O(n+k)$	k is range of input
Average	$O(n+k)$	$O(n+k)$	Consistent performance
Worst	$O(n+k)$	$O(n+k)$	All cases same

20.4 Key Properties

1. **Integer Sorting**: Only works with discrete values (integers, characters)
2. **Stable Version**: Preserves order of equal elements (important for radix sort)
3. **Linear Time**: Beats $O(n \log n)$ lower bound for comparison sorts
4. **Range Dependent**: Efficiency depends on range k of input values

20.5 Comparison with Other Sorts

Algorithm	Time	Space	Stable	Best For
Counting Sort	$O(n+k)$	$O(n+k)$	Yes	Small integer ranges
QuickSort	$O(n \log n)$ avg	$O(\log n)$	No	General purpose
MergeSort	$O(n \log n)$	$O(n)$	Yes	Large datasets
Radix Sort	$O(d(n+b))$	$O(n+b)$	Yes	Large numbers (d digits)

20.6 Practical Considerations

- **Range Limitation**: Impractical when k is much larger than n
- **Negative Numbers**: Requires offset (handled in standard implementation)
- **Memory Usage**: Can be prohibitive for large ranges
- **Real-world Use**: Often used as a subroutine in Radix Sort

20.7 Common Mistakes

- Ignoring negative numbers
- Index errors in count array (off-by-one)
- Stability violations by implementing unstable versions
- Incorrect range calculation

20.8 Real-World Applications

- Sorting small integer ranges (e.g., test scores 0–100)
- Histogram computation
- Building block for Radix Sort
- Frequency counting problems
- Database indexing for small-range keys

20.9 Practice Problems

- Sort Colors (LeetCode 75)
- Height Checker (LeetCode 1051)
- Relative Sort Array (LeetCode 1122)
- Maximum Gap (LeetCode 164)

Counting Sort's linear time complexity makes it exceptionally powerful for the right problems, though its applicability is limited to specific cases where the input range is appropriately bounded.

21 Radix Sort Algorithm: Comprehensive Guide

Radix Sort is a non-comparative integer sorting algorithm that achieves linear time complexity by processing individual digits or groups of digits. It's particularly efficient for sorting large numbers or strings with fixed-length representations.

21.1 Core Algorithm

Key Characteristics

- **Digit-by-Digit Sorting**: Processes digits from least significant (LSD) to most significant (MSD)
- **Stable Sorting**: Requires stable subroutine sort (typically Counting Sort)
- **Linear Time**: $O(d(n + b))$ where d is digit count, b is base
- **Non-Comparative**: Doesn't compare elements directly

LSD Radix Sort Implementation (Base 10)

```cpp
void countingSort(vector<int>& arr, int exp) {
    vector<int> output(arr.size());
    vector<int> count(10, 0);  // Base 10 digits 0-9

    for (int num : arr) {
        int digit = (num / exp) % 10;
        count[digit]++;
    }

    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    for (int i = arr.size() - 1; i >= 0; i--) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    arr = output;
}

void radixSort(vector<int>& arr) {
    if (arr.empty()) return;

    int max_num = *max_element(arr.begin(), arr.end());

    for (int exp = 1; max_num / exp > 0; exp *= 10) {
        countingSort(arr, exp);
    }
}
```

21.2 Optimizations & Variations

A. MSD Radix Sort (Most Significant Digit First)

```cpp
void msdRadixSort(vector<int>& arr, int left, int right, int exp) {
    if (right <= left || exp <= 0) return;

    vector<int> count(11, 0);  // 0-9 buckets + 1 for indexing
    vector<int> output(right - left + 1);

    for (int i = left; i <= right; i++) {
        int digit = (arr[i] / exp) % 10;
        count[digit + 1]++;
    }

    for (int i = 1; i < 11; i++) {
        count[i] += count[i - 1];
    }

    for (int i = left; i <= right; i++) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit]++] = arr[i];
    }

    for (int i = left; i <= right; i++) {
        arr[i] = output[i - left];
    }

    for (int i = 0; i < 10; i++) {
        msdRadixSort(arr, left + count[i], left + count[i + 1] - 1, exp / 10);
    }
}
```

B. Hybrid Radix Sort (With Insertion Sort)

```
const int INSERTION_THRESHOLD = 16;

void insertionSort(vector<int>& arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void hybridRadixSort(vector<int>& arr) {
    if (arr.size() <= INSERTION_THRESHOLD) {
        insertionSort(arr, 0, arr.size() - 1);
        return;
    }
    radixSort(arr);
}
```

21.3 Complexity Analysis

Case	Time Complexity	Space Complexity	Notes
Best	$O(d(n+b))$	$O(n+b)$	d digits, b base
Average	$O(d(n+b))$	$O(n+b)$	Consistent performance
Worst	$O(d(n+b))$	$O(n+b)$	All cases same

Where:

- n: Number of elements
- d: Number of digits
- b: Base (typically 10 for decimal, 256 for bytes)

21.4 Key Properties

1. **Integer/String Sorting**: Works well for fixed-length data
2. **Stable Version**: Preserves order of equal elements
3. **Linear Time**: When d is constant and b is $O(n)$
4. **Non-Comparative**: Doesn't use element comparisons

21.5 Comparison with Other Sorts

Algorithm	**Time**	**Space**	**Stable**	**Best For**
Radix Sort	$O(d(n+b))$	$O(n+b)$	Yes	Fixed-length keys
QuickSort	$O(n \log n)$ avg	$O(\log n)$	No	General purpose
Counting Sort	$O(n+k)$	$O(n+k)$	Yes	Small integer ranges
MergeSort	$O(n \log n)$	$O(n)$	Yes	Large datasets

21.6 Practical Considerations

- **Base Selection**: Higher bases reduce passes but increase memory
- **Negative Numbers**: Requires special handling (two's complement)
- **Real-world Use**: Used in card sorting machines, string sorting
- **Cache Performance**: Better with smaller base sizes

21.7 Common Mistakes

- Digit extraction errors (incorrect digit calculation)
- Base mismatch between passes
- Negative number handling oversight
- Stability violation using unstable subroutine

21.8 Real-World Applications

- Sorting large datasets of fixed-length records
- DNA sequence sorting (fixed-length strings)
- IP address sorting
- Database indexing for fixed-length keys
- Card sorting machines (original use case)

21.9 Practice Problems

- Maximum Gap (LeetCode 164)
- Largest Number (LeetCode 179)
- Sort Colors (LeetCode 75)
- Custom Sort String (LeetCode 791)

Radix Sort's linear time complexity makes it exceptionally powerful for sorting fixed-length data, though its memory requirements and digit-by-digit processing make it less general-purpose than comparison-based sorts for arbitrary data.

22 Standard Binary Search: Comprehensive Guide

Binary search is an efficient $O(\log n)$ algorithm for finding an element in a **sorted** array by repeatedly dividing the search interval in half.

22.1 Standard Implementation

Iterative Approach

```
int binarySearch(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2; // Avoids overflow

        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1; // Target not found
}
```

Recursive Approach

```
int binarySearchRecursive(vector<int>& nums, int target, int left, int right) {
    if (left > right) return -1;

    int mid = left + (right - left) / 2;

    if (nums[mid] == target) return mid;
    if (nums[mid] < target)
        return binarySearchRecursive(nums, target, mid + 1, right);
    else
        return binarySearchRecursive(nums, target, left, mid - 1);
}
```

22.2 Key Components

1. **Precondition**: Array must be sorted
2. **Three Pointers**:
    - `left` - current start of search space
    - `right` - current end of search space
    - `mid` - middle element being checked
3. **Termination Condition**: `left > right` (search space exhausted)

22.3 Edge Cases & Defensive Programming

```
int binarySearchSafe(vector<int>& nums, int target) {
    if (nums.empty()) return -1;

    int left = 0;
    int right = nums.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}
```

22.4 Common Variations

A. Lower Bound (First Occurrence)

```
int lowerBound(vector<int>& nums, int target) {
    int left = 0, right = nums.size();

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }

    return left; // First index where target could be inserted
}
```

B. Upper Bound (Last Occurrence)

```
int upperBound(vector<int>& nums, int target) {
    int left = 0, right = nums.size();

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] <= target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }

    return left - 1; // Last occurrence of target
}
```

22.5 Time & Space Complexity

Approach	Time	Space
Iterative	$O(\log n)$	$O(1)$
Recursive	$O(\log n)$	$O(\log n)$ stack

22.6 When to Use Binary Search

- Sorted arrays/lists
- Monotonic functions (searching for a specific value)
- Problems where $O(n)$ is too slow
- When you can discard half the search space each step

22.7 Common Mistakes

- **Incorrect Mid Calculation**: Using `(left + right)/2` may overflow
- **Termination Condition**: Wrong loop condition (`<` vs `<=`)
- **Bound Updates**: Forgetting `+1` or `-1` when updating bounds
- **Unsorted Input**: Applying to unsorted data

22.8 Practice Problems

- Binary Search (LeetCode 704)
- Search Insert Position (LeetCode 35)
- First Bad Version (LeetCode 278)
- Find Peak Element (LeetCode 162)

Mastering binary search is fundamental for efficient searching in sorted data and forms the basis for more advanced algorithms like binary search trees and divide-and-conquer approaches.

23 Lower Bound and Upper Bound in Binary Search

Lower bound and upper bound are essential variations of binary search that find the first and last positions of a target value in a sorted array, respectively. These are fundamental for solving range-based search problems efficiently.

23.1 Lower Bound (First Occurrence)

Finds the **first** element not less than the target (leftmost insertion point).

C++ Implementation

```
int lowerBound(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size(); // Note: nums.size() not nums.size()-1

    while (left < right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }

    return left; // Returns nums.size() if all elements are smaller
}
```

Key Points:

- Returns index of first element $\geq$ target
- If target ¿ all elements, returns `nums.size()`
- Uses `left < right` instead of `left <= right`
- `right` starts at `nums.size()` (one past last element)

23.2 Upper Bound (Last Occurrence)

Finds the **first** element greater than the target (rightmost insertion point).

C++ Implementation

```
int upperBound(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size();

    while (left < right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] <= target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }

    return left; // Returns nums.size() if all elements are  target
}
```

Key Points:

- Returns index of first element $>$ target
- To get last occurrence of target, use `upperBound(nums, target) - 1`
- Same structure as lower bound but with `<=` comparison

23.3 STL Equivalents

C++ provides built-in functions in `<algorithm>`:

```cpp
#include <algorithm>

auto lb = lower_bound(nums.begin(), nums.end(), target); // Returns iterator
auto ub = upper_bound(nums.begin(), nums.end(), target); // Returns iterator

int first_pos = lb - nums.begin(); // Convert to index
int last_pos = ub - nums.begin() - 1; // Last occurrence
```

23.4 Practical Applications

A. Count Occurrences of Target

```cpp
int countOccurrences(vector<int>& nums, int target) {
    int first = lowerBound(nums, target);
    int last = upperBound(nums, target) - 1;

    if (first <= last && nums[first] == target) {
        return last - first + 1;
    }
    return 0;
}
```

B. Range Search

```cpp
vector<int> searchRange(vector<int>& nums, int target) {
    int first = lowerBound(nums, target);
    int last = upperBound(nums, target) - 1;

    if (first <= last && nums[first] == target) {
        return {first, last};
    }
    return {-1, -1};
}
```

23.5 Comparison Table

Function	**Returns**	**Condition**	**STL Equivalent**
Lower Bound	First index where element $\geq$ target	`nums[mid] < target`	`std::lower_bound`
Upper Bound	First index where element $>$ target	`nums[mid] <= target`	`std::upper_bound`

Table 1: Comparison of Lower Bound and Upper Bound in Binary Search

23.6 Edge Cases

1. **Empty array**: Returns 0 (lower/upper bound)
2. **All elements smaller than target**: Returns `nums.size()`
3. **All elements larger than target**: Returns 0
4. **Exact match not found**: Returns insertion point

23.7 Common Mistakes

- **Infinite loops**: Using `left <= right` instead of `left < right`
- **Index errors**: Forgetting to subtract 1 for last occurrence
- **Comparison operators**: Mixing `<` and `<=` conditions
- **Initial right value**: Using `nums.size()-1` instead of `nums.size()`

23.8 Practice Problems

- First Bad Version (LeetCode 278) (Lower bound)
- Find First and Last Position (LeetCode 34)
- Search Insert Position (LeetCode 35)
- Count of Smaller Numbers (LeetCode 315)

Mastering lower and upper bounds is crucial for solving many binary search problems efficiently, especially those involving ranges or duplicate values in sorted arrays.

24 Binary Search on Answer: Optimization by Monotonicity

Binary Search on Answer (also known as "Answer Space Binary Search") is a powerful algorithmic technique used to find the optimal value of a solution when direct search is inefficient. It works by leveraging the monotonic nature of the answer space to apply binary search over possible values rather than indices.

24.1 Core Concept

Key Characteristics

- **Applies to Answer Space**: Search is over possible outcomes, not array indices.
- **Monotonic Predicate**: The search space must allow a clear division between "valid" and "invalid" values.
- **Validation-Driven**: Uses a custom `isValid()` function to verify each candidate.
- **Min/Max Optimization**: Suited for problems asking for the minimum or maximum feasible value.

24.2 Standard Template

```
int binarySearchOnAnswer(int min_val, int max_val) {
    int left = min_val, right = max_val, answer = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (isValid(mid)) {
            answer = mid;
            right = mid - 1;  // For minimization
            // left = mid + 1; // For maximization
        } else {
            left = mid + 1;
            // right = mid - 1; // For maximization
        }
    }

    return answer;
}
```

24.3 Common Patterns

A. Minimization Problems

Find the smallest x such that the condition is satisfied.
Example: Book Allocation Problem

```
bool isValid(vector<int>& books, int m, int max_pages) {
    int students = 1, current = 0;
    for (int pages : books) {
        current += pages;
        if (current > max_pages) {
            students++;
            current = pages;
            if (students > m) return false;
        }
    }
    return true;
}

int allocateBooks(vector<int>& books, int m) {
    int left = *max_element(books.begin(), books.end());
    int right = accumulate(books.begin(), books.end(), 0);
    int answer = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (isValid(books, m, mid)) {
            answer = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return answer;
}
```

B. Maximization Problems

Find the largest x such that the condition is satisfied.
Example: Maximum Profit with Limited Capital

```
bool isValid(vector<int>& profits, int k, int min_profit) {
    // Problem-specific validation logic
}

int findMaxProfit(vector<int>& profits, int k) {
    int left = 0;
    int right = *max_element(profits.begin(), profits.end());
    int answer = 0;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (isValid(profits, k, mid)) {
            answer = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return answer;
}
```

24.4 Implementation Checklist

- **Answer Bounds**: Set `left` and `right` to span all possible answers.
- **Efficient `isValid()`**: Should be significantly faster than brute force, ideally $\mathcal{O}(n)$ or better.
- **Track Valid Answers**: Store the latest valid `mid` in `answer`.
- **Loop Until Converged**: Continue while `left` $\leq$ `right`.

24.5 Common Pitfalls

- Incorrect initialization of `left` and `right`.
- Confusing updates for min vs. max search.
- Forgetting to store valid `mid` in `answer`.
- Inefficient or incorrect `isValid()` logic.

24.6 Complexity Analysis

Aspect	**Time Complexity**	**Space Complexity**	**Remarks**
Search Loop	$\mathcal{O}(\log R)$	$\mathcal{O}(1)$	$R =$ `right` $-$ `left`
Validation	$\mathcal{O}(n)$	$\mathcal{O}(1)$	Per call to `isValid()`
Overall	$\mathcal{O}(n \log R)$	$\mathcal{O}(1)$	Efficient for large answer ranges

24.7 Real-World Applications

- **Resource Allocation**: Minimum days/machines to complete tasks.
- **Search Optimization**: Finding optimal threshold values.
- **Game Theory**: Binary search over possible scores.
- **Scheduling**: Minimum feasible load distribution.

24.8 Practice Problems

- Split Array Largest Sum (LeetCode 410)
- Capacity To Ship Packages (LeetCode 1011)
- Koko Eating Bananas (LeetCode 875)
- Minimum Number of Days (LeetCode 1482)

24.9 Summary

Binary Search on Answer is a vital tool in algorithm design for optimization problems. By reducing the solution space via a monotonic check, it transforms brute-force searches into logarithmic efficiency. Understanding when and how to apply this method is crucial for tackling a wide range of interview and competitive programming challenges.

25 Ternary Search: Optimizing Unimodal Functions

Ternary Search is an efficient search algorithm used to find the maximum or minimum of a **unimodal function**—a function that either strictly increases and then decreases (peak) or decreases and then increases (valley). Unlike binary search which splits the domain into two, ternary search splits it into three.

25.1 Core Concept

Unimodal Function Properties

- **Peak Unimodal**: Increases, then decreases
- **Valley Unimodal**: Decreases, then increases
- **Single Extremum**: Only one local/global maximum or minimum

Comparison with Binary Search

Feature	Binary Search	Ternary Search
Applicable To	Monotonic functions	Unimodal functions
Divides Domain Into	2 Parts	3 Parts
Comparisons per Step	1	2
Goal	Find exact value	Find extrema

25.2 Standard Implementations

Finding the Maximum (Continuous Domain)

```
double ternarySearchMax(double left, double right, double eps) {
    while (right - left > eps) {
        double m1 = left + (right - left) / 3;
        double m2 = right - (right - left) / 3;
        if (f(m1) < f(m2))
            left = m1;
        else
            right = m2;
    }
    return (left + right) / 2;
}
```

Finding the Minimum (Continuous Domain)

```
double ternarySearchMin(double left, double right, double eps) {
    while (right - left > eps) {
        double m1 = left + (right - left) / 3;
        double m2 = right - (right - left) / 3;
        if (f(m1) > f(m2))
            left = m1;
        else
            right = m2;
    }
    return (left + right) / 2;
}
```

25.3 Discrete Version (Array Peak Search)

```
int ternaryPeakSearch(const vector<int>& nums) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        int m1 = left + (right - left) / 3;
        int m2 = right - (right - left) / 3;
        if (nums[m1] < nums[m2])
            left = m1 + 1;
        else
```

```
            right = m2 - 1;
    }
    return left; // or right, both converge
}
```

25.4 Implementation Notes

- **Termination Condition**:
  - Continuous: When interval length $< \epsilon$
  - Discrete: When left == right
- **Point Selection**: Divide range into thirds: m_1 and m_2
- **Golden Section Variant**: Uses ϕ (golden ratio) for faster convergence

25.5 Applications

- **Optimization Problems**:
  - Find best parameters in ML, physics simulations
  - Minimize/maximize cost, time, or error
- **Computational Geometry**:
  - Minimum distance from point to curve
  - Maximum height/projection from angle
- **Competitive Programming**:
  - Single-peak array problems
  - Real-number function optimization without derivatives

25.6 Common Pitfalls

- Applying ternary search to non-unimodal functions
- Confusing maximum/minimum conditions
- Incorrect update of m_1 and m_2 values
- Poor ϵ leading to imprecise answers or infinite loops

25.7 Complexity Analysis

Case	**Time Complexity**	**Space Complexity**
Continuous Domain	$O(\log(1/\epsilon))$	$O(1)$
Discrete Domain	$O(\log_3 n)$	$O(1)$

25.8 Practice Problems

- Find Peak Element (LeetCode 162)
- Maximum in Mountain Sequence (LintCode 585)
- Egg Drop With 2 Eggs (LeetCode 1884)
- Minimize Max Distance (LeetCode 774)

25.9 Conclusion

Ternary Search is a powerful tool for unimodal function optimization, especially when derivative-based methods are infeasible. While it involves more comparisons per iteration than binary search, it guarantees convergence to the global extremum for unimodal functions with minimal overhead.

26 Breadth-First Search (BFS)

Breadth-First Search (BFS) is a graph traversal algorithm that explores all neighbors of a node before moving on to their neighbors. It is ideal for finding shortest paths in unweighted graphs and analyzing connected components.

26.1 Standard BFS Implementation

Adjacency List Representation

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

using namespace std;

void bfsTraversal(const vector<vector<int>>& graph, int start) {
    int n = graph.size();
    vector<bool> visited(n, false);
    queue<int> q;

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

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        // Process node u here
        cout << u << " ";

        for (int v : graph[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

Complexity Analysis

- **Time Complexity:** $O(V + E)$
- **Space Complexity:** $O(V)$

26.2 Key Applications

Shortest Path in Unweighted Graphs

```cpp
vector<int> shortestPaths(const vector<vector<int>>& graph, int start) {
    vector<int> distance(graph.size(), -1);
    queue<int> q;

    distance[start] = 0;
    q.push(start);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v : graph[u]) {
            if (distance[v] == -1) {
                distance[v] = distance[u] + 1;
                q.push(v);
            }
        }
    }
    return distance;
}
```

Flood Fill (Matrix Traversal)

```
void floodFill(vector<vector<int>>& image, int sr, int sc, int newColor) {
    int m = image.size(), n = image[0].size();
    int oldColor = image[sr][sc];
    if (oldColor == newColor) return;

    queue<pair<int,int>> q;
    q.push({sr, sc});

    while (!q.empty()) {
        auto [r, c] = q.front();
        q.pop();
        image[r][c] = newColor;

        vector<pair<int,int>> directions = {{-1,0}, {1,0}, {0,-1}, {0,1}};
        for (auto [dr, dc] : directions) {
            int nr = r + dr, nc = c + dc;
            if (nr >= 0 && nr < m && nc >= 0 && nc < n && image[nr][nc] == oldColor) {
                q.push({nr, nc});
            }
        }
    }
}
```

26.3 Advanced Variations

Multi-source BFS

```
vector<int> multiSourceBFS(const vector<vector<int>>& graph, vector<int> sources) {
    vector<int> distance(graph.size(), -1);
    queue<int> q;

    for (int src : sources) {
        distance[src] = 0;
        q.push(src);
    }

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v : graph[u]) {
            if (distance[v] == -1) {
                distance[v] = distance[u] + 1;
                q.push(v);
            }
        }
    }
    return distance;
}
```

Bidirectional BFS

```
int bidirectionalBFS(const vector<vector<int>>& graph, int start, int end) {
    if (start == end) return 0;

    unordered_set<int> visitedA, visitedB;
    queue<int> qA, qB;
    int steps = 0;

    qA.push(start); visitedA.insert(start);
    qB.push(end); visitedB.insert(end);

    while (!qA.empty() && !qB.empty()) {
        steps++;
        for (int i = qA.size(); i > 0; i--) {
            int u = qA.front(); qA.pop();
            for (int v : graph[u]) {
                if (visitedB.count(v)) return steps;
                if (!visitedA.count(v)) {
                    visitedA.insert(v);
                    qA.push(v);
                }
            }
        }

        steps++;
        for (int i = qB.size(); i > 0; i--) {
            int u = qB.front(); qB.pop();
            for (int v : graph[u]) {
                if (visitedA.count(v)) return steps;
                if (!visitedB.count(v)) {
                    visitedB.insert(v);
                    qB.push(v);
                }
            }
        }
    }

    return -1;
}
```

26.4 Common Mistakes to Avoid

- Forgetting to mark nodes as visited before enqueueing (leads to infinite loops)
- Incorrect queue usage (e.g., pushing nodes multiple times)
- Using DFS instead of BFS for shortest path problems
- Not accounting for disconnected components

26.5 Practice Problems

- Number of Islands (LeetCode 200)
- Rotting Oranges (LeetCode 994)
- Word Ladder (LeetCode 127)
- 01 Matrix (LeetCode 542)

26.6 Conclusion

BFS is an essential algorithm in graph theory, especially for unweighted shortest paths and level-order traversal problems. Understanding both the base implementation and its variations is critical for solving a wide range of algorithmic challenges.

27 Depth-First Search (DFS)

Depth-First Search (DFS) explores as far as possible along each branch before backtracking. It is particularly useful for problems such as topological sorting, cycle detection, and pathfinding.

27.1 Standard DFS Implementations

Recursive Implementation (Adjacency List)

```cpp
#include <vector>
#include <unordered_set>

using namespace std;

void dfsRecursive(const vector<vector<int>>& graph, int node, vector<bool>& visited) {
    visited[node] = true;
    // Process node here (pre-order)

    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            dfsRecursive(graph, neighbor, visited);
        }
    }

    // Process node here (post-order)
}

void dfsTraversal(const vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false);
    dfsRecursive(graph, start, visited);
}
```

Iterative Implementation (Using Stack)

```cpp
void dfsIterative(const vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false);
    stack<int> s;
    s.push(start);

    while (!s.empty()) {
        int node = s.top();
        s.pop();

        if (!visited[node]) {
            visited[node] = true;
            // Process node here

            for (auto it = graph[node].rbegin(); it != graph[node].rend(); ++it) {
                if (!visited[*it]) {
                    s.push(*it);
                }
            }
        }
    }
}
```

27.2 Key Applications

Connected Components

```cpp
int countComponents(const vector<vector<int>>& graph) {
    vector<bool> visited(graph.size(), false);
    int components = 0;

    for (int i = 0; i < graph.size(); i++) {
        if (!visited[i]) {
            components++;
            dfsRecursive(graph, i, visited);
        }
    }
    return components;
}
```

Cycle Detection in Undirected Graphs

```
bool hasCycleUndirected(const vector<vector<int>>& graph) {
    vector<bool> visited(graph.size(), false);

    for (int i = 0; i < graph.size(); i++) {
        if (!visited[i] && hasCycleDFS(graph, i, -1, visited)) {
            return true;
        }
    }
    return false;
}

bool hasCycleDFS(const vector<vector<int>>& graph, int node, int parent, vector<bool>& visited) {
    visited[node] = true;

    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            if (hasCycleDFS(graph, neighbor, node, visited))
                return true;
        }
        else if (neighbor != parent) {
            return true;
        }
    }
    return false;
}
```

Topological Sorting

```
vector<int> topologicalSort(const vector<vector<int>>& graph) {
    vector<bool> visited(graph.size(), false);
    vector<int> result;

    for (int i = 0; i < graph.size(); i++) {
        if (!visited[i]) {
            topologicalDFS(graph, i, visited, result);
        }
    }

    reverse(result.begin(), result.end());
    return result;
}

void topologicalDFS(const vector<vector<int>>& graph, int node, vector<bool>& visited, vector<int>& result) {
    visited[node] = true;

    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            topologicalDFS(graph, neighbor, visited, result);
        }
    }

    result.push_back(node); // Post-order processing
}
```

27.3 Advanced Variations

Path Finding with Backtracking

```
vector<vector<int>> allPaths(const vector<vector<int>>& graph, int start, int end) {
    vector<vector<int>> paths;
    vector<int> current;
    vector<bool> visited(graph.size(), false);

    allPathsDFS(graph, start, end, visited, current, paths);
    return paths;
}

void allPathsDFS(const vector<vector<int>>& graph, int node, int end,
                vector<bool>& visited, vector<int>& current,
                vector<vector<int>>& paths) {
    visited[node] = true;
    current.push_back(node);

    if (node == end) {
        paths.push_back(current);
    } else {
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                allPathsDFS(graph, neighbor, end, visited, current, paths);
            }
        }
    }

    visited[node] = false;
    current.pop_back();
}
```

Articulation Points (Cut Vertices)

```
vector<int> findArticulationPoints(const vector<vector<int>>& graph) {
    vector<int> disc(graph.size(), -1), low(graph.size(), -1);
    vector<bool> isAP(graph.size(), false);
    int time = 0;

    for (int i = 0; i < graph.size(); i++) {
        if (disc[i] == -1) {
            articulationDFS(graph, i, -1, disc, low, isAP, time);
        }
    }

    vector<int> result;
    for (int i = 0; i < graph.size(); i++) {
        if (isAP[i]) result.push_back(i);
    }
    return result;
}

void articulationDFS(const vector<vector<int>>& graph, int u, int parent,
                    vector<int>& disc, vector<int>& low,
                    vector<bool>& isAP, int& time) {
    disc[u] = low[u] = ++time;
    int children = 0;

    for (int v : graph[u]) {
        if (disc[v] == -1) {
            children++;
            articulationDFS(graph, v, u, disc, low, isAP, time);
            low[u] = min(low[u], low[v]);

            if (parent != -1 && low[v] >= disc[u]) {
                isAP[u] = true;
            }
        } else if (v != parent) {
            low[u] = min(low[u], disc[v]);
        }
    }

    if (parent == -1 && children > 1) {
        isAP[u] = true;
    }
}
```

27.4 Common Mistakes to Avoid

- Stack overflow on deep recursion (consider iterative DFS)
- Forgetting to mark nodes as visited
- Not handling disconnected graphs
- Incorrect parent tracking during cycle detection
- Applying topological sort to cyclic graphs (works only for DAGs)

27.5 Practice Problems

- Number of Islands (LeetCode 200)
- Course Schedule (LeetCode 207)
- Longest Increasing Path (LeetCode 329)
- Critical Connections (LeetCode 1192)

27.6 Conclusion

DFS is a foundational algorithm for exploring graphs. It supports a wide variety of applications from simple traversals to detecting structural properties of graphs. Mastery of both recursive and iterative approaches is crucial for algorithmic problem-solving.

28 Dijkstra's Algorithm for Shortest Path

Dijkstra's algorithm is a greedy algorithm that finds the shortest paths from a single source node to all other nodes in a weighted graph with **non-negative edge weights**.

28.1 Core Algorithm

Key Properties

- **Time Complexity**: $\mathcal{O}((V + E) \log V)$ with priority queue
- **Space Complexity**: $\mathcal{O}(V)$
- **Optimal for**: Weighted graphs with non-negative edges
- **Guarantees**: Finds shortest paths when no negative weights exist

Standard Implementation (C++ with Priority Queue)

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

using namespace std;

vector<int> dijkstra(const vector<vector<pair<int, int>>>& graph, int start) {
    int n = graph.size();
    vector<int> dist(n, INT_MAX);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    dist[start] = 0;
    pq.push({0, start});

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

        if (current_dist > dist[u]) continue;

        for (auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;

            if (dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

28.2 Path Reconstruction

To reconstruct the actual shortest path (not just distances):

```cpp
vector<int> reconstructPath(int start, int end,
                            const vector<int>& dist,
                            const vector<vector<pair<int, int>>>& graph) {
    vector<int> path;
    if (dist[end] == INT_MAX) return path;

    int current = end;
    path.push_back(current);

    while (current != start) {
        for (auto& edge : graph[current]) {
            int neighbor = edge.first;
            int weight = edge.second;
            if (dist[current] == dist[neighbor] + weight) {
                current = neighbor;
                path.push_back(current);
                break;
            }
        }
    }

    reverse(path.begin(), path.end());
    return path;
}
```

28.3 Optimizations

A. Early Termination

```
// Add inside the main loop:
if (u == target) break;
```

B. Fibonacci Heap (Theoretical)

- Reduces time to $\mathcal{O}(E + V \log V)$
- Not practical in contests/interviews

28.4 Comparison with Other Algorithms

Algorithm	Time	Negative Weights	Best For
Dijkstra	$\mathcal{O}((V + E) \log V)$	No	Non-negative weights
Bellman-Ford	$\mathcal{O}(VE)$	Yes	Negative weights, detection
Floyd-Warshall	$\mathcal{O}(V^3)$	Yes	All-pairs shortest paths
A*	$\mathcal{O}((V + E) \log V)$	No	Heuristic-guided search

28.5 Common Mistakes

1. Using Dijkstra with negative edge weights
2. Incorrect min-heap configuration
3. Forgetting `current_dist > dist[u]` check
4. Misconstructing adjacency list

28.6 Practice Problems

- Network Delay Time (LeetCode 743)
- Cheapest Flights Within K Stops (LeetCode 787)
- Path With Maximum Probability (LeetCode 1514)
- Minimum Cost to Reach Destination (LeetCode 1928)

28.7 Pro Tips

- Use `vector<vector<pair<int, int>>>` for efficient adjacency representation
- Always use a min-heap (with `greater<>` comparator)
- For sparse graphs, Dijkstra is often faster than Floyd-Warshall
- Track predecessors to reconstruct paths efficiently

Conclusion: Dijkstra's algorithm remains the most efficient tool for shortest path problems in graphs with non-negative weights. It underpins many real-world systems in routing and navigation.

29 Bellman-Ford Algorithm for Shortest Path

The Bellman-Ford algorithm computes shortest paths from a single source vertex to all other vertices in a weighted graph, even with **negative edge weights** (but no negative cycles reachable from the source).

29.1 Core Algorithm

Key Properties

- **Time Complexity**: $\mathcal{O}(VE)$
- **Space Complexity**: $\mathcal{O}(V)$
- **Handles**: Negative edge weights (detects negative cycles)
- **Works for**: Directed and undirected graphs (treat undirected edges as two directed ones)

Standard Implementation (C++)

```cpp
#include <vector>
#include <climits>

using namespace std;

vector<int> bellmanFord(const vector<vector<pair<int, int>>>& graph, int start) {
    int n = graph.size();
    vector<int> dist(n, INT_MAX);
    dist[start] = 0;

    for (int i = 0; i < n - 1; ++i) {
        for (int u = 0; u < n; ++u) {
            if (dist[u] == INT_MAX) continue;
            for (auto& edge : graph[u]) {
                int v = edge.first;
                int weight = edge.second;
                if (dist[v] > dist[u] + weight) {
                    dist[v] = dist[u] + weight;
                }
            }
        }
    }

    for (int u = 0; u < n; ++u) {
        if (dist[u] == INT_MAX) continue;
        for (auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (dist[v] > dist[u] + weight) {
                return {};
            }
        }
    }

    return dist;
}
```

29.2 Path Reconstruction

To reconstruct the shortest path and detect negative cycles:

```cpp
vector<int> reconstructPath(int start, int end,
                            const vector<int>& dist,
                            const vector<int>& parent) {
    if (dist[end] == INT_MAX) return {};

    vector<int> path;
    for (int v = end; v != start; v = parent[v]) {
        if (v == -1) return {};
        path.push_back(v);
    }
    path.push_back(start);
    reverse(path.begin(), path.end());
    return path;
}
```

29.3 Optimizations

A. Early Stopping

```
bool relaxed = true;
for (int i = 0; i < n - 1 && relaxed; ++i) {
    relaxed = false;
    for (int u = 0; u < n; ++u) {
        if (dist[u] == INT_MAX) continue;
        for (auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
                parent[v] = u;
                relaxed = true;
            }
        }
    }
}
```

B. SPFA (Shortest Path Faster Algorithm)
A queue-based optimization that often performs better in practice:

```
vector<int> spfa(const vector<vector<pair<int, int>>>& graph, int start) {
    int n = graph.size();
    vector<int> dist(n, INT_MAX);
    vector<bool> inQueue(n, false);
    queue<int> q;

    dist[start] = 0;
    q.push(start);
    inQueue[start] = true;

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inQueue[u] = false;

        for (auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
                if (!inQueue[v]) {
                    q.push(v);
                    inQueue[v] = true;
                }
            }
        }
    }

    return dist;
}
```

29.4 4. Comparison with Other Algorithms

Algorithm	Time	Negative Weights	Detects Cycles	Best For
Bellman-Ford	$\mathcal{O}(VE)$	Yes	Yes	Graphs with negative weights
Dijkstra	$\mathcal{O}((V+E)\log V)$	No	No	Non-negative weights
SPFA	$\mathcal{O}(VE)$ avg	Yes	Yes	Practical optimization
Floyd-Warshall	$\mathcal{O}(V^3)$	Yes	Yes	All-pairs shortest paths

29.5 Common Mistakes

1. Not relaxing all edges $V - 1$ times
2. Skipping negative cycle detection
3. Forgetting to initialize $dist[start] = 0$
4. Not storing parent pointers during relaxation

29.6 Practice Problems

- Cheapest Flights Within K Stops (LeetCode 787)
- Network Delay Time (LeetCode 743)
- Currency Arbitrage (GFG)
- Path With Minimum Effort (LeetCode 1631)

29.7 Pro Tips

- Always check for negative cycles after $V - 1$ relaxations
- Use edge list for simpler edge iteration in some cases
- SPFA often performs faster in sparse or practical graphs
- Track parent nodes to recover paths efficiently

Conclusion: The Bellman-Ford algorithm is essential for graphs with negative weights and is the foundation for many advanced algorithms in graph theory and network analysis.

30 Floyd-Warshall Algorithm for All-Pairs Shortest Paths

The Floyd-Warshall algorithm computes shortest paths between **all pairs** of vertices in a weighted graph, handling **negative edge weights** (but no negative cycles).

30.1 Core Algorithm

Key Properties

- **Time Complexity**: $\mathcal{O}(V^3)$
- **Space Complexity**: $\mathcal{O}(V^2)$
- **Handles**: Negative weights (detects negative cycles)
- **Works for**: Both directed and undirected graphs

Standard Implementation (C++)

```cpp
#include <vector>
#include <climits>

using namespace std;

vector<vector<int>> floydWarshall(const vector<vector<int>>& graph) {
    int n = graph.size();
    vector<vector<int>> dist(n, vector<int>(n, INT_MAX));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == j) dist[i][j] = 0;
            else if (graph[i][j] != 0) dist[i][j] = graph[i][j];
        }
    }

    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }

    for (int i = 0; i < n; ++i) {
        if (dist[i][i] < 0) return {};
    }

    return dist;
}
```

30.2 Path Reconstruction

To reconstruct the actual shortest paths (not just distances):

```cpp
vector<vector<int>> next(n, vector<int>(n, -1));

// Initialize next matrix
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (graph[i][j] != INT_MAX) {
            next[i][j] = j;
        }
    }
}

// During DP update
if (dist[i][k] + dist[k][j] < dist[i][j]) {
    dist[i][j] = dist[i][k] + dist[k][j];
    next[i][j] = next[i][k];
}

// Reconstruct path
vector<int> getPath(int u, int v, const vector<vector<int>>& next) {
    if (next[u][v] == -1) return {};
    vector<int> path = {u};
    while (u != v) {
        u = next[u][v];
        path.push_back(u);
    }
    return path;
}
```

30.3 Optimizations

A. Space Optimization (2D → 1D)

```
vector<int> dist(n, INT_MAX);
// Requires careful overwriting order
```

B. Early Termination for Negative Cycle Detection

```
for (int i = 0; i < n; ++i) {
    if (dist[i][i] < 0) return {};
}
```

30.4 Comparison with Other Algorithms

Algorithm	Time	Space	Negative Weights	Best For
Floyd-Warshall	$\mathcal{O}(V^3)$	$\mathcal{O}(V^2)$	Yes	Dense graphs, all-pairs
Dijkstra (all pairs)	$\mathcal{O}(V(E+V)\log V)$	$\mathcal{O}(V^2)$	No	Sparse graphs
Bellman-Ford (all pairs)	$\mathcal{O}(V^2E)$	$\mathcal{O}(V^2)$	Yes	Sparse graphs with negatives
Johnson's	$\mathcal{O}(VE + V^2\log V)$	$\mathcal{O}(V^2)$	Yes	Sparse graphs with negatives

30.5 Common Mistakes

1. Incorrect initialization: Not setting diagonal to 0
2. Wrong loop order: Must be k, then i, then j
3. Integer overflow: Not guarding against INT_MAX + something
4. Cycle check: Only works with diagonal values

30.6 Practice Problems

- Find the City With Smallest Number of Neighbors (LeetCode 1334)
- Network Delay Time (LeetCode 743)
- Currency Arbitrage (LeetCode Discuss)
- Cheapest Flights Within K Stops (LeetCode 787)

30.7 Pro Tips

- Use for dense graphs: Outperforms Dijkstra when $E \approx V^2$
- Negative cycle detection: Check if $dist[i][i] < 0$
- Path queries: Store next matrix for $O(V)$ reconstructions

- Parallelizable: The k-loop can often be run in parallel

Conclusion: Floyd-Warshall is the default choice for dense graphs and all-pairs shortest paths, especially when negative weights are involved. Johnson's algorithm may be preferred for sparse graphs.

31 Floyd-Warshall Algorithm for All-Pairs Shortest Paths

The Floyd-Warshall algorithm computes shortest paths between **all pairs** of vertices in a weighted graph, handling **negative edge weights** (but no negative cycles).

31.1 Core Algorithm

Key Properties

- **Time Complexity**: $\mathcal{O}(V^3)$
- **Space Complexity**: $\mathcal{O}(V^2)$
- **Handles**: Negative weights (detects negative cycles)
- **Works for**: Both directed and undirected graphs

Standard Implementation (C++)

```cpp
#include <vector>
#include <climits>

using namespace std;

vector<vector<int>> floydWarshall(const vector<vector<int>>& graph) {
    int n = graph.size();
    vector<vector<int>> dist(n, vector<int>(n, INT_MAX));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == j) dist[i][j] = 0;
            else if (graph[i][j] != 0) dist[i][j] = graph[i][j];
        }
    }

    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }

    for (int i = 0; i < n; ++i) {
        if (dist[i][i] < 0) return {};
    }

    return dist;
}
```

31.2 Path Reconstruction

To reconstruct the actual shortest paths (not just distances):

```cpp
vector<vector<int>> next(n, vector<int>(n, -1));

// Initialize next matrix
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (graph[i][j] != INT_MAX) {
            next[i][j] = j;
        }
    }
}

// During DP update
if (dist[i][k] + dist[k][j] < dist[i][j]) {
    dist[i][j] = dist[i][k] + dist[k][j];
    next[i][j] = next[i][k];
}

// Reconstruct path
vector<int> getPath(int u, int v, const vector<vector<int>>& next) {
    if (next[u][v] == -1) return {};
    vector<int> path = {u};
    while (u != v) {
        u = next[u][v];
        path.push_back(u);
    }
    return path;
}
```

31.3 Optimizations

A. Space Optimization (2D → 1D)

```
vector<int> dist(n, INT_MAX);
// Requires careful overwriting order
```

B. Early Termination for Negative Cycle Detection

```
for (int i = 0; i < n; ++i) {
    if (dist[i][i] < 0) return {};
}
```

31.4 Comparison with Other Algorithms

Algorithm	Time	Space	Negative Weights	Best For
Floyd-Warshall	$\mathcal{O}(V^3)$	$\mathcal{O}(V^2)$	Yes	Dense graphs, all-pairs
Dijkstra (all pairs)	$\mathcal{O}(V(E+V)\log V)$	$\mathcal{O}(V^2)$	No	Sparse graphs
Bellman-Ford (all pairs)	$\mathcal{O}(V^2E)$	$\mathcal{O}(V^2)$	Yes	Sparse graphs with negatives
Johnson's	$\mathcal{O}(VE+V^2\log V)$	$\mathcal{O}(V^2)$	Yes	Sparse graphs with negatives

31.5 Common Mistakes

1. Incorrect initialization: Not setting diagonal to 0
2. Wrong loop order: Must be k, then i, then j
3. Integer overflow: Not guarding against INT_MAX + something
4. Cycle check: Only works with diagonal values

31.6 Practice Problems

- Find the City With Smallest Number of Neighbors (LeetCode 1334)
- Network Delay Time (LeetCode 743)
- Currency Arbitrage (LeetCode Discuss)
- Cheapest Flights Within K Stops (LeetCode 787)

31.7 Pro Tips

- Use for dense graphs: Outperforms Dijkstra when $E \approx V^2$
- Negative cycle detection: Check if $dist[i][i] < 0$
- Path queries: Store next matrix for $O(V)$ reconstructions

- Parallelizable: The k-loop can often be run in parallel

Conclusion: Floyd-Warshall is the default choice for dense graphs and all-pairs shortest paths, especially when negative weights are involved. Johnson's algorithm may be preferred for sparse graphs.

32 Kruskal's Algorithm for Minimum Spanning Trees (MST)

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree (MST) for a connected, undirected graph with weighted edges. It adds edges in increasing weight order, avoiding cycles.

32.1 Core Algorithm

Key Properties

- **Time Complexity**: $\mathcal{O}(E \log E)$ or $\mathcal{O}(E \log V)$
- **Space Complexity**: $\mathcal{O}(V)$ (for Union-Find)
- **Optimal for**: Sparse graphs
- **Requires**: Union-Find (Disjoint Set Union, DSU)

Implementation (C++) with DSU

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

using namespace std;

class DSU {
    vector<int> parent, rank;
public:
    DSU(int n) : parent(n), rank(n, 0) {
        for (int i = 0; i < n; ++i)
            parent[i] = i;
    }

    int find(int u) {
        if (parent[u] != u)
            parent[u] = find(parent[u]); // Path compression
        return parent[u];
    }

    bool unionSets(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);
        if (rootU == rootV) return false;

        if (rank[rootU] > rank[rootV]) {
            parent[rootV] = rootU;
        } else {
            parent[rootU] = rootV;
            if (rank[rootU] == rank[rootV])
                rank[rootV]++;
        }
        return true;
    }
};

vector<vector<int>> kruskalMST(int n, vector<vector<int>>& edges) {
    sort(edges.begin(), edges.end(),
        [](const vector<int>& a, const vector<int>& b) {
            return a[2] < b[2];
        });

    DSU dsu(n);
    vector<vector<int>> mst;
    int mstWeight = 0;

    for (auto& edge : edges) {
        int u = edge[0], v = edge[1], weight = edge[2];
        if (dsu.unionSets(u, v)) {
            mst.push_back(edge);
            mstWeight += weight;
            if (mst.size() == n - 1) break;
        }
    }

    return mst;
}
```

32.2 Key Components

- **Edge Sorting**: All edges sorted by weight, $\mathcal{O}(E \log E)$
- **Union-Find**:
  - `find()` with path compression ($\mathcal{O}(\alpha(V))$ amortized)
  - `unionSets()` with union by rank
- **Cycle Detection**: Ensures no cycles by checking connected components

32.3 Optimizations

A. Early Termination

```
if (mst.size() == n - 1) break;
```

B. Edge Filtering: Pre-filter edges that obviously cannot be part of MST (e.g., duplicates or heavy parallel edges)

32.4 Comparison with Prim's Algorithm

Feature	Kruskal's	Prim's
Time	$\mathcal{O}(E \log E)$	$\mathcal{O}(E \log V)$
Best For	Sparse graphs	Dense graphs
Data Structure	Union-Find	Priority Queue
Edge Handling	Sorts all edges	Uses adjacent edges

32.5 Applications

- Network design: telephone, electrical, or water systems
- Approximation of NP-hard problems (e.g., TSP)
- Cluster analysis
- Image segmentation

32.6 Common Mistakes

1. Forgetting to sort edges
2. Incorrect DSU (missing path compression or rank optimization)
3. Assuming graph is connected — may result in forest
4. Integer overflow when summing weights

32.7 Practice Problems

- Min Cost to Connect All Points (LeetCode 1584)
- Connecting Cities With Minimum Cost (LeetCode 1135)
- Water Distribution in Village (LeetCode 1168)
- Network Delay Time (LeetCode 743)

32.8 Pro Tips

- **Edge Preprocessing**: Keep only the lightest edge between two nodes
- **Parallel Sorting**: Can be used for very large edge sets
- **DSU Optimization**: Always use both path compression and union by rank
- **Disconnected Graphs**: Kruskal's produces a forest; DSU can help count components

Conclusion: Kruskal's algorithm is a highly efficient and elegant method for MSTs in sparse graphs, highlighting the power of union-find data structures in graph theory.

33 Prim's Algorithm for Minimum Spanning Trees (MST)

Prim's algorithm is a greedy method to find a minimum spanning tree (MST) for a connected, undirected graph. It grows the MST by repeatedly selecting the shortest edge from the current tree to a new vertex.

33.1 Core Algorithm

Key Properties

- **Time Complexity**: $\mathcal{O}(E \log V)$ using a priority queue
- **Space Complexity**: $\mathcal{O}(V + E)$
- **Optimal for**: Dense graphs ($E \approx V^2$)
- **Approach**: Vertex-based greedy algorithm

Implementation (C++ with Priority Queue)

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

using namespace std;

vector<vector<pair<int, int>>> primMST(const vector<vector<pair<int, int>>>& graph) {
    int n = graph.size();
    vector<bool> inMST(n, false);
    vector<int> key(n, INT_MAX);
    vector<int> parent(n, -1);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    key[0] = 0;
    pq.push({0, 0});

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

        if (inMST[u]) continue;
        inMST[u] = true;

        for (auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;

            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = u;
                pq.push({key[v], v});
            }
        }
    }

    vector<vector<pair<int, int>>> mst(n);
    for (int i = 1; i < n; i++) {
        if (parent[i] != -1) {
            mst[parent[i]].push_back({i, key[i]});
            mst[i].push_back({parent[i], key[i]});
        }
    }

    return mst;
}
```

33.2 Key Components

- **Priority Queue**: Picks the next minimum-weight edge
- **Key Array**: Stores minimum edge weights to each vertex

- **In-MST Array**: Tracks inclusion of vertices in MST
- **Parent Array**: Records tree structure

33.3 Optimizations

A. Dense Graphs ($\mathcal{O}(V^2)$ Version)

```
vector<vector<pair<int, int>>> primDense(const vector<vector<pair<int, int>>>& graph) {
    int n = graph.size();
    vector<bool> inMST(n, false);
    vector<int> key(n, INT_MAX);
    vector<int> parent(n, -1);

    key[0] = 0;

    for (int count = 0; count < n; count++) {
        int u = -1;
        for (int v = 0; v < n; v++) {
            if (!inMST[v] && (u == -1 || key[v] < key[u]))
                u = v;
        }

        inMST[u] = true;

        for (auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = u;
            }
        }
    }

    // Construct MST (same logic as before)
}
```

B. Early Termination

```
if (mstEdges == n - 1) break;
```

33.4 Comparison with Kruskal's Algorithm

Feature	**Prim's**	**Kruskal's**
Time	$\mathcal{O}(E \log V)$	$\mathcal{O}(E \log E)$
Best For	Dense graphs	Sparse graphs
Data Structure	Priority Queue	Union-Find
Approach	Vertex-based	Edge-based

33.5 Applications

- Network design (e.g., cable, internet, roads)
- Cluster analysis
- Heuristics for NP-hard problems
- Image segmentation

33.6 Common Mistakes

1. Forgetting to check `inMST` in the priority queue
2. Failing to update both `key` and `parent`
3. Applying to disconnected graphs — Prim's requires connectivity
4. Using `int` with large weights — risk of overflow

33.7 Practice Problems

- Min Cost to Connect All Points (LeetCode 1584)
- Connecting Cities With Minimum Cost (LeetCode 1135)
- Water Distribution in Village (LeetCode 1168)
- Network Delay Time (LeetCode 743)

33.8 Pro Tips

- Use adjacency lists for sparse graphs
- Fibonacci heap gives $\mathcal{O}(E + V \log V)$ — but rarely practical
- Use $\mathcal{O}(V^2)$ version for very dense graphs
- Keep only the lightest edge between two vertices

Conclusion: Prim's algorithm is well-suited for dense graphs due to its vertex-based approach and efficient use of a priority queue. Its conceptual simplicity and strong practical performance make it a go-to choice in many MST applications.

34 Kosaraju's Algorithm for Strongly Connected Components (SCCs)

Kosaraju's algorithm finds all strongly connected components (SCCs) in a directed graph in linear time $\mathcal{O}(V + E)$. An SCC is a maximal set of vertices where each vertex is reachable from every other vertex in the same set.

34.1 Core Algorithm

Key Steps

1. **First Pass DFS**: Perform DFS on the original graph, pushing vertices onto a stack in order of finishing time.
2. **Transpose Graph**: Reverse all edges of the graph.
3. **Second Pass DFS**: Process nodes from the stack on the transposed graph to extract SCCs.

Implementation (C++)

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

using namespace std;

void dfsFirstPass(int u, const vector<vector<int>>& graph, vector<bool>& visited, stack<int>& st) {
    visited[u] = true;
    for (int v : graph[u]) {
        if (!visited[v]) {
            dfsFirstPass(v, graph, visited, st);
        }
    }
    st.push(u);
}

void dfsSecondPass(int u, const vector<vector<int>>& graph, vector<bool>& visited, vector<int>& component) {
    visited[u] = true;
    component.push_back(u);
    for (int v : graph[u]) {
        if (!visited[v]) {
            dfsSecondPass(v, graph, visited, component);
        }
    }
}

vector<vector<int>> kosarajuSCC(const vector<vector<int>>& graph) {
    int n = graph.size();
    stack<int> st;
    vector<bool> visited(n, false);

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfsFirstPass(i, graph, visited, st);
        }
    }

    vector<vector<int>> transpose(n);
    for (int u = 0; u < n; u++) {
        for (int v : graph[u]) {
            transpose[v].push_back(u);
        }
    }

    fill(visited.begin(), visited.end(), false);
    vector<vector<int>> sccs;

    while (!st.empty()) {
        int u = st.top(); st.pop();
        if (!visited[u]) {
            vector<int> component;
            dfsSecondPass(u, transpose, visited, component);
            sccs.push_back(component);
        }
    }

    return sccs;
}
```

34.2 Key Optimizations

A. Iterative DFS to Prevent Stack Overflow

```
void dfsIterative(int u, const vector<vector<int>>& graph, vector<bool>& visited, stack<int>& st) {
    stack<pair<int, bool>> s;
    s.push({u, false});

    while (!s.empty()) {
        auto [node, processed] = s.top();
        s.pop();

        if (processed) {
            st.push(node);
            continue;
        }

        if (visited[node]) continue;
        visited[node] = true;

        s.push({node, true});

        for (auto it = graph[node].rbegin(); it != graph[node].rend(); ++it) {
            if (!visited[*it]) {
                s.push({*it, false});
            }
        }
    }
}
```

B. Component Tracking Using IDs

```
vector<int> componentId(n, -1);
int currentId = 0;

void dfsSecondPass(int u, const vector<vector<int>>& graph, vector<bool>& visited, vector<int>& componentId, int id) {
    visited[u] = true;
    componentId[u] = id;
    for (int v : graph[u]) {
        if (!visited[v]) {
            dfsSecondPass(v, graph, visited, componentId, id);
        }
    }
}

int componentCount = 0;
while (!st.empty()) {
    int u = st.top(); st.pop();
    if (!visited[u]) {
        dfsSecondPass(u, transpose, visited, componentId, componentCount++);
    }
}
```

34.3 Complexity Analysis

Step	**Time Complexity**	**Space Complexity**
First DFS Pass	$\mathcal{O}(V+E)$	$\mathcal{O}(V)$
Transpose Graph	$\mathcal{O}(V+E)$	$\mathcal{O}(V+E)$
Second DFS Pass	$\mathcal{O}(V+E)$	$\mathcal{O}(V)$
Total	$\mathcal{O}(V+E)$	$\mathcal{O}(V+E)$

34.4 Comparison with Tarjan's Algorithm

Feature	**Kosaraju's**	**Tarjan's**
Time	$\mathcal{O}(V + E)$	$\mathcal{O}(V + E)$
Passes	2 DFS passes	1 DFS pass
Extra Space	Stack + Transpose	Stack + Low-link array
Ease of Implementation	Simpler	More complex

34.5 Applications

- Compiler Optimization (e.g., call graph analysis)
- Social Networks (e.g., community detection)
- Circuit Design (e.g., feedback loop detection)
- Web Crawling (e.g., link clustering)
- Formal Verification (e.g., model checking)

34.6 Common Mistakes

1. Not reversing all edges during transposition
2. Incorrect stack usage in the second pass
3. Forgetting to reset the `visited` array
4. Failing to maintain or assign component IDs

34.7 Practice Problems

- Strongly Connected Components (Kosaraju's) - GFG
- Course Schedule II (LeetCode 210)
- Critical Connections (LeetCode 1192)
- Largest Color Value in Graph (LeetCode 1857)

34.8 Pro Tips

- Use adjacency lists for sparse graphs
- Prefer iterative DFS for deep graphs
- Build a meta-graph of SCCs for further analysis
- First pass can be parallelized with care

Conclusion: Kosaraju's algorithm offers a clean and intuitive method for finding SCCs. Its simplicity and linear runtime make it ideal for many real-world applications despite the two-pass requirement.

35 Tarjan's Algorithm for Strongly Connected Components (SCCs)

Tarjan's algorithm is a single-pass DFS-based method to find all strongly connected components in a directed graph, with optimal $\mathcal{O}(V + E)$ time complexity. It's more space-efficient than Kosaraju's as it doesn't require graph transposition.

35.1 Core Algorithm

Key Data Structures

- `disc[u]`: Discovery time of vertex u (also acts as a visited flag)
- `low[u]`: Earliest visited vertex reachable from u
- `stack`: Tracks vertices in current SCC
- `onStack`: Boolean array indicating if vertex is in the stack

C++ Implementation

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

using namespace std;

void tarjanDFS(int u, const vector<vector<int>>& graph, vector<int>& disc,
               vector<int>& low, stack<int>& st, vector<bool>& onStack,
               vector<vector<int>>& sccs, int& time) {
    disc[u] = low[u] = ++time;
    st.push(u);
    onStack[u] = true;

    for (int v : graph[u]) {
        if (disc[v] == -1) {
            tarjanDFS(v, graph, disc, low, st, onStack, sccs, time);
            low[u] = min(low[u], low[v]);
        } else if (onStack[v]) {
            low[u] = min(low[u], disc[v]);
        }
    }

    if (low[u] == disc[u]) {
        vector<int> component;
        while (true) {
            int v = st.top(); st.pop();
            onStack[v] = false;
            component.push_back(v);
            if (v == u) break;
        }
        sccs.push_back(component);
    }
}

vector<vector<int>> tarjanSCC(const vector<vector<int>>& graph) {
    int n = graph.size();
    vector<int> disc(n, -1), low(n, -1);
    vector<bool> onStack(n, false);
    stack<int> st;
    vector<vector<int>> sccs;
    int time = 0;

    for (int i = 0; i < n; ++i) {
        if (disc[i] == -1) {
            tarjanDFS(i, graph, disc, low, st, onStack, sccs, time);
        }
    }
    return sccs;
}
```

35.2 Key Optimizations

Iterative Version (Avoids Stack Overflow)

```
void tarjanIterative(const vector<vector<int>>& graph, vector<vector<int>>& sccs) {
    int n = graph.size();
    vector<int> disc(n, -1), low(n, -1);
    vector<bool> onStack(n, false);
    stack<int> st, dfs_st;
    int time = 0;

    for (int i = 0; i < n; ++i) {
        if (disc[i] != -1) continue;

        dfs_st.push(i);
        while (!dfs_st.empty()) {
            int u = dfs_st.top();
            if (disc[u] == -1) {
                disc[u] = low[u] = ++time;
                st.push(u);
                onStack[u] = true;

                for (auto it = graph[u].rbegin(); it != graph[u].rend(); ++it) {
                    if (disc[*it] == -1) dfs_st.push(*it);
                }
            } else {
                dfs_st.pop();
                for (int v : graph[u]) {
                    if (onStack[v]) low[u] = min(low[u], low[v]);
                }

                if (low[u] == disc[u]) {
                    vector<int> component;
                    while (true) {
                        int v = st.top(); st.pop();
                        onStack[v] = false;
                        component.push_back(v);
                        if (v == u) break;
                    }
                    sccs.push_back(component);
                }
            }
        }
    }
}
```

Component Condensation

```
vector<vector<int>> buildCondensedGraph(const vector<vector<int>>& graph,
                                       const vector<vector<int>>& sccs) {
    int n = graph.size();
    vector<int> componentId(n);

    for (int i = 0; i < sccs.size(); ++i) {
        for (int u : sccs[i]) {
            componentId[u] = i;
        }
    }

    vector<vector<int>> condensed(sccs.size());
    vector<unordered_set<int>> added(sccs.size());

    for (int u = 0; u < n; ++u) {
        for (int v : graph[u]) {
            if (componentId[u] != componentId[v] &&
                !added[componentId[u]].count(componentId[v])) {
                condensed[componentId[u]].push_back(componentId[v]);
                added[componentId[u]].insert(componentId[v]);
            }
        }
    }
    return condensed;
}
```

35.3 Complexity Analysis

Operation	Time Complexity	Space Complexity
Main Algorithm	$\mathcal{O}(V+E)$	$\mathcal{O}(V)$
Component Condensation	$\mathcal{O}(V+E)$	$\mathcal{O}(V+E)$
Iterative Version	$\mathcal{O}(V+E)$	$\mathcal{O}(V)$

35.4 Comparison with Kosaraju's Algorithm

Feature	Tarjan's	Kosaraju's
Passes	Single DFS	Two DFS Passes
Space	Lower (no transpose)	Higher (stores transpose)
Implementation	More complex	Simpler
Order of SCCs	Reverse topological	Natural topological

35.5 Applications

- Compiler Design: Dead code elimination
- Formal Methods: Model checking
- Circuit Analysis: Feedback loops
- Dependency Resolution: Package managers
- Social Networks: Community detection

35.6 Common Mistakes

- Incorrect `low` update: Missing `low[u] = min(low[u], disc[v])`
- Improper stack management: Forgetting `onStack` tracking
- Wrong root node check: Misidentifying SCC boundaries
- Time mismanagement: Not incrementing `time` properly

35.7 Practice Problems

- LeetCode 1192 - Critical Connections
- SPOJ - Strongly Connected Components
- CSES - 2-SAT
- Codeforces - Network Reliability

35.8 Pro Tips

- Use iterative DFS in large graphs to avoid stack overflow
- `low[u]` represents the lowest discovery time reachable from u
- Condensation helps convert SCCs into a DAG for easier analysis
- Tarjan's can also detect bridges when `low[v] > disc[u]`

Tarjan's algorithm is a compact, elegant solution for finding SCCs with a single DFS. It is widely applicable in compiler theory, dependency resolution, and real-time systems requiring cycle detection.

36 Ford-Fulkerson Algorithm for Maximum Flow

The Ford-Fulkerson method is a greedy algorithm for computing the **maximum flow** in a flow network. It works by repeatedly finding augmenting paths through the residual graph and augmenting the flow until no more paths exist.

36.1 Core Algorithm

36.1.1 Key Concepts

- **Residual Graph**: Represents remaining capacity after current flow
- **Augmenting Path**: Path from source to sink with available capacity
- **Max-Flow Min-Cut Theorem**: Maximum flow equals capacity of minimum cut

36.1.2 Implementation (C++ with BFS - Edmonds-Karp variant)

```cpp
#include <vector>
#include <queue>
#include <climits>
#include <algorithm>

using namespace std;

bool bfs(const vector<vector<int>>& residualGraph, vector<int>& parent, int source, int sink) {
    int n = residualGraph.size();
    vector<bool> visited(n, false);
    queue<int> q;

    q.push(source);
    visited[source] = true;
    parent[source] = -1;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v = 0; v < n; v++) {
            if (!visited[v] && residualGraph[u][v] > 0) {
                parent[v] = u;
                if (v == sink) return true;
                visited[v] = true;
                q.push(v);
            }
        }
    }
    return false;
}

int fordFulkerson(vector<vector<int>>& graph, int source, int sink) {
    int n = graph.size();
    vector<vector<int>> residualGraph = graph;
    vector<int> parent(n);
    int maxFlow = 0;

    while (bfs(residualGraph, parent, source, sink)) {
        int pathFlow = INT_MAX;
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            pathFlow = min(pathFlow, residualGraph[u][v]);
        }

        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            residualGraph[u][v] -= pathFlow;
            residualGraph[v][u] += pathFlow;
        }

        maxFlow += pathFlow;
    }

    return maxFlow;
}
```

36.2 Key Optimizations

36.2.1 Capacity Scaling (Improves Runtime)

```
int capacityScalingMaxFlow(vector<vector<int>>& graph, int source, int sink) {
    int n = graph.size();
    vector<vector<int>> residualGraph = graph;
    int maxFlow = 0;
    int delta = 1 << 30;

    while (delta > 0) {
        vector<int> parent(n, -1);
        queue<int> q;
        q.push(source);
        parent[source] = -2;

        while (!q.empty()) {
            int u = q.front();
            q.pop();

            for (int v = 0; v < n; v++) {
                if (parent[v] == -1 && residualGraph[u][v] >= delta) {
                    parent[v] = u;
                    if (v == sink) break;
                    q.push(v);
                }
            }
        }

        if (parent[sink] != -1) {
            int pathFlow = INT_MAX;
            for (int v = sink; v != source; v = parent[v]) {
                pathFlow = min(pathFlow, residualGraph[parent[v]][v]);
            }

            for (int v = sink; v != source; v = parent[v]) {
                residualGraph[parent[v]][v] -= pathFlow;
                residualGraph[v][parent[v]] += pathFlow;
            }

            maxFlow += pathFlow;
        } else {
            delta >>= 1;
        }
    }

    return maxFlow;
}
```

36.2.2 Dinic's Algorithm (More Efficient)

```
// Implementation of Dinic's algorithm would go here
// Typically achieves O(V^2 * E) time complexity
```

36.3 Complexity Analysis

Variant	Time Complexity	Notes
Ford-Fulkerson	$\mathcal{O}(E \cdot \text{max\_flow})$	Poor for large flows
Edmonds-Karp	$\mathcal{O}(VE^2)$	Uses BFS for path finding
Capacity Scaling	$\mathcal{O}(E^2 \log U)$	U = max capacity
Dinic's	$\mathcal{O}(V^2E)$	Most efficient in practice

36.4 Applications

- **Network Routing**: Maximize data flow
- **Bipartite Matching**: Solve assignment problems
- **Image Segmentation**: Computer vision applications
- **Airline Scheduling**: Maximize flight utilization
- **Baseball Elimination**: Determine if team can win division

36.5 Common Mistakes

- **Missing Reverse Edges**: Forgetting to add residual capacities
- **Infinite Loops**: With irrational capacities (theoretical issue)
- **Incorrect Residual Graph**: Not properly updating capacities
- **Source/Sink Confusion**: Reversing source and sink

36.6 Practice Problems

- SPOJ - Maximum Flow
- POJ 3281 - Dining
- CSES - Project Selection
- LeetCode - Min Cost Flow (related)

36.7 Pro Tips

- **Adjacency Lists**: More efficient than matrices for sparse graphs
- **Early Termination**: Stop if flow reaches required value
- **Reverse Edges**: Crucial for algorithm correctness
- **Test Cases**: Include graphs with multiple paths and bottlenecks

The Ford-Fulkerson method is fundamental to network flow problems, and its Edmonds-Karp variant (using BFS) is particularly practical for coding interviews and programming competitions. For production systems, more advanced algorithms like Dinic's are often preferred.

37 Edmonds-Karp Algorithm for Maximum Flow

The Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson method that uses **Breadth-First Search (BFS)** to find augmenting paths, guaranteeing polynomial time complexity. It is efficient for solving maximum flow problems in networks.

37.1 Core Algorithm

37.1.1 Key Features

- **Time Complexity**: $\mathcal{O}(VE^2)$ (guaranteed polynomial time)
- **Always Terminates**: Unlike basic Ford-Fulkerson for irrational capacities
- **Uses BFS**: Ensures shortest augmenting paths in the residual network

37.1.2 C++ Implementation

```cpp
#include <vector>
#include <queue>
#include <climits>
#include <algorithm>
using namespace std;

bool bfs(const vector<vector<int>>& residualGraph, vector<int>& parent,
         int source, int sink) {
    int n = residualGraph.size();
    vector<bool> visited(n, false);
    queue<int> q;

    q.push(source);
    visited[source] = true;
    parent[source] = -1;

    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v = 0; v < n; v++) {
            if (!visited[v] && residualGraph[u][v] > 0) {
                parent[v] = u;
                if (v == sink) return true;
                visited[v] = true;
                q.push(v);
            }
        }
    }
    return false;
}

int edmondsKarp(vector<vector<int>>& graph, int source, int sink) {
    int n = graph.size();
    vector<vector<int>> residualGraph = graph;
    vector<int> parent(n);
    int maxFlow = 0;

    while (bfs(residualGraph, parent, source, sink)) {
        int pathFlow = INT_MAX;
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            pathFlow = min(pathFlow, residualGraph[u][v]);
        }
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            residualGraph[u][v] -= pathFlow;
            residualGraph[v][u] += pathFlow;
        }
        maxFlow += pathFlow;
    }
    return maxFlow;
}
```

37.2 Key Components

- **Residual Graph**: Tracks remaining capacities
- **BFS**: Finds shortest augmenting paths
- **Parent Array**: Used to reconstruct paths
- **Reverse Edges**: Allow undoing flow

37.3 Optimizations

Adjacency List Representation

```
struct Edge {
    int to, capacity, flow, reverseEdge;
};

vector<vector<Edge>> buildFlowNetwork(const vector<vector<int>>& cap) {
    int n = cap.size();
    vector<vector<Edge>> graph(n);

    for (int u = 0; u < n; ++u) {
        for (int v = 0; v < n; ++v) {
            if (cap[u][v] > 0) {
                int forward = graph[u].size();
                int reverse = graph[v].size();
                graph[u].push_back({v, cap[u][v], 0, reverse});
                graph[v].push_back({u, 0, 0, forward});
            }
        }
    }
    return graph;
}
```

Early Termination

```
if (maxFlow >= targetFlow) break;
```

37.4 Complexity Analysis

Operation	Time	Space
BFS	$\mathcal{O}(V+E)$	$\mathcal{O}(V)$
Augmentation	$\mathcal{O}(V)$	$\mathcal{O}(1)$
Total	$\mathcal{O}(VE^2)$	$\mathcal{O}(V^2)$

37.5 Comparison with Other Flow Algorithms

Algorithm	Time	Features
Edmonds-Karp	$\mathcal{O}(VE^2)$	Simple, guaranteed polynomial
Dinic's	$\mathcal{O}(V^2E)$	Faster in practice, uses level graphs
Push-Relabel	$\mathcal{O}(V^2\sqrt{E})$	Suitable for dense networks
Capacity Scaling	$\mathcal{O}(E^2 \log U)$	U = max edge capacity

37.6 Applications

- Network routing
- Bipartite matching
- Image segmentation
- Baseball elimination
- Supply chain optimization

37.7 Common Mistakes

- Missing reverse edges
- Incorrect residual updates
- Mixing up source and sink
- Not initializing reverse flows

37.8 Practice Problems

- SPOJ: Maximum Flow
- POJ 3281: Dining
- CSES 1146: Project Selection
- LeetCode: Min Cost Flow

37.9 Pro Tips

- Use multiple augmenting paths and test bottlenecks
- Visualize residual graphs for debugging
- Test edge cases like single edge, disconnected graphs
- For large graphs, prefer Dinic's or Push-Relabel

Conclusion: Edmonds-Karp provides a good tradeoff between clarity and performance for many maximum flow problems, especially in educational and competitive programming settings.

38 Kuhn's Algorithm for Bipartite Matching

Kuhn's algorithm (also known as the augmented path algorithm) is an efficient method for finding the **maximum cardinality matching** in bipartite graphs. It runs in $\mathcal{O}(VE)$ time, where V is the number of vertices and E the number of edges.

38.1 Core Algorithm

38.1.1 Key Concepts

- **Bipartite Graph**: Graph whose vertices can be divided into two disjoint sets U and V
- **Matching**: Set of edges without shared vertices
- **Augmenting Path**: Path alternating between matched and unmatched edges

38.1.2 C++ Implementation

```cpp
#include <vector>
#include <algorithm>
using namespace std;

bool tryKuhn(int u, vector<bool>& used, vector<int>& matching,
             const vector<vector<int>>& adj) {
    if (used[u]) return false;
    used[u] = true;

    for (int v : adj[u]) {
        if (matching[v] == -1 || tryKuhn(matching[v], used, matching, adj)) {
            matching[v] = u;
            return true;
        }
    }
    return false;
}

int maxBipartiteMatching(const vector<vector<int>>& adj, int n, int m) {
    vector<int> matching(m, -1);
    int result = 0;

    for (int u = 0; u < n; ++u) {
        vector<bool> used(n, false);
        if (tryKuhn(u, used, matching, adj)) {
            result++;
        }
    }

    return result;
}
```

38.2 Optimizations

Visited Array Optimization

```cpp
vector<bool> usedGlobal(n, false);

for (int u = 0; u < n; ++u) {
    if (tryKuhn(u, usedGlobal, matching, adj)) {
        result++;
        usedGlobal.assign(n, false); // Reset only if needed
    }
}
```

Heuristic Initialization

```cpp
// Greedy initialization
for (int u = 0; u < n; ++u) {
    for (int v : adj[u]) {
        if (matching[v] == -1) {
            matching[v] = u;
            result++;
            break;
        }
    }
}
```

38.3 Complexity Analysis

Variant	Time Complexity	Space Complexity
Basic Kuhn	$\mathcal{O}(VE)$	$\mathcal{O}(V)$
With Greedy Init	$\mathcal{O}(VE)$ (better average)	$\mathcal{O}(V)$
Hopcroft-Karp	$\mathcal{O}(E\sqrt{V})$	$\mathcal{O}(V)$

38.4 Applications

- Job assignment
- Dating systems
- Ad allocation
- University admission
- Sports scheduling

38.5 Common Mistakes

- Incorrect graph representation (non-bipartite)
- Forgetting to reset visited array between DFS calls
- Matching array size must match right partition
- Assuming a perfect matching always exists

38.6 Practice Problems

- SPOJ: Maximum Bipartite Matching
- SPOJ: Courses
- POJ 1274: The Perfect Stall
- Codeforces: Chessboard Coverage

38.7 Pro Tips

- Ensure correct bipartition in graph construction
- Keep left and right partition vertex IDs distinct
- Use Hopcroft-Karp for large datasets
- Visualize small graphs to debug matching

Conclusion: Kuhn's algorithm offers a simple yet powerful technique for solving bipartite matching problems. It serves as a gateway to more advanced algorithms and is widely used in theoretical and applied contexts.

39 Fibonacci Sequence: Dynamic Programming Solutions

The Fibonacci sequence is a classic problem that demonstrates fundamental dynamic programming concepts. Here are optimized approaches to compute Fibonacci numbers efficiently.

39.1 Problem Definition

The Fibonacci sequence is defined as:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

39.2 Solution Approaches

Recursive (Naive) – $\mathcal{O}(2^n)$

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

Problem: Exponential time due to redundant calculations

Memoization (Top-Down DP) – $\mathcal{O}(n)$

```
int fibMemo(int n, vector<int>& memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    return memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo);
}

int fib(int n) {
    vector<int> memo(n+1, -1);
    return fibMemo(n, memo);
}
```

Advantage: Avoids recomputation by storing results

Tabulation (Bottom-Up DP) – $\mathcal{O}(n)$

```
int fib(int n) {
    if (n <= 1) return n;

    vector<int> dp(n+1);
    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    return dp[n];
}
```

Space Optimization: Only need last two values

```
int fib(int n) {
    if (n <= 1) return n;

    int a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }

    return b;
}
```

Matrix Exponentiation – $\mathcal{O}(\log n)$

```
void multiply(int F[2][2], int M[2][2]) {
    int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

    F[0][0] = x; F[0][1] = y;
    F[1][0] = z; F[1][1] = w;
}

void power(int F[2][2], int n) {
    if (n <= 1) return;

    int M[2][2] = {{1,1}, {1,0}};
    power(F, n/2);
    multiply(F, F);

    if (n % 2 != 0) multiply(F, M);
}

int fib(int n) {
    if (n <= 1) return n;

    int F[2][2] = {{1,1}, {1,0}};
    power(F, n-1);
    return F[0][0];
}
```

Binet's Formula (Approximation) – $\mathcal{O}(1)$

```
int fib(int n) {
    double phi = (1 + sqrt(5)) / 2;
    return round(pow(phi, n) / sqrt(5));
}
```

Note: Precision degrades for large n (typically $n > 70$)

39.3 Complexity Comparison

Method	Time	Space	Best For
Recursive	$\mathcal{O}(2^n)$	$\mathcal{O}(n)$	Educational only
Memoization	$\mathcal{O}(n)$	$\mathcal{O}(n)$	Small n, top-down
Tabulation	$\mathcal{O}(n)$	$\mathcal{O}(1)$ (opt.)	General purpose
Matrix	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	Very large n
Binet's	$\mathcal{O}(1)$	$\mathcal{O}(1)$	Approximate values

39.4 Applications

- Algorithm analysis: recursion and dynamic programming
- Financial modeling: Fibonacci retracements
- Computer science: hashing, sorting
- Nature modeling: phyllotaxis and spiral patterns

39.5 Extended Problems

- **Climbing Stairs (LeetCode 70)**: Like Fibonacci
- **N-th Tribonacci Number (LeetCode 1137)**: $F(n) = F(n-1) + F(n-2) + F(n-3)$
- **Fibonacci Modulo**: Compute $F(n)$ % m efficiently
- **Fibonacci GCD**: $\gcd(F(m), F(n)) = F(\gcd(m, n))$

39.6 Pro Tips

- For large n, use matrix exponentiation ($n > 10^6$)
- Apply modulo operations for competitive programming
- Precompute values when Fibonacci is called repeatedly
- Handle edge cases like $n = 0$ and $n = 1$ separately

Conclusion: The Fibonacci sequence serves as a great introduction to dynamic programming, showing the evolution from naive recursion to highly optimized solutions. For interviews, the space-optimized iterative version is usually the most practical.

40 Knapsack Problem: Dynamic Programming Solutions

The knapsack problem is a classic optimization problem where, given a set of items with weights and values, the goal is to determine the most valuable combination that fits within a given weight capacity.

40.1 Problem Definition

0/1 Knapsack Problem

- **Input**:
  - `int W`: Maximum weight capacity
  - `vector<int> wt`: Item weights
  - `vector<int> val`: Item values
  - `int n`: Number of items
- **Output**: Maximum value achievable without exceeding weight `W`

40.2 Solution Approaches

Recursive (Naive) – $\mathcal{O}(2^n)$

```
int knapsack(int W, vector<int>& wt, vector<int>& val, int n) {
    if (n == 0 || W == 0) return 0;

    if (wt[n-1] > W)
        return knapsack(W, wt, val, n-1);
    else
        return max(
            val[n-1] + knapsack(W - wt[n-1], wt, val, n-1),
            knapsack(W, wt, val, n-1)
        );
}
```

Memoization (Top-Down DP) – $\mathcal{O}(nW)$

```
int knapsackMemo(int W, vector<int>& wt, vector<int>& val, int n, vector<vector<int>>& memo) {
    if (n == 0 || W == 0) return 0;
    if (memo[n][W] != -1) return memo[n][W];

    if (wt[n-1] > W)
        return memo[n][W] = knapsackMemo(W, wt, val, n-1, memo);
    else
        return memo[n][W] = max(
            val[n-1] + knapsackMemo(W - wt[n-1], wt, val, n-1, memo),
            knapsackMemo(W, wt, val, n-1, memo)
        );
}

int knapsack(int W, vector<int>& wt, vector<int>& val, int n) {
    vector<vector<int>> memo(n+1, vector<int>(W+1, -1));
    return knapsackMemo(W, wt, val, n, memo);
}
```

Tabulation (Bottom-Up DP) – $\mathcal{O}(nW)$

```
int knapsack(int W, vector<int>& wt, vector<int>& val, int n) {
    vector<vector<int>> dp(n+1, vector<int>(W+1, 0));

    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= W; w++) {
            if (wt[i-1] <= w)
                dp[i][w] = max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w]);
            else
                dp[i][w] = dp[i-1][w];
        }
    }

    return dp[n][W];
}
```

Space-Optimized DP – $\mathcal{O}(W)$

```
int knapsack(int W, vector<int>& wt, vector<int>& val, int n) {
    vector<int> dp(W+1, 0);

    for (int i = 0; i < n; i++) {
        for (int w = W; w >= wt[i]; w--) {
            dp[w] = max(dp[w], val[i] + dp[w - wt[i]]);
        }
    }

    return dp[W];
}
```

40.3 Variations of Knapsack Problem

Unbounded Knapsack (Items can be reused)

```
int unboundedKnapsack(int W, vector<int>& wt, vector<int>& val, int n) {
    vector<int> dp(W+1, 0);

    for (int w = 1; w <= W; w++) {
        for (int i = 0; i < n; i++) {
            if (wt[i] <= w)
                dp[w] = max(dp[w], val[i] + dp[w - wt[i]]);
        }
    }

    return dp[W];
}
```

Fractional Knapsack (Greedy Solution)

```
bool compare(pair<int, int> a, pair<int, int> b) {
    double r1 = (double)a.second / a.first;
    double r2 = (double)b.second / b.first;
    return r1 > r2;
}

double fractionalKnapsack(int W, vector<int>& wt, vector<int>& val, int n) {
    vector<pair<int, int>> items;
    for (int i = 0; i < n; i++)
        items.push_back({wt[i], val[i]});

    sort(items.begin(), items.end(), compare);

    double totalValue = 0.0;
    for (int i = 0; i < n && W > 0; i++) {
        if (items[i].first <= W) {
            W -= items[i].first;
            totalValue += items[i].second;
        } else {
            totalValue += items[i].second * ((double)W / items[i].first);
            break;
        }
    }

    return totalValue;
}
```

40.4 Complexity Analysis

Method	**Time**	**Space**	**Best For**
Recursive	$\mathcal{O}(2^n)$	$\mathcal{O}(n)$	Educational only
Memoization	$\mathcal{O}(nW)$	$\mathcal{O}(nW)$	Small/medium W
Tabulation	$\mathcal{O}(nW)$	$\mathcal{O}(nW)$	General case
Space-Optimized	$\mathcal{O}(nW)$	$\mathcal{O}(W)$	Large n, small W
Unbounded	$\mathcal{O}(nW)$	$\mathcal{O}(W)$	Reusable items
Fractional	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	Divisible items

40.5 Applications

- Resource Allocation: Budgeting, investments
- Inventory Management: Optimal product selection
- Network Design: Bandwidth allocation
- Cryptography: Subset sum problems
- Machine Learning: Feature selection

40.6 Practice Problems

- Partition Equal Subset Sum (LeetCode 416)
- Target Sum (LeetCode 494)
- Coin Change (LeetCode 322)
- Last Stone Weight II (LeetCode 1049)

40.7 Pro Tips

- Identify the knapsack variant: 0/1 vs unbounded vs fractional
- Space Optimization: Use 1D array when possible
- Initialization: `dp[0] = 0` for zero capacity
- Item Order: Use reverse loop for space-optimized 0/1 knapsack
- For large constraints: Consider **meet-in-the-middle** for $n \leq 40$

The knapsack problem is foundational to dynamic programming and appears in various forms in algorithm interviews and real-world optimization. The space-optimized version is particularly useful in competitive programming and interviews.

41 Longest Common Subsequence (LCS): Dynamic Programming Solutions

The Longest Common Subsequence problem finds the longest sequence that appears in the same order in two given sequences (not necessarily contiguous).

41.1 Problem Definition

Given two strings `text1` and `text2`, return the length of their longest common subsequence.

Example:

```
Input: text1 = "abcde", text2 = "ace"
Output: 3 ("ace")
```

41.2 Solution Approaches

41.2.1 Recursive (Naive) – $\mathcal{O}(2^n)$

```cpp
int lcs(string text1, string text2, int m, int n) {
    if (m == 0 || n == 0) return 0;
    if (text1[m-1] == text2[n-1])
        return 1 + lcs(text1, text2, m-1, n-1);
    else
        return max(lcs(text1, text2, m, n-1),
                   lcs(text1, text2, m-1, n));
}
```

41.2.2 Memoization (Top-Down DP) – $\mathcal{O}(mn)$

```cpp
int lcsMemo(string& text1, string& text2, int m, int n, vector<vector<int>>& memo) {
    if (m == 0 || n == 0) return 0;
    if (memo[m][n] != -1) return memo[m][n];

    if (text1[m-1] == text2[n-1])
        return memo[m][n] = 1 + lcsMemo(text1, text2, m-1, n-1, memo);
    else
        return memo[m][n] = max(
            lcsMemo(text1, text2, m, n-1, memo),
            lcsMemo(text1, text2, m-1, n, memo)
        );
}

int longestCommonSubsequence(string text1, string text2) {
    int m = text1.size(), n = text2.size();
    vector<vector<int>> memo(m+1, vector<int>(n+1, -1));
    return lcsMemo(text1, text2, m, n, memo);
}
```

41.2.3 Tabulation (Bottom-Up DP) – $\mathcal{O}(mn)$

```cpp
int longestCommonSubsequence(string text1, string text2) {
    int m = text1.size(), n = text2.size();
    vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (text1[i-1] == text2[j-1])
                dp[i][j] = 1 + dp[i-1][j-1];
            else
                dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
        }
    }

    return dp[m][n];
}
```

41.2.4 Space-Optimized DP – $\mathcal{O}(\min(m, n))$

```cpp
int longestCommonSubsequence(string text1, string text2) {
    if (text1.size() < text2.size())
        return longestCommonSubsequence(text2, text1);

    int m = text1.size(), n = text2.size();
    vector<int> prev(n+1, 0), curr(n+1, 0);

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (text1[i-1] == text2[j-1])
                curr[j] = 1 + prev[j-1];
            else
                curr[j] = max(curr[j-1], prev[j]);
        }
        swap(prev, curr);
    }

    return prev[n];
}
```

41.3 LCS String Reconstruction

```cpp
string getLCS(string text1, string text2) {
    int m = text1.size(), n = text2.size();
    vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

    // Build DP table
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (text1[i-1] == text2[j-1])
                dp[i][j] = 1 + dp[i-1][j-1];
            else
                dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
        }
    }

    // Reconstruct LCS
    string lcs;
    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (text1[i-1] == text2[j-1]) {
            lcs = text1[i-1] + lcs;
            i--; j--;
        } else if (dp[i-1][j] > dp[i][j-1]) {
            i--;
        } else {
            j--;
        }
    }

    return lcs;
}
```

41.4 Complexity Analysis

Method	Time	Space	Best For
Recursive	$\mathcal{O}(2^n)$	$\mathcal{O}(1)$	Educational only
Memoization	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	Small-medium strings
Tabulation	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	General case
Space-Optimized	$\mathcal{O}(mn)$	$\mathcal{O}(\min(m, n))$	Large strings
Reconstruction	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	When LCS string needed

41.5 Variations and Applications

- Diff Tools: File comparison in version control

- DNA Alignment: Bioinformatics sequence matching
- Plagiarism Detection: Document similarity
- Edit Distance: Convert one string to another
- Shortest Common Supersequence: Find shortest string containing both inputs

41.6 Practice Problems

- Longest Palindromic Subsequence (LeetCode 516)
- Delete Operation for Two Strings (LeetCode 583)
- Shortest Common Supersequence (LeetCode 1092)
- Uncrossed Lines (LeetCode 1035)

41.7 Pro Tips

- String Order: Doesn't matter which string is rows/columns
- Space Optimization: Use when only length needed
- Initialization: First row/column always 0
- Reconstruction: Requires full DP table
- Edge Cases: Empty strings, identical strings

The LCS problem is fundamental in dynamic programming and string processing, with applications ranging from bioinformatics to version control systems. The space-optimized DP solution is particularly valuable for coding interviews.

42 Longest Increasing Subsequence (LIS): Dynamic Programming Solutions

The Longest Increasing Subsequence (LIS) problem finds the longest subsequence of a given sequence where elements are in strictly increasing order.

42.1 Problem Definition

Given an integer array `nums`, return the length of the longest strictly increasing subsequence.
Example:

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4 ([2,3,7,101])
```

42.2 Solution Approaches

42.2.1 Dynamic Programming – $\mathcal{O}(n^2)$

```
int lengthOfLIS(vector<int>& nums) {
    int n = nums.size();
    vector<int> dp(n, 1); // Each element is a subsequence of length 1

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
    }

    return *max_element(dp.begin(), dp.end());
}
```

42.2.2 Binary Search Optimization – $\mathcal{O}(n \log n)$

```
int lengthOfLIS(vector<int>& nums) {
    vector<int> tails;

    for (int num : nums) {
        auto it = lower_bound(tails.begin(), tails.end(), num);
        if (it == tails.end()) {
            tails.push_back(num);
        } else {
            *it = num;
        }
    }

    return tails.size();
}
```

42.2.3 Segment Tree / Fenwick Tree – $\mathcal{O}(n \log n)$

Advanced optimization used primarily when reconstruction of LIS is required or queries/updates are involved in an online fashion.

42.3 LIS Reconstruction

42.3.1 DP Approach with Path Tracking

```
vector<int> findLIS(vector<int>& nums) {
    int n = nums.size();
    vector<int> dp(n, 1), parent(n, -1);
    int max_len = 1, last_idx = 0;
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j] && dp[i] < dp[j] + 1) {
                dp[i] = dp[j] + 1;
                parent[i] = j;
            }
        }
        if (dp[i] > max_len) {
            max_len = dp[i];
            last_idx = i;
        }
    }

    vector<int> lis;
    while (last_idx != -1) {
        lis.push_back(nums[last_idx]);
        last_idx = parent[last_idx];
    }
    reverse(lis.begin(), lis.end());
    return lis;
}
```

42.4 Complexity Analysis

Method	**Time**	**Space**	**Best For**
Dynamic Programming	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	Small n ($< 10^4$)
Binary Search	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	Large n ($\leq 10^5$)
Segment Tree	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	When sequence is needed

42.5 Variations and Applications

- Longest Decreasing Subsequence: Reverse the array and apply LIS
- Russian Doll Envelopes: A 2D version of LIS
- Maximum Sum Increasing Subsequence: Maximize sum instead of length
- Building Bridges: Solve using LIS on sorted pairs
- Dilworth's Theorem: Relates to minimum number of chains

42.6 Practice Problems

- Russian Doll Envelopes (LeetCode 354)
- Minimum Number of Removals (LeetCode 1671)
- Longest Increasing Path in Matrix (LeetCode 329)
- Number of LIS (LeetCode 673)

42.7 Pro Tips

- Binary Search method maintains potential candidates of LIS
- DP array starts with all 1's, assuming each element is its own LIS
- Use `lower_bound` for strict LIS, `upper_bound` for non-strict
- Path reconstruction needs parent tracking
- For $n > 10^4$, always use $\mathcal{O}(n \log n)$ approach

42.8 Advanced: Patience Sorting

The $\mathcal{O}(n \log n)$ binary search solution is a variation of the Patience Sorting algorithm.

```
int lengthOfLIS(vector<int>& nums) {
    vector<int> piles; // Top card of each pile

    for (int num : nums) {
        auto pile = lower_bound(piles.begin(), piles.end(), num);
        if (pile == piles.end()) {
            piles.push_back(num);
        } else {
            *pile = num;
        }
    }

    return piles.size();
}
```

This mimics the patience card game where:

- Start a new pile if the number is greater than all current pile tops
- Otherwise, replace the leftmost pile top $\geq$ current number

The number of piles at the end equals the LIS length.

43 Memoization vs Tabulation in Dynamic Programming

43.1 Fundamental Concepts

43.1.1 Memoization (Top-Down DP)

- **Approach**: Recursive solution with cached results
- **Philosophy**: "Remember what you've already computed"
- **Execution**: Starts from the target problem and breaks it into subproblems

43.1.2 Tabulation (Bottom-Up DP)

- **Approach**: Iterative solution that fills a table
- **Philosophy**: "Build up solutions from the base cases"
- **Execution**: Starts from base cases and builds toward the target

43.2 Implementation Comparison

43.2.1 Fibonacci Example

Memoization Approach

```cpp
int fibMemo(int n, vector<int>& memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n]; // Return cached result
    return memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo);
}

int fibonacci(int n) {
    vector<int> memo(n+1, -1);
    return fibMemo(n, memo);
}
```

Tabulation Approach

```cpp
int fibonacci(int n) {
    if (n <= 1) return n;
    vector<int> dp(n+1);
    dp[0] = 0; dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    return dp[n];
}
```

43.3 Key Differences

Characteristic	Memoization	Tabulation
Direction	Top-down	Bottom-up
Approach	Recursive + caching	Iterative table filling
Base Cases	Naturally handled	Explicit initialization
Computation	Only needed subproblems	All subproblems
Stack Usage	Uses call stack	No recursion stack
Code Style	Often more intuitive	More systematic
Overhead	Function call overhead	No call overhead

43.4 When to Use Each

43.4.1 Choose Memoization When

- The problem has a natural recursive structure
- Not all subproblems need to be computed
- The state space is sparse (many unreachable states)
- You want more readable or intuitive code

43.4.2 Choose Tabulation When

- You need to optimize for performance
- The problem requires strict $O(1)$ space optimizations
- You're dealing with large inputs and need to avoid stack overflow
- You need to reconstruct the solution (easier with tabulation)

43.5 Performance Considerations

43.5.1 Time Complexity

- Both usually have the same asymptotic complexity
- Memoization can be faster when many subproblems are avoided
- Tabulation often has better constant factors

43.5.2 Space Complexity

- Memoization: $O(\text{recursion depth}) + O(\text{memo table})$
- Tabulation: $O(\text{table size})$, which is often optimizable

43.6 Advanced Techniques

43.6.1 Memoization Optimization (Sparse State)

```cpp
unordered_map<int, int> memo;
int fibMemo(int n) {
    if (n <= 1) return n;
    if (memo.count(n)) return memo[n];
    return memo[n] = fibMemo(n-1) + fibMemo(n-2);
}
```

43.6.2 Tabulation Space Optimization

```cpp
int fib(int n) {
    if (n <= 1) return n;
    int a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

43.7 Practical Recommendations

- Start with memoization for quick prototyping or interviews
- Convert to tabulation for production or when optimal performance is needed
- Some problems benefit from a hybrid approach
- Visualizing subproblem dependencies helps understand the approach

43.8 Example Problems

43.8.1 Memoization Preferred

- Problems with complex recursive state transitions
- Game theory problems (e.g., minimax with pruning)
- When many subproblems are never actually reached

43.8.2 Tabulation Preferred

- String alignment problems (e.g., edit distance)
- Knapsack and subset-sum type problems
- Grid-based path-counting or reachability

43.9 Common Pitfalls

43.9.1 Memoization

- Forgetting to cache results
- Stack overflow for deep recursion
- Incorrect cache key or state representation

43.9.2 Tabulation

- Wrong iteration order — dependencies must be solved first
- Over-allocating memory
- Missing or incorrect base case initialization

Both techniques are fundamental tools in dynamic programming. Mastering when and how to use each can significantly improve your ability to solve complex optimization problems.

44 Bitmask Dynamic Programming for Traveling Salesman Problem (TSP)

The Traveling Salesman Problem (TSP) asks: *Given a list of cities and distances between them, what is the shortest route that visits each city exactly once and returns to the origin?*

Bitmask dynamic programming offers an efficient solution for small to medium values of n (typically $n \leq 20$).

44.1 Bitmask Fundamentals

44.1.1 Key Concepts

- **Bitmask**: An integer where each bit represents whether a city has been visited.
- **State**: `dp[mask][u]` = shortest path starting at city 0, visiting all cities in `mask`, and ending at city `u`.

44.1.2 Bit Operations Cheatsheet

```
// Set bit (mark city as visited)
mask |= (1 << city)

// Check if city is visited
(mask & (1 << city))

// Count visited cities
__builtin_popcount(mask) // GCC built-in
```

44.2 Standard Implementation

44.2.1 Recursive with Memoization (Top-Down)

```
int n; // Number of cities
vector<vector<int>> dist; // Distance matrix
vector<vector<int>> memo; // DP memo table

int tsp(int mask, int u) {
    if (mask == (1 << n) - 1) // All cities visited
        return dist[u][0];

    if (memo[mask][u] != -1)
        return memo[mask][u];

    int res = INT_MAX;
    for (int v = 0; v < n; v++) {
        if (!(mask & (1 << v))) {
            int newMask = mask | (1 << v);
            res = min(res, dist[u][v] + tsp(newMask, v));
        }
    }

    return memo[mask][u] = res;
}

int main() {
    n = ...;
    dist.assign(n, vector<int>(n));
    memo.assign(1 << n, vector<int>(n, -1));
    cout << tsp(1, 0); // Start from city 0, mask 0001
}
```

44.2.2 Iterative (Bottom-Up)

```
int tsp() {
    const int INF = 1e9;
    vector<vector<int>> dp(1 << n, vector<int>(n, INF));

    dp[1][0] = 0; // Base case: start at city 0

    for (int mask = 1; mask < (1 << n); mask++) {
        for (int u = 0; u < n; u++) {
            if (dp[mask][u] == INF) continue;

            for (int v = 0; v < n; v++) {
                if (!(mask & (1 << v))) {
                    int newMask = mask | (1 << v);
                    dp[newMask][v] = min(dp[newMask][v], dp[mask][u] + dist[u][v]);
                }
            }
        }
    }

    int finalMask = (1 << n) - 1;
    int res = INF;
    for (int u = 1; u < n; u++) {
        res = min(res, dp[finalMask][u] + dist[u][0]);
    }
    return res;
}
```

44.3 Optimizations

44.3.1 Early Pruning

```
if (__builtin_popcount(mask) > current_min_path_length / min_edge_weight)
    return INF; // Prune branches that can't improve
```

44.3.2 Symmetry Breaking

- Fix city 0 as the starting point to avoid duplicate cycles.
- Reduces the state space by a factor of n.

44.3.3 Nearest Neighbor Heuristic

- Initialize with a greedy tour to help pruning with a better initial upper bound.

44.4 Complexity Analysis

Aspect	Complexity	Notes
Time	$\mathcal{O}(n^2 \cdot 2^n)$	Efficient for $n \leq 20$
Space	$\mathcal{O}(n \cdot 2^n)$	Can be reduced using tricks
States	$n \times 2^n$	One for each $(\text{mask}, \text{position})$ pair

44.5 Applications

1. Route optimization for logistics
2. Circuit board drilling path minimization
3. DNA sequencing fragment assembly
4. Data collection in X-ray crystallography
5. Job scheduling with setup costs

44.6 Variations

44.6.1 Hamiltonian Path (No Return)

```
// Modify base case to not return to the starting city
if (mask == (1 << n) - 1) return 0;
```

44.6.2 Multiple Salesmen

```
// Track multiple salesmen in the state
dp[mask][u][k]; // k = number of salesmen used
```

44.6.3 Prize-Collecting TSP

- Some cities may be optional, possibly incurring penalties for skipping.

44.7 Practice Problems

1. TSP (Classic) - SPOJ
2. Little Elephant and TSP - Codeforces
3. TSP with Time Windows - Baekjoon
4. LeetCode Bitmask DP Tag

44.8 Pro Tips

- **Subset Iteration**:

  ```
  // Iterate over all subsets of a mask
  for (int subset = mask; subset; subset = (subset - 1) & mask)
  ```

- Use 1D DP for memory efficiency if current and previous states suffice.
- Precompute distances for faster lookup.
- Always test edge cases like $n = 0$, $n = 1$, symmetric matrices.
- Draw state transition diagrams for small n to debug logic.

Bitmask dynamic programming is a powerful technique for problems involving subsets of elements. While TSP is the classic example, many subset-based optimization problems benefit from the same approach.

45 Digit Dynamic Programming (Counting Numbers with Properties)

Digit DP is a powerful technique for counting numbers in a range $[L, R]$ that satisfy specific digit-based properties.

45.1 Core Concept

Digit DP breaks the number down digit-by-digit while maintaining:

- **Current position** in the number
- **Tight constraint** (whether current digits match the upper bound)
- **Problem-specific state(s)** such as digit sum, modulo value, etc.

45.2 Standard Template

Problem Statement

Count the numbers in $[L, R]$ such that no two adjacent digits are the same.

Solution (C++)

```cpp
#include <cstring>
#include <string>
#include <vector>

using namespace std;

int dp[20][10][2]; // pos, prev_digit, tight

int digitDP(string num, int pos, int prev, int tight, vector<int>& digit) {
    if (pos == num.size()) return 1;

    if (dp[pos][prev][tight] != -1)
        return dp[pos][prev][tight];

    int limit = tight ? digit[pos] : 9;
    int count = 0;

    for (int d = 0; d <= limit; d++) {
        if (d == prev) continue; // Skip adjacent duplicates

        int new_tight = tight && (d == digit[pos]);
        count += digitDP(num, pos + 1, d, new_tight, digit);
    }

    return dp[pos][prev][tight] = count;
}

int countNumbers(int L, int R) {
    string upper = to_string(R);
    vector<int> digit;
    for (char c : upper) digit.push_back(c - '0');

    memset(dp, -1, sizeof(dp));
    int total = digitDP(upper, 0, -1, 1, digit);

    if (L > 0) {
        string lower = to_string(L - 1);
        digit.clear();
        for (char c : lower) digit.push_back(c - '0');

        memset(dp, -1, sizeof(dp));
        total -= digitDP(lower, 0, -1, 1, digit);
    }
    return total;
}
```

45.3 Key Components

1. **State Representation**:
    - `pos` – current digit position
    - `prev` – previous digit used
    - `tight` – whether prefix is still tight to the input number
2. **Memoization Table**:
    - Cache results for subproblems
    - Dimensions depend on tracked state parameters
3. **Digit Processing**:
    - Convert bounds to digit vectors
    - Handle leading zeros separately if needed

45.4 Common Variations

A. Digit Sum Constraints

Count numbers where the sum of digits equals a target S.

```
int dp[20][180][2]; // pos, sum, tight

int digitSumDP(..., int sum) {
    if (pos == num.size()) return sum == target;
    // ... rest similar
}
```

B. Modulo Constraints

Count numbers divisible by K.

```
int dp[20][100][2]; // pos, mod, tight

int modDP(..., int mod) {
    if (pos == num.size()) return mod == 0;
    // update mod = (mod * 10 + d) % K
}
```

C. Non-decreasing Digits

Count numbers with digits in non-decreasing order.

```
if (d < prev && prev != -1) continue; // Enforce d >= prev
```

45.5 Complexity Analysis

Parameter	**Typical Value**	**Notes**
Time	$\mathcal{O}(10 \cdot D \cdot S \cdot 2)$	D = digits, S = state size
Space	$\mathcal{O}(D \cdot S \cdot 2)$	From memoization table
Practical Limit	$D \leq 20$	Supports 64-bit integers

45.6 Applications

- Number theory: primes with digit rules
- Game mechanics: digit-valid combinations
- Cryptography: restricted digit formats
- Combinatorics: bounded enumeration

45.7 Practice Problems

1. Count Special Integers (LeetCode 2376)
2. Numbers At Most N (LeetCode 902)
3. Digit Sum in Range (LeetCode 1067)
4. Non-negative Integers Without Consecutive Ones (LeetCode 600)

45.8 Optimization Tips

- **State Reduction**: Track only what's necessary (e.g., skip prev if not needed)
- **Leading Zeros**: Treat as a special case if needed
- **Preprocessing**: Convert bounds to digit arrays early
- **Early Exit**: Prune impossible branches quickly

Digit DP provides a flexible framework for solving a wide range of number counting problems with digit-level constraints. It's especially effective when:

- Dealing with large numeric ranges (e.g., up to 10^{18})
- Applying digit-based conditions (e.g., no repeating digits, modulo, sum constraints)
- Needing efficient combinatorial enumeration

Once you master the template, adapting Digit DP to new problems becomes a straightforward exercise in state design and recursion.

46 Dynamic Programming on Trees

Tree DP is a powerful technique for solving optimization problems on tree structures by breaking them down into subproblems rooted at each node.

46.1 Key Concepts

Tree Representation

```
struct TreeNode {
    int val;
    vector<TreeNode*> children;
    // or for binary trees:
    TreeNode *left, *right;
};
```

DP Approaches

- **Post-order traversal**: process children before the parent
- **State definition**: store relevant decisions at each node
- **Memoization**: cache subtree results

46.2 Classic Problems

A. Maximum Independent Set

Find the largest set of nodes such that no two are adjacent.
States:

- `dp[node][0]`: excluding the current node
- `dp[node][1]`: including the current node

Solution:

```
pair<int, int> mis(TreeNode* node) {
    if (!node) return {0, 0};

    int exclude = 0, include = 1;

    for (auto child : node->children) {
        auto [child_exclude, child_include] = mis(child);
        exclude += max(child_exclude, child_include);
        include += child_exclude;
    }

    return {exclude, include};
}

int maxIndependentSet(TreeNode* root) {
    auto [excl, incl] = mis(root);
    return max(excl, incl);
}
```

B. Tree Diameter

Find the longest path between any two nodes.
States: Track the two longest downward paths from each node.
Solution:

```
int diameter = 0;

int dfsDiameter(TreeNode* node) {
    if (!node) return 0;
    int max1 = 0, max2 = 0;
    for (auto child : node->children) {
        int depth = dfsDiameter(child);
        if (depth > max1) {
            max2 = max1;
            max1 = depth;
        } else if (depth > max2) {
            max2 = depth;
        }
    }

    diameter = max(diameter, max1 + max2);
    return 1 + max1;
}

int treeDiameter(TreeNode* root) {
    diameter = 0;
    dfsDiameter(root);
    return diameter;
}
```

46.3 Advanced Techniques

A. Rerooting DP

Useful when we need results from all nodes' perspectives.

Example: Sum of distances from each node to all others

```
vector<int> sumDistances(int n, vector<vector<int>>& edges) {
    vector<vector<int>> adj(n);
    for (auto& e : edges) {
        adj[e[0]].push_back(e[1]);
        adj[e[1]].push_back(e[0]);
    }

    vector<int> count(n, 1), res(n, 0);

    function<void(int, int)> post = [&](int u, int parent) {
        for (int v : adj[u]) {
            if (v == parent) continue;
            post(v, u);
            count[u] += count[v];
            res[u] += res[v] + count[v];
        }
    };

    function<void(int, int)> pre = [&](int u, int parent) {
        for (int v : adj[u]) {
            if (v == parent) continue;
            res[v] = res[u] - count[v] + (n - count[v]);
            pre(v, u);
        }
    };

    post(0, -1);
    pre(0, -1);
    return res;
}
```

B. Binary Tree Specific Problems

Example: Maximum path sum in binary tree

```
int maxPathSum(TreeNode* root) {
    int max_sum = INT_MIN;

    function<int(TreeNode*)> dfs = [&](TreeNode* node) {
        if (!node) return 0;
        int left = max(0, dfs(node->left));
        int right = max(0, dfs(node->right));
        max_sum = max(max_sum, left + right + node->val);
        return node->val + max(left, right);
    };

    dfs(root);
    return max_sum;
}
```

46.4 Complexity Analysis

Problem	Time	Space
Independent Set	$\mathcal{O}(V)$	$\mathcal{O}(V)$
Diameter	$\mathcal{O}(V)$	$\mathcal{O}(V)$
Rerooting	$\mathcal{O}(V)$	$\mathcal{O}(V)$
Path Sum	$\mathcal{O}(V)$	$\mathcal{O}(V)$

46.5 Practice Problems

1. House Robber III (LeetCode 337)
2. Binary Tree Cameras (LeetCode 968)
3. Sum of Distances in Tree (LeetCode 834)
4. Diameter of N-ary Tree (LeetCode 1522)

46.6 Pro Tips

- Use **post-order traversal** for most DP aggregations
- Design states around **parent-child transitions**
- Always handle **null nodes or leaf cases**
- Memoization is often **implicit** in recursion
- **Draw sample trees** to understand transitions clearly

Tree DP problems follow a predictable pattern: solve subtrees, combine results, and build up to the full solution. Mastery comes from practicing state design and traversal strategy.

47 Dynamic Programming with Matrix Exponentiation (Fast Recurrence Solving)

Matrix exponentiation is a powerful technique to solve linear recurrence relations in logarithmic time, which is especially useful when dealing with problems where the recurrence needs to be evaluated for very large values of n (like $n = 10^{18}$).

47.1 When to Use Matrix Exponentiation

This technique is applicable when:

- You have a linear recurrence relation
- The recurrence depends on a fixed number of previous terms
- You need to compute the nth term efficiently (in $O(\log n)$ time)

47.2 General Approach

1. **Identify the recurrence relation**: Express your problem as a recurrence.
2. **Construct the transformation matrix**: This matrix encodes how to get from one state to the next.
3. **Exponentiate the matrix**: Raise the matrix to the $(n-k)$th power (where k is the order of the recurrence).
4. **Multiply by initial vector**: Combine with initial conditions to get the result.

47.3 Example: Fibonacci Numbers

The Fibonacci sequence is defined by:

$$F(n) = F(n-1) + F(n-2), \quad \text{with } F(0) = 0, \quad F(1) = 1$$

47.3.1 Step 1: Represent as matrix transformation

We can represent the recurrence as:

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$$

47.3.2 Step 2: General form

To find $F(n)$, we can compute:

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}$$

47.3.3 Implementation

```cpp
#include <vector>
using namespace std;

using Matrix = vector<vector<long long>>;

Matrix matrix_mult(const Matrix &a, const Matrix &b) {
    int n = a.size(), m = b[0].size(), p = b.size();
    Matrix result(n, vector<long long>(m, 0));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            for (int k = 0; k < p; ++k)
                result[i][j] += a[i][k] * b[k][j];
    return result;
}
```

```cpp
Matrix matrix_pow(Matrix mat, long long power) {
    int n = mat.size();
    Matrix result(n, vector<long long>(n, 0));
    for (int i = 0; i < n; ++i) result[i][i] = 1; // Identity matrix
    while (power > 0) {
        if (power & 1) result = matrix_mult(result, mat);
        mat = matrix_mult(mat, mat);
        power >>= 1;
    }
    return result;
}

long long fibonacci(long long n) {
    if (n == 0) return 0;
    Matrix transformation = {{1,1},{1,0}};
    Matrix result = matrix_pow(transformation, n-1);
    return result[0][0];
}
```

47.4 Generalizing to k-th Order Recurrence

For a recurrence of the form:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + \cdots + a_k f(n-k)$$

The transformation matrix is $k \times k$:

$$\begin{bmatrix} a_1 & a_2 & a_3 & \cdots & a_k & \\ 1 & 0 & 0 & \cdots & 0 & \\ 0 & 1 & 0 & \cdots & 0 & \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

47.5 Time Complexity

- Naive DP approach: $O(n)$
- Matrix exponentiation: $O(k^3 \log n)$, where k is the order of the recurrence

This makes matrix exponentiation vastly superior for very large n.

47.6 Applications

1. Computing Fibonacci numbers for very large n
2. Counting paths in graphs (number of walks of length n between nodes)
3. Solving linear recurrence relations in competitive programming
4. Problems where direct DP would be too slow due to large constraints

47.7 Example: Tribonacci Numbers

Recurrence:

$$T(n) = T(n-1) + T(n-2) + T(n-3)$$

Transformation matrix:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Implementation would be similar but with 3×3 matrices.

Matrix exponentiation is a powerful tool in a competitive programmer's arsenal for solving recurrence relations efficiently.

48 Sliding Window Optimization in Dynamic Programming

The sliding window technique is a space optimization method for certain DP problems where the recurrence relation depends only on a fixed number of previous states. This allows us to reduce space complexity from $O(n)$ to $O(1)$ or $O(k)$ where k is the window size.

48.1 When to Use Sliding Window Optimization

- When the DP state transition only depends on a limited window of previous states
- When you need to optimize space complexity
- Common in problems like Fibonacci sequence, staircase climbing, or sequence problems with limited dependencies

48.2 Classic Example: Fibonacci Sequence

48.2.1 Standard DP Approach ($O(n)$ space)

```
int fib(int n) {
    if (n <= 1) return n;
    vector<int> dp(n+1);
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

48.2.2 Sliding Window Optimized ($O(1)$ space)

```
int fib(int n) {
    if (n <= 1) return n;
    int a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

48.3 General Pattern for Sliding Window Optimization

1. Identify that the DP state depends only on a fixed window of previous states
2. Replace the DP array with a few variables representing the window
3. Update these variables in each iteration, sliding the window forward

48.4 Example: Climbing Stairs (k steps)

Problem: You can climb 1, 2, ..., k steps at a time. How many ways to reach the top?

48.4.1 Standard DP ($O(n)$ space)

```
int climbStairs(int n, int k) {
    vector<int> dp(n+1, 0);
    dp[0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= k; j++) {
            if (i - j >= 0) {
                dp[i] += dp[i - j];
            }
        }
    }
    return dp[n];
}
```

48.4.2 Sliding Window Optimized ($O(k)$ space)

```
int climbStairs(int n, int k) {
    if (n == 0) return 1;
    vector<int> dp(k, 0);
    dp[0] = 1;
    for (int i = 1; i <= n; i++) {
        int sum = 0;
        for (int j = 0; j < k; j++) {
            if (i - j - 1 >= 0) {
                sum += dp[(i - j - 1) % k];
            }
        }
        dp[i % k] = sum;
    }
    return dp[n % k];
}
```

48.5 Example: Maximum Sum Subarray of Size K

48.5.1 Standard Approach ($O(n)$ space)

```
int maxSumSubarray(vector<int>& nums, int k) {
    int n = nums.size();
    vector<int> dp(n, 0);
    dp[0] = nums[0];
    int max_sum = dp[0];

    for (int i = 1; i < n; i++) {
        dp[i] = nums[i] + (i >= k ? 0 : dp[i-1]);
        max_sum = max(max_sum, dp[i]);
    }
    return max_sum;
}
```

48.5.2 Sliding Window Optimized ($O(1)$ space)

```
int maxSumSubarray(vector<int>& nums, int k) {
    int window_sum = 0;
    int max_sum = 0;

    for (int i = 0; i < nums.size(); i++) {
        window_sum += nums[i];
        if (i >= k) {
            window_sum -= nums[i - k];
        }
        if (i >= k - 1) {
            max_sum = max(max_sum, window_sum);
        }
    }
    return max_sum;
}
```

48.6 Benefits of Sliding Window Optimization

1. **Space Efficiency**: Reduces space complexity significantly
2. **Cache Friendly**: Fewer variables mean better cache utilization
3. **Same Time Complexity**: Maintains the original time complexity while using less space

48.7 When Not to Use

- When the DP state depends on many or all previous states
- When you need to reconstruct the solution path (may need to keep the full DP array)

The sliding window technique is a powerful optimization that can make your DP solutions more memory efficient while maintaining the same time complexity.

49 Convex Hull Trick (CHT) for DP Optimization

The Convex Hull Trick is an optimization technique that reduces the time complexity of certain dynamic programming problems from $O(n^2)$ to $O(n \log n)$ or even $O(n)$ by maintaining the convex hull of linear functions and enabling efficient minimum/maximum queries.

49.1 When to Use Convex Hull Trick

- **Applicable when**: Your DP transition has the form:

$$dp[i] = \min / \max_{j<i} (dp[j] + a[i] \cdot b[j] + c[j])$$

- **Common in problems**: Where you need to find optimal previous states that can be represented as linear functions
- **Key requirement**: The coefficients must satisfy certain monotonicity conditions for $O(n)$ optimization

49.2 Standard Form

The general form where CHT applies:

$$dp[i] = \min / \max (a[i] \cdot b[j] + c[j] + d[i]) + \text{constant}$$

where:

- $a[i]$ is a function of the current state
- $b[j], c[j]$ are functions of the previous state
- $d[i]$ is a function of the current state that doesn't affect the optimization

49.3 Implementation Approaches

49.3.1 Basic CHT (Offline, $O(n \log n)$)

When queries $a[i]$ and insertions $(b[j], c[j])$ are arbitrary:

```cpp
#include <vector>
#include <algorithm>
using namespace std;

struct Line {
    long long m, b;
    long long eval(long long x) const { return m * x + b; }
};

class CHT {
    vector<Line> hull;

    // Returns true if line l3 is better than l2 at the intersection of l1-l2
    bool bad(const Line& l1, const Line& l2, const Line& l3) {
        return (l3.b - l1.b) * (l1.m - l2.m) <= (l2.b - l1.b) * (l1.m - l3.m);
    }

public:
    void add_line(long long m, long long b) {
        Line l = {m, b};
        while (hull.size() >= 2 && bad(hull[hull.size()-2], hull.back(), l)) {
            hull.pop_back();
        }
        hull.push_back(l);
    }

    long long query(long long x) {
        int l = 0, r = hull.size() - 1;
        while (l < r) {
            int mid = (l + r) / 2;
            if (hull[mid].eval(x) < hull[mid+1].eval(x)) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return hull[l].eval(x);
    }
};
```

49.3.2 Online CHT with Dynamic Programming

```
vector<long long> dp(n);
CHT cht;
cht.add_line(b[0], c[0]); // Initial line

for (int i = 1; i < n; i++) {
    dp[i] = cht.query(a[i]) + d[i]; // Get minimum value
    cht.add_line(b[i], c[i]);       // Add new line to the hull
}
```

49.3.3 Fully Dynamic CHT (Li-Chao Tree)

For cases where both insertions and queries are arbitrary and online:

```
struct LiChaoNode {
    Line line;
    LiChaoNode *left = nullptr, *right = nullptr;
    LiChaoNode(Line l) : line(l) {}
};

class LiChaoTree {
    LiChaoNode* root = nullptr;
    long long L, R;

public:
    LiChaoTree(long long l, long long r) : L(l), R(r) {}

    void insert(Line new_line) {
        insert(root, L, R, new_line);
    }

    void insert(LiChaoNode* &node, long long l, long long r, Line new_line) {
        if (!node) {
            node = new LiChaoNode(new_line);
            return;
        }

        long long m = (l + r) / 2;
        bool left = new_line.eval(l) < node->line.eval(l);
        bool mid = new_line.eval(m) < node->line.eval(m);

        if (mid) {
            swap(node->line, new_line);
        }

        if (left != mid) {
            insert(node->left, l, m, new_line);
        } else {
            insert(node->right, m+1, r, new_line);
        }
    }

    long long query(long long x) {
        return query(root, L, R, x);
    }

    long long query(LiChaoNode* node, long long l, long long r, long long x) {
        if (!node) return LLONG_MAX;

        long long curr = node->line.eval(x);
        long long m = (l + r) / 2;

        if (x < m) {
            return min(curr, query(node->left, l, m, x));
        } else {
            return min(curr, query(node->right, m+1, r, x));
        }
    }
};
```

49.4 Example Problem: Special Sequence

Problem: Given a sequence $a[1..n]$, compute:

$$dp[i] = \min_{j<i} \left(dp[j] + (a[i] - a[j])^2 + C \right)$$

49.4.1 Naive DP Solution ($O(n^2)$)

```
vector<long long> dp(n, LLONG_MAX);
dp[0] = 0;
for (int i = 1; i < n; i++) {
    for (int j = 0; j < i; j++) {
        dp[i] = min(dp[i], dp[j] + (a[i]-a[j])*(a[i]-a[j]) + C);
    }
}
```

49.4.2 Optimized with CHT ($O(n)$)

First, expand the equation:

$$dp[i] = \min_{j} \left(-2a[j]a[i] + a[j]^2 + dp[j] \right) + a[i]^2 + C$$

This fits our CHT form where:

- $m[j] = -2a[j]$ (slope)
- $b[j] = a[j]^2 + dp[j]$ (intercept)
- Query at $x = a[i]$

```
vector<long long> dp(n);
dp[0] = 0;

CHT cht;
cht.add_line(-2 * a[0], a[0]*a[0] + dp[0]);

for (int i = 1; i < n; i++) {
    long long min_val = cht.query(a[i]);
    dp[i] = min_val + a[i]*a[i] + C;
    cht.add_line(-2 * a[i], a[i]*a[i] + dp[i]);
}
```

49.5 Advanced Variations

- Maximum instead of minimum: Reverse the comparator in all operations
- Fully dynamic CHT: When lines can be added in any order (using Li-Chao Tree)
- Slope trick: Special case where slopes are integers and differ by 1

49.6 Complexity Analysis

- Basic CHT: $O(n)$ when lines are inserted in order of increasing/decreasing slope
- Li-Chao Tree: $O(\log n)$ per insertion and query
- General CHT: $O(n \log n)$ for arbitrary insertions and queries

49.7 Practical Considerations

1. Overflow: Use 128-bit integers if needed
2. Floating points: Avoid when possible due to precision issues
3. Initialization: Make sure to handle edge cases properly

The Convex Hull Trick is a powerful optimization that can dramatically speed up certain DP problems by exploiting the linear structure of the transitions.

50 Prime Numbers in Number Theory

Prime numbers are the building blocks of number theory, with applications ranging from cryptography to algorithm optimization. Here's a comprehensive overview of prime number concepts and algorithms in C++.

50.1 Basic Primality Tests

50.1.1 Trial Division (O($\sqrt{n}$))

```cpp
bool isPrime(int n) {
    if (n <= 1) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;

    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0) return false;
    }
    return true;
}
```

50.1.2 Optimized Trial Division

```cpp
bool isPrime(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;

    // Check divisors of form 6k ± 1
    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0)
            return false;
    }
    return true;
}
```

50.2 Sieve Algorithms

50.2.1 Sieve of Eratosthenes (O($n \log \log n$))

```cpp
vector<bool> sieve(int n) {
    vector<bool> is_prime(n+1, true);
    is_prime[0] = is_prime[1] = false;

    for (int p = 2; p * p <= n; p++) {
        if (is_prime[p]) {
            for (int i = p * p; i <= n; i += p) {
                is_prime[i] = false;
            }
        }
    }
    return is_prime;
}
```

50.2.2 Segmented Sieve (for large ranges)

```cpp
void segmentedSieve(int L, int H) {
    int limit = sqrt(H);
    vector<bool> mark(limit + 1, true);
    vector<int> primes;

    // Regular sieve to find primes up to H
    for (int p = 2; p * p <= limit; p++) {
        if (mark[p]) {
            for (int i = p * p; i <= limit; i += p) {
                mark[i] = false;
            }
        }
    }

    // Store primes up to H
    for (int p = 2; p <= limit; p++) {
        if (mark[p]) primes.push_back(p);
    }

    // Sieve in the range [L, H]
    vector<bool> is_prime(H - L + 1, true);
    for (int p : primes) {
        int start = max(p * p, (L + p - 1) / p * p);
        for (int j = start; j <= H; j += p) {
            is_prime[j - L] = false;
        }
    }

    // Output primes in [L, H]
    for (int i = max(L, 2); i <= H; i++) {
        if (is_prime[i - L]) {
            cout << i << " ";
        }
    }
}
```

50.3 Probabilistic Primality Tests

50.3.1 Miller-Rabin Test ($\mathbf{O}(k \log^3 n)$)

```
using ll = long long;

ll powmod(ll a, ll b, ll mod) {
    ll res = 1;
    a %= mod;
    while (b > 0) {
        if (b & 1) res = (res * a) % mod;
        a = (a * a) % mod;
        b >>= 1;
    }
    return res;
}

bool millerTest(ll d, ll n) {
    ll a = 2 + rand() % (n - 4);
    ll x = powmod(a, d, n);

    if (x == 1 || x == n - 1) return true;

    while (d != n - 1) {
        x = (x * x) % n;
        d *= 2;
        if (x == 1) return false;
        if (x == n - 1) return true;
    }
    return false;
}

bool isPrime(ll n, int k = 5) {
    if (n <= 1 || n == 4) return false;
    if (n <= 3) return true;

    ll d = n - 1;
    while (d % 2 == 0) d /= 2;

    for (int i = 0; i < k; i++) {
        if (!millerTest(d, n)) return false;
    }
    return true;
}
```

50.4 Prime Factorization

50.4.1 Trial Division Factorization

```cpp
vector<int> factorize(int n) {
    vector<int> factors;
    while (n % 2 == 0) {
        factors.push_back(2);
        n /= 2;
    }

    for (int i = 3; i * i <= n; i += 2) {
        while (n % i == 0) {
            factors.push_back(i);
            n /= i;
        }
    }

    if (n > 1) factors.push_back(n);
    return factors;
}
```

50.4.2 Pollard's Rho Algorithm (for large numbers)

```cpp
ll pollardsRho(ll n) {
    if (n % 2 == 0) return 2;
    if (n % 3 == 0) return 3;

    ll x = rand() % (n - 2) + 2;
    ll y = x;
    ll c = rand() % (n - 1) + 1;
    ll d = 1;

    auto f = [&](ll x) { return (powmod(x, 2, n) + c) % n; };

    while (d == 1) {
        x = f(x);
        y = f(f(y));
        d = __gcd(abs(x - y), n);
    }
    return d;
}

vector<ll> factorize(ll n) {
    vector<ll> factors;
    if (n == 1) return factors;
    if (isPrime(n)) {
        factors.push_back(n);
        return factors;
    }

    ll d = pollardsRho(n);
    vector<ll> left = factorize(d);
    vector<ll> right = factorize(n / d);

    factors.insert(factors.end(), left.begin(), left.end());
    factors.insert(factors.end(), right.begin(), right.end());
    return factors;
}
```

50.5 Prime Counting Function $\pi(n)$

50.5.1 Legendre's Formula

```cpp
int countPrimes(int n) {
    if (n < 2) return 0;
    vector<bool> is_prime(n+1, true);
    is_prime[0] = is_prime[1] = false;
    int count = 0;
    for (int p = 2; p * p <= n; p++) {
        if (is_prime[p]) {
            for (int i = p * p; i <= n; i += p) {
                is_prime[i] = false;
            }
        }
    }

    for (bool prime : is_prime) {
        if (prime) count++;
    }
    return count;
}
```

50.6 Applications

50.6.1 Prime Generation in Range

```
vector<int> generatePrimes(int L, int H) {
    vector<int> primes;
    vector<bool> is_prime(H - L + 1, true);

    for (int p = 2; p * p <= H; p++) {
        int start = max(p * p, (L + p - 1) / p * p);
        for (int j = start; j <= H; j += p) {
            is_prime[j - L] = false;
        }
    }

    for (int i = max(L, 2); i <= H; i++) {
        if (is_prime[i - L]) {
            primes.push_back(i);
        }
    }
    return primes;
}
```

50.6.2 Next Prime Number

```
int nextPrime(int n) {
    if (n < 2) return 2;
    while (true) {
        n++;
        if (isPrime(n)) return n;
    }
}
```

These algorithms form the foundation for working with prime numbers in competitive programming and cryptography applications. The choice of algorithm depends on the specific requirements of your problem, particularly the size of the numbers involved and the performance constraints.

51 Modular Arithmetic in Number Theory

Modular arithmetic is a fundamental concept in number theory with wide applications in cryptography, computer science, and competitive programming. Here's a comprehensive guide to modular arithmetic operations and algorithms in C++.

51.1 Basic Operations

51.1.1 Modular Addition and Subtraction

```
const int MOD = 1e9+7;

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

int sub(int a, int b) {
    return (a - b + MOD) % MOD;
}
```

51.1.2 Modular Multiplication

```
int mul(int a, int b) {
    return (1LL * a * b) % MOD;
}
```

51.1.3 Modular Exponentiation (Binary Exponentiation)

```
int power(int a, int b) {
    int res = 1;
    while (b > 0) {
        if (b & 1) res = mul(res, a);
        a = mul(a, a);
        b >>= 1;
    }
    return res;
}
```

51.2 Modular Inverse

51.2.1 Using Fermat's Little Theorem (for prime MOD)

```
int inv(int a) {
    return power(a, MOD-2);
}
```

51.2.2 Using Extended Euclidean Algorithm

```
int extended_gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = extended_gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}

int inv(int a) {
    int x, y;
    extended_gcd(a, MOD, x, y);
    return (x % MOD + MOD) % MOD;
}
```

51.3 Division Under Modulo

```
int divide(int a, int b) {
    return mul(a, inv(b));
}
```

51.4 Modular Arithmetic with Factorials

51.4.1 Precompute Factorials and Inverses

```
const int MAXN = 1e6+5;
int fact[MAXN], inv_fact[MAXN];

void precompute() {
    fact[0] = 1;
    for (int i = 1; i < MAXN; i++) {
        fact[i] = mul(fact[i-1], i);
    }

    inv_fact[MAXN-1] = inv(fact[MAXN-1]);
    for (int i = MAXN-2; i >= 0; i--) {
        inv_fact[i] = mul(inv_fact[i+1], i+1);
    }
}

int comb(int n, int k) {
    if (k < 0 || k > n) return 0;
    return mul(fact[n], mul(inv_fact[k], inv_fact[n-k]));
}
```

51.5 Chinese Remainder Theorem (CRT)

```
pair<int, int> crt(int a1, int m1, int a2, int m2) {
    int p, q;
    int g = extended_gcd(m1, m2, p, q);
    if ((a1 - a2) % g != 0) return {0, -1}; // no solution

    int lcm = m1 / g * m2;
    int x = (a1 + (a2 - a1)/g * p % (m2/g) * m1) % lcm;
    return {x < 0 ? x + lcm : x, lcm};
}

// Generalized CRT for multiple congruences
pair<int, int> crt(vector<pair<int, int>> congruences) {
    int a = 0, m = 1;
    for (auto [ai, mi] : congruences) {
        auto [x, lcm] = crt(a, m, ai, mi);
        if (lcm == -1) return {0, -1};
        a = x;
        m = lcm;
    }
    return {a, m};
}
```

51.6 Discrete Logarithm (Baby-step Giant-step)

```
int baby_step_giant_step(int a, int b, int m) {
    a %= m;
    b %= m;

    int n = sqrt(m) + 1;
    unordered_map<int, int> vals;
    for (int p = 1, curr = 1; p <= n; p++) {
        curr = (1LL * curr * a) % m;
        vals[curr] = p;
    }

    int an = 1;
    for (int i = 0; i < n; i++) {
        an = (1LL * an * a) % m;
    }

    for (int q = 0, curr = b; q <= n; q++) {
        if (vals.count(curr)) {
            return vals[curr] + q * n;
        }
        curr = (1LL * curr * an) % m;
    }
    return -1; // no solution
}
```

51.7 Lucas Theorem (for binomial coefficients modulo prime)

```
int lucas(int n, int k, int p) {
    if (k == 0) return 1;
    return (1LL * comb(n % p, k % p, p) * lucas(n / p, k / p, p)) % p;
}
```

51.8 Wilson's Theorem Applications

```
// (p-1)!  -1 mod p for prime p
bool isPrimeWilson(int p) {
    if (p <= 1) return false;
    int fact = 1;
    for (int i = 2; i < p; i++) {
        fact = (1LL * fact * i) % p;
    }
    return fact == p - 1;
}
```

51.9 Modulo Operations with Negative Numbers

```
int mod(int a, int m) {
    return (a % m + m) % m;
}
```

51.10 Fast Modular Multiplication (for large numbers)

```
// Russian peasant algorithm
uint64_t mul_mod(uint64_t a, uint64_t b, uint64_t m) {
    a %= m;
    b %= m;
    uint64_t res = 0;
    while (b > 0) {
        if (b & 1) res = (res + a) % m;
        a = (a << 1) % m;
        b >>= 1;
    }
    return res;
}
```

These modular arithmetic operations form the foundation for many advanced algorithms in number theory and cryptography. The implementations provided are optimized for performance and correctness, handling edge cases like negative numbers and overflow.

52 Modular Arithmetic in Number Theory

Modular arithmetic is a fundamental concept in number theory with wide applications in cryptography, computer science, and competitive programming. Here's a comprehensive guide to modular arithmetic operations and algorithms in C++.

52.1 Basic Operations

52.1.1 Modular Addition and Subtraction

```
const int MOD = 1e9+7;

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

int sub(int a, int b) {
    return (a - b + MOD) % MOD;
}
```

52.1.2 Modular Multiplication

```
int mul(int a, int b) {
    return (1LL * a * b) % MOD;
}
```

52.1.3 Modular Exponentiation (Binary Exponentiation)

```
int power(int a, int b) {
    int res = 1;
    while (b > 0) {
        if (b & 1) res = mul(res, a);
        a = mul(a, a);
        b >>= 1;
    }
    return res;
}
```

52.2 Modular Inverse

52.2.1 Using Fermat's Little Theorem (for prime MOD)

```
int inv(int a) {
    return power(a, MOD-2);
}
```

52.2.2 Using Extended Euclidean Algorithm

```
int extended_gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = extended_gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}

int inv(int a) {
    int x, y;
    extended_gcd(a, MOD, x, y);
    return (x % MOD + MOD) % MOD;
}
```

52.3 Division Under Modulo

```
int divide(int a, int b) {
    return mul(a, inv(b));
}
```

52.4 Modular Arithmetic with Factorials

52.4.1 Precompute Factorials and Inverses

```
const int MAXN = 1e6+5;
int fact[MAXN], inv_fact[MAXN];

void precompute() {
    fact[0] = 1;
    for (int i = 1; i < MAXN; i++) {
        fact[i] = mul(fact[i-1], i);
    }

    inv_fact[MAXN-1] = inv(fact[MAXN-1]);
    for (int i = MAXN-2; i >= 0; i--) {
        inv_fact[i] = mul(inv_fact[i+1], i+1);
    }
}

int comb(int n, int k) {
    if (k < 0 || k > n) return 0;
    return mul(fact[n], mul(inv_fact[k], inv_fact[n-k]));
}
```

52.5 Chinese Remainder Theorem (CRT)

```
pair<int, int> crt(int a1, int m1, int a2, int m2) {
    int p, q;
    int g = extended_gcd(m1, m2, p, q);
    if ((a1 - a2) % g != 0) return {0, -1}; // no solution

    int lcm = m1 / g * m2;
    int x = (a1 + (a2 - a1)/g * p % (m2/g) * m1) % lcm;
    return {x < 0 ? x + lcm : x, lcm};
}

// Generalized CRT for multiple congruences
pair<int, int> crt(vector<pair<int, int>> congruences) {
    int a = 0, m = 1;
    for (auto [ai, mi] : congruences) {
        auto [x, lcm] = crt(a, m, ai, mi);
        if (lcm == -1) return {0, -1};
        a = x;
        m = lcm;
    }
    return {a, m};
}
```

52.6 Discrete Logarithm (Baby-step Giant-step)

```
int baby_step_giant_step(int a, int b, int m) {
    a %= m;
    b %= m;

    int n = sqrt(m) + 1;
    unordered_map<int, int> vals;
    for (int p = 1, curr = 1; p <= n; p++) {
        curr = (1LL * curr * a) % m;
        vals[curr] = p;
    }

    int an = 1;
    for (int i = 0; i < n; i++) {
        an = (1LL * an * a) % m;
    }

    for (int q = 0, curr = b; q <= n; q++) {
        if (vals.count(curr)) {
            return vals[curr] + q * n;
        }
        curr = (1LL * curr * an) % m;
    }
    return -1; // no solution
}
```

52.7 Lucas Theorem (for binomial coefficients modulo prime)

```
int lucas(int n, int k, int p) {
    if (k == 0) return 1;
    return (1LL * comb(n % p, k % p, p) * lucas(n / p, k / p, p)) % p;
}
```

52.8 Wilson's Theorem Applications

```
// (p-1)!  -1 mod p for prime p
bool isPrimeWilson(int p) {
    if (p <= 1) return false;
    int fact = 1;
    for (int i = 2; i < p; i++) {
        fact = (1LL * fact * i) % p;
    }
    return fact == p - 1;
}
```

52.9 Modulo Operations with Negative Numbers

```
int mod(int a, int m) {
    return (a % m + m) % m;
}
```

52.10 Fast Modular Multiplication (for large numbers)

```
// Russian peasant algorithm
uint64_t mul_mod(uint64_t a, uint64_t b, uint64_t m) {
    a %= m;
    b %= m;
    uint64_t res = 0;
    while (b > 0) {
        if (b & 1) res = (res + a) % m;
        a = (a << 1) % m;
        b >>= 1;
    }
    return res;
}
```

These modular arithmetic operations form the foundation for many advanced algorithms in number theory and cryptography. The implementations provided are optimized for performance and correctness, handling edge cases like negative numbers and overflow.

53 Combinatorics in Number Theory and Mathematics

Combinatorics is a fundamental area of mathematics with wide applications in computer science, probability, and optimization. Here's a comprehensive guide to combinatorial concepts and algorithms with C++ implementations.

53.1 Basic Counting Principles

53.1.1 Factorial Computation

```cpp
const int MOD = 1e9+7;
vector<int> fact(MAXN, 1); // Precompute up to MAXN

void precompute_factorials() {
    for (int i = 1; i < MAXN; i++) {
        fact[i] = (1LL * fact[i-1] * i) % MOD;
    }
}

int factorial(int n) {
    return fact[n];
}
```

53.1.2 Permutations (nPk)

```cpp
int permutations(int n, int k) {
    if (k > n) return 0;
    return (1LL * fact[n] * inv(fact[n-k])) % MOD;
}
```

53.2 Combinations (nCk)

53.2.1 Basic Combination Formula

```cpp
int combinations(int n, int k) {
    if (k < 0 || k > n) return 0;
    return (1LL * fact[n] * inv((1LL * fact[k] * fact[n-k]) % MOD)) % MOD;
}
```

53.2.2 Pascal's Triangle Approach (O(n^2) preprocess, O(1) query)

```cpp
vector<vector<int>> comb(MAXN, vector<int>(MAXN, 0));

void build_pascals_triangle() {
    comb[0][0] = 1;
    for (int n = 1; n < MAXN; n++) {
        comb[n][0] = 1;
        for (int k = 1; k <= n; k++) {
            comb[n][k] = (comb[n-1][k-1] + comb[n-1][k]) % MOD;
        }
    }
}
```

53.2.3 Lucas Theorem (for large n, k modulo small prime p)

```cpp
int lucas_comb(int n, int k, int p) {
    int res = 1;
    while (n > 0 || k > 0) {
        int ni = n % p;
        int ki = k % p;
        if (ki > ni) return 0;
        res = (res * comb[ni][ki]) % p;
        n /= p;
        k /= p;
    }
    return res;
}
```

53.3 Advanced Combinatorial Concepts

53.3.1 Catalan Numbers

```
vector<int> catalan(MAXN);

void precompute_catalan() {
    catalan[0] = catalan[1] = 1;
    for (int i = 2; i < MAXN; i++) {
        catalan[i] = 0;
        for (int j = 0; j < i; j++) {
            catalan[i] = (catalan[i] + (1LL * catalan[j] * catalan[i-j-1]) % MOD) % MOD;
        }
    }
}

// Direct formula
int catalan_number(int n) {
    return divide(comb(2*n, n), n+1);
}
```

53.3.2 Derangements (Subfactorial)

```
vector<int> derange(MAXN);

void precompute_derangements() {
    derange[0] = 1;
    derange[1] = 0;
    for (int i = 2; i < MAXN; i++) {
        derange[i] = (1LL * (i-1) * (derange[i-1] + derange[i-2])) % MOD;
    }
}
```

53.3.3 Stirling Numbers

First Kind (Cyclic Permutations)

```
vector<vector<int>> stirling1(MAXN, vector<int>(MAXN, 0));

void precompute_stirling1() {
    stirling1[0][0] = 1;
    for (int n = 1; n < MAXN; n++) {
        for (int k = 1; k <= n; k++) {
            stirling1[n][k] = (stirling1[n-1][k-1] + (1LL * (n-1) * stirling1[n-1][k]) % MOD) % MOD;
        }
    }
}
```

Second Kind (Partitions)

```
vector<vector<int>> stirling2(MAXN, vector<int>(MAXN, 0));

void precompute_stirling2() {
    stirling2[0][0] = 1;
    for (int n = 1; n < MAXN; n++) {
        for (int k = 1; k <= n; k++) {
            stirling2[n][k] = (stirling2[n-1][k-1] + (1LL * k * stirling2[n-1][k]) % MOD) % MOD;
        }
    }
}

// Using inclusion-exclusion
```

```cpp
int stirling2_fast(int n, int k) {
    int res = 0;
    for (int i = 0; i <= k; i++) {
        int term = (1LL * comb(k, i) * power(k - i, n)) % MOD;
        if (i % 2 == 0) {
            res = (res + term) % MOD;
        } else {
            res = (res - term + MOD) % MOD;
        }
    }
    return (1LL * res * inv(fact[k])) % MOD;
}
```

53.4 Generating Functions and Series

53.4.1 Binomial Theorem

```cpp
// Compute (x + y)^n mod MOD
int binomial_expansion(int x, int y, int n) {
    int res = 0;
    for (int k = 0; k <= n; k++) {
        int term = (1LL * comb(n, k) * power(x, n-k) % MOD * power(y, k)) % MOD;
        res = (res + term) % MOD;
    }
    return res;
}
```

53.4.2 Fibonacci Numbers

```cpp
vector<int> fib(MAXN);

void precompute_fibonacci() {
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i < MAXN; i++) {
        fib[i] = (fib[i-1] + fib[i-2]) % MOD;
    }
}

// Matrix exponentiation alternative
int fibonacci(int n) {
    if (n == 0) return 0;
    int a = 0, b = 1;
    for (int i = 2; i <= n; i++) {
        int c = (a + b) % MOD;
        a = b;
        b = c;
    }
    return b;
}
```

53.5 Inclusion-Exclusion Principle

```cpp
int inclusion_exclusion(vector<int>& primes, int n) {
    int k = primes.size();
    int total = 0;

    for (int mask = 1; mask < (1 << k); mask++) {
        int bits = __builtin_popcount(mask);
        int product = 1;

        for (int i = 0; i < k; i++) {
            if (mask & (1 << i)) {
                product *= primes[i];
                if (product > n) break;
            }
        }
        if (bits % 2 == 1) {
            total += n / product;
        } else {
            total -= n / product;
        }
    }

    return n - total;
}
```

53.6 Multinomial Coefficients

```
int multinomial(vector<int>& ks) {
    int n = accumulate(ks.begin(), ks.end(), 0);
    int res = fact[n];
    for (int k : ks) {
        res = (1LL * res * inv(fact[k])) % MOD;
    }
    return res;
}
```

53.7 Burnside's Lemma (Counting Distinct Objects)

```
int count_distinct_colorings(int n, int k) {
    // For cyclic group C_n (rotations)
    int total = 0;
    for (int i = 1; i <= n; i++) {
        total = (total + power(k, __gcd(i, n))) % MOD;
    }
    return (1LL * total * inv(n)) % MOD;
}
```

These combinatorial algorithms and formulas provide powerful tools for solving counting problems efficiently. The implementations are optimized for performance and include modular arithmetic operations for handling large numbers.

54 Fast Exponentiation Techniques in Number Theory

Fast exponentiation (also known as exponentiation by squaring) is a fundamental algorithm for efficiently computing large powers of numbers, especially in modular arithmetic. Here's a comprehensive guide with C++ implementations.

54.1 Basic Binary Exponentiation (Iterative)

```cpp
// Compute a^b
long long power(long long a, long long b) {
    long long result = 1;
    while (b > 0) {
        if (b & 1)  // If b is odd
            result *= a;
        a *= a;
        b >>= 1;     // Divide b by 2
    }
    return result;
}

// Compute a^b mod m
long long power_mod(long long a, long long b, long long m) {
    a %= m;
    long long result = 1;
    while (b > 0) {
        if (b & 1)
            result = (result * a) % m;
        a = (a * a) % m;
        b >>= 1;
    }
    return result;
}
```

54.2 Recursive Implementation

```cpp
long long power_recursive(long long a, long long b) {
    if (b == 0) return 1;
    long long temp = power_recursive(a, b/2);
    if (b % 2 == 0)
        return temp * temp;
    else
        return a * temp * temp;
}

long long power_mod_recursive(long long a, long long b, long long m) {
    if (b == 0) return 1 % m;
    a %= m;
    long long temp = power_mod_recursive(a, b/2, m);
    temp = (temp * temp) % m;
    if (b % 2 == 0)
        return temp;
    else
        return (a * temp) % m;
}
```

54.3 Modular Exponentiation with Safe Multiplication

For cases where intermediate results might overflow even 64-bit integers:

```cpp
// Safe multiplication under modulo
long long mul_mod(long long a, long long b, long long m) {
    a %= m;
    b %= m;
    long long res = 0;
    while (b > 0) {
        if (b & 1)
            res = (res + a) % m;
        a = (a << 1) % m;
        b >>= 1;
    }
    return res;
}

long long power_mod_safe(long long a, long long b, long long m) {
    a %= m;
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = mul_mod(res, a, m);
        a = mul_mod(a, a, m);
        b >>= 1;
    }
    return res;
}
```

54.4 Matrix Exponentiation

For fast computation of linear recurrences:

```
const int MAT_SIZE = 2;
const long long MOD = 1e9+7;

struct Matrix {
    long long mat[MAT_SIZE][MAT_SIZE];
    Matrix() { memset(mat, 0, sizeof(mat)); }
};

Matrix matMul(Matrix a, Matrix b) {
    Matrix res;
    for (int i = 0; i < MAT_SIZE; i++)
        for (int j = 0; j < MAT_SIZE; j++)
            for (int k = 0; k < MAT_SIZE; k++)
                res.mat[i][j] = (res.mat[i][j] + a.mat[i][k] * b.mat[k][j]) % MOD;
    return res;
}

Matrix matPow(Matrix base, long long power) {
    Matrix res;
    for (int i = 0; i < MAT_SIZE; i++)
        res.mat[i][i] = 1;

    while (power > 0) {
        if (power & 1)
            res = matMul(res, base);
        base = matMul(base, base);
        power >>= 1;
    }
    return res;
}
```

54.5 Applications

Fibonacci Numbers in $O(\log n)$ time

```
long long fibonacci(long long n) {
    if (n == 0) return 0;
    Matrix mat;
    mat.mat[0][0] = 1; mat.mat[0][1] = 1;
    mat.mat[1][0] = 1; mat.mat[1][1] = 0;

    mat = matPow(mat, n - 1);
    return mat.mat[0][0];
}
```

Modular Multiplicative Inverse using Fermat's Little Theorem

```
long long modInverse(long long a, long long mod) {
    return power_mod(a, mod - 2, mod);
}
```

54.6 Performance Considerations

- Time Complexity: All versions run in $O(\log n)$ time
- Space Complexity: $O(1)$ for iterative, $O(\log n)$ stack space for recursive
- Iterative vs Recursive: Iterative is generally preferred for performance
- Modulo Operations: Can be expensive – minimize when possible

54.7 Special Cases Handling

```
long long power_optimized(long long a, long long b) {
    if (b == 0) return 1;
    if (a == 0) return 0;
    if (a == 1) return 1;
    if (b == 1) return a;

    if (b < 0) return 1 / power_optimized(a, -b);

    if (b <= 5) {
        long long res = 1;
        for (int i = 0; i < b; i++)
            res *= a;
        return res;
    }

    long long res = 1;
    while (b > 0) {
        if (b & 1) res *= a;
        a *= a;
        b >>= 1;
    }
    return res;
}
```

These fast exponentiation techniques are essential for efficient computation in number theory, cryptography, and competitive programming. The choice of implementation depends on your specific needs regarding modular arithmetic, overflow safety, and whether you're working with numbers or matrices.

55 GCD and LCM in Number Theory

55.1 Greatest Common Divisor (GCD)

The GCD of two numbers is the largest number that divides both of them without leaving a remainder.

55.1.1 Euclidean Algorithm (Basic)

```
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

55.1.2 Recursive Implementation

```
int gcd_recursive(int a, int b) {
    return b == 0 ? a : gcd_recursive(b, a % b);
}
```

55.1.3 Built-in C++17 GCD

```
#include <numeric>
int gcd = std::gcd(a, b);
```

55.1.4 Extended Euclidean Algorithm

Returns GCD and coefficients x, y such that $ax + by = \gcd(a, b)$

```
int extended_gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = extended_gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
```

55.2 Least Common Multiple (LCM)

The LCM of two numbers is the smallest number that is a multiple of both.

55.2.1 Using GCD

```
int lcm(int a, int b) {
    return (a / gcd(a, b)) * b;
}
```

55.2.2 Built-in C++17 LCM

```
#include <numeric>
int lcm = std::lcm(a, b);
```

55.3 GCD Properties and Applications

55.3.1 GCD of Multiple Numbers

```
int gcd_multiple(vector<int> numbers) {
    int result = numbers[0];
    for (int num : numbers) {
        result = gcd(result, num);
        if (result == 1) break;
    }
    return result;
}
```

55.3.2 LCM of Multiple Numbers

```
int lcm_multiple(vector<int> numbers) {
    int result = numbers[0];
    for (int num : numbers) {
        result = lcm(result, num);
    }
    return result;
}
```

55.3.3 Binary GCD (Stein's Algorithm)

More efficient for very large numbers

```
int binary_gcd(int a, int b) {
    if (a == 0) return b;
    if (b == 0) return a;

    int shift;
    for (shift = 0; ((a | b) & 1) == 0; ++shift) {
        a >>= 1;
        b >>= 1;
    }

    while ((a & 1) == 0)
        a >>= 1;

    do {
        while ((b & 1) == 0)
            b >>= 1;
        if (a > b)
            swap(a, b);
        b -= a;
    } while (b != 0);

    return a << shift;
}
```

55.4 Applications

55.4.1 Solving Linear Diophantine Equations

```
bool find_any_solution(int a, int b, int c, int &x0, int &y0, int &g) {
    g = extended_gcd(abs(a), abs(b), x0, y0);
    if (c % g)
        return false;

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}
```

55.4.2 Modular Multiplicative Inverse

```
int mod_inverse(int a, int m) {
    int x, y;
    int g = extended_gcd(a, m, x, y);
    if (g != 1)
        return -1; // No inverse exists
    else {
        x = (x % m + m) % m;
        return x;
    }
}
```

55.4.3 Reducing Fractions

```
void reduce_fraction(int &numerator, int &denominator) {
    int common = gcd(numerator, denominator);
    numerator /= common;
    denominator /= common;
}
```

55.5 Performance Considerations

- Time Complexity: $O(\log(\min(a, b)))$ for Euclidean algorithm
- Space Complexity: $O(1)$ for iterative, $O(\log n)$ for recursive
- Binary GCD: Often faster in practice due to bitwise operations
- Extended GCD: Essential for solving linear Diophantine equations

These GCD and LCM implementations form the foundation for many number theory algorithms and are widely used in competitive programming and cryptography applications.

56 Advanced Pattern Matching Algorithms

Pattern matching is a cornerstone of string processing with applications ranging from text search to bioinformatics. Let's explore advanced pattern matching techniques with optimized C++ implementations.

56.1 Aho-Corasick Algorithm (Multiple Patterns)

Efficiently searches for multiple patterns simultaneously using a trie with failure links.

```cpp
struct AhoCorasickNode {
    unordered_map<char, int> children;
    int fail = -1;
    vector<int> output;
};

class AhoCorasick {
    vector<AhoCorasickNode> trie;

public:
    AhoCorasick() {
        trie.emplace_back(); // root node
    }

    void insert(const string& pattern, int pattern_id) {
        int node = 0;
        for (char c : pattern) {
            if (!trie[node].children.count(c)) {
                trie[node].children[c] = trie.size();
                trie.emplace_back();
            }
            node = trie[node].children[c];
        }
        trie[node].output.push_back(pattern_id);
    }

    void build_failure_links() {
        queue<int> q;
        for (auto [c, child] : trie[0].children) {
            q.push(child);
        }

        while (!q.empty()) {
            int u = q.front(); q.pop();

            for (auto [c, v] : trie[u].children) {
                int fail = trie[u].fail;
                while (fail != -1 && !trie[fail].children.count(c)) {
                    fail = trie[fail].fail;
                }
                trie[v].fail = (fail == -1) ? 0 : trie[fail].children[c];
                trie[v].output.insert(trie[v].output.end(),
                                      trie[trie[v].fail].output.begin(),
                                      trie[trie[v].fail].output.end());
                q.push(v);
            }
        }
    }

    vector<vector<int>> search(const string& text) {
        vector<vector<int>> matches(trie.size());
        int node = 0;

        for (int i = 0; i < text.size(); i++) {
            char c = text[i];
            while (node != 0 && !trie[node].children.count(c)) {
                node = trie[node].fail;
            }
            if (trie[node].children.count(c)) {
                node = trie[node].children[c];
            }
            for (int pattern_id : trie[node].output) {
                matches[pattern_id].push_back(i);
            }
        }
        return matches;
    }
};
```

Time Complexity: $O(n + m + z)$ where n is text length, m is total pattern length, z is number of matches.

56.2 Suffix Automaton

Efficient for various string operations including pattern matching.

```
struct State {
    int len, link;
    unordered_map<char, int> next;
};

class SuffixAutomaton {
    vector<State> st;
    int last;

public:
    SuffixAutomaton() {
        st.emplace_back();
        st[0].len = 0;
        st[0].link = -1;
        last = 0;
    }

    void sa_extend(char c) {
        int p = last;
        int curr = st.size();
        st.emplace_back();
        st[curr].len = st[p].len + 1;

        while (p >= 0 && !st[p].next.count(c)) {
            st[p].next[c] = curr;
            p = st[p].link;
        }

        if (p == -1) {
            st[curr].link = 0;
        } else {
            int q = st[p].next[c];
            if (st[p].len + 1 == st[q].len) {
                st[curr].link = q;
            } else {
                int clone = st.size();
                st.push_back(st[q]);
                st[clone].len = st[p].len + 1;
                while (p >= 0 && st[p].next[c] == q) {
                    st[p].next[c] = clone;
                    p = st[p].link;
                }
                st[q].link = st[curr].link = clone;
            }
        }
        last = curr;
    }

    bool is_substring(const string& pattern) {
        int node = 0;
        for (char c : pattern) {
            if (!st[node].next.count(c)) return false;
            node = st[node].next[c];
        }
        return true;
    }
};
```

Time Complexity: $O(n)$ construction, $O(m)$ search per pattern.

56.3 Bit-Parallel (Shift-Or) Algorithm

Efficient for small patterns (word size).

```
vector<int> shift_or_search(const string& text, const string& pattern) {
    vector<int> matches;
    int m = pattern.size();
    if (m == 0) return matches;

    const int WORD_BITS = sizeof(uint64_t) * 8;
    if (m > WORD_BITS) {
        cerr << "Pattern too long for Shift-Or algorithm" << endl;
        return matches;
    }

    uint64_t mask[256];
    fill_n(mask, 256, ~0ULL);
    for (int i = 0; i < m; i++) {
        mask[pattern[i]] &= ~(1ULL << i);
    }

    uint64_t state = ~0ULL;
    uint64_t goal = 1ULL << (m - 1);

    for (int i = 0; i < text.size(); i++) {
        state = (state << 1) | mask[text[i]];
        if ((state & goal) == 0) {
            matches.push_back(i - m + 1);
        }
    }
    return matches;
}
```

Time Complexity: $O(n)$ with very small constant factor.

56.4 FM-Index (Compressed Suffix Array)

Space-efficient for large texts using Burrows-Wheeler Transform.

```
class FMIndex {
    string bwt;
    vector<int> sa;
    unordered_map<char, int> C;
    vector<vector<int>> Occ;

public:
    FMIndex(const string& text) {
        int n = text.size();
        sa.resize(n);
        iota(sa.begin(), sa.end(), 0);
        sort(sa.begin(), sa.end(), [&](int i, int j) {
            return text.substr(i) < text.substr(j);
        });

        bwt.resize(n);
        for (int i = 0; i < n; i++) {
            bwt[i] = (sa[i] == 0) ? '$' : text[sa[i] - 1];
        }

        string sorted = text;
        sort(sorted.begin(), sorted.end());
        sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());

        for (char c : sorted) {
            C[c] = lower_bound(sorted.begin(), sorted.end(), c) - sorted.begin();
        }

        Occ.resize(sorted.size(), vector<int>(n + 1, 0));
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < sorted.size(); j++) {
                Occ[j][i + 1] = Occ[j][i] + (bwt[i] == sorted[j]);
            }
        }
    }

    vector<int> search(const string& pattern) {
        int m = pattern.size();
        if (m == 0) return {};

        int sp = 0, ep = bwt.size();
        for (int i = m - 1; i >= 0; i--) {
            char c = pattern[i];
            if (!C.count(c)) return {};

            int rank_c = C.at(c);
            sp = C.at(c) + Occ[rank_c][sp];
            ep = C.at(c) + Occ[rank_c][ep];

            if (sp >= ep) return {};
        }

        vector<int> matches;
        for (int i = sp; i < ep; i++) {
            matches.push_back(sa[i]);
        }
        sort(matches.begin(), matches.end());
        return matches;
    }
};
```

Time Complexity: $O(m)$ search time after $O(n \log n)$ construction.

56.5 Algorithm Selection Guide

- Multiple patterns: Aho-Corasick
- General purpose with suffix operations: Suffix Automaton
- Small patterns: Bit-Parallel (Shift-Or)
- Large static texts: FM-Index
- Genomic data: BWT-based methods

These advanced algorithms provide efficient solutions for various pattern matching scenarios, from searching multiple patterns simultaneously to handling massive texts with compressed indexes.

57 Trie (Prefix Tree) Data Structure and Algorithms

A trie (pronounced "try") is a tree-like data structure that stores strings by breaking them into characters. It's particularly efficient for prefix-based searches and dictionary implementations.

57.1 Basic Trie Implementation

```
class TrieNode {
public:
    unordered_map<char, TrieNode*> children;
    bool isEndOfWord;

    TrieNode() : isEndOfWord(false) {}
};

class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }

    void insert(const string& word) {
        TrieNode* current = root;
        for (char c : word) {
            if (current->children.find(c) == current->children.end()) {
                current->children[c] = new TrieNode();
            }
            current = current->children[c];
        }
        current->isEndOfWord = true;
    }

    bool search(const string& word) {
        TrieNode* current = root;
        for (char c : word) {
            if (current->children.find(c) == current->children.end()) {
                return false;
            }
            current = current->children[c];
        }
        return current->isEndOfWord;
    }

    bool startsWith(const string& prefix) {
        TrieNode* current = root;
        for (char c : prefix) {
            if (current->children.find(c) == current->children.end()) {
                return false;
            }
            current = current->children[c];
        }
        return true;
    }
};
```

57.2 Advanced Trie Operations

57.2.1 Word Deletion

```
bool deleteHelper(TrieNode* current, const string& word, int index) {
    if (index == word.length()) {
        if (!current->isEndOfWord) return false;
        current->isEndOfWord = false;
        return current->children.empty();
    }
    char c = word[index];
    if (current->children.find(c) == current->children.end()) return false;

    bool shouldDeleteChild = deleteHelper(current->children[c], word, index + 1);

    if (shouldDeleteChild) {
        delete current->children[c];
        current->children.erase(c);
        return current->children.empty() && !current->isEndOfWord;
    }
    return false;
}

void deleteWord(const string& word) {
    deleteHelper(root, word, 0);
}
```

57.2.2 Count Words with Prefix

```
int countWordsWithPrefix(const string& prefix) {
    TrieNode* current = root;
    for (char c : prefix) {
        if (current->children.find(c) == current->children.end()) {
            return 0;
        }
        current = current->children[c];
    }
    return countWords(current);
}

int countWords(TrieNode* node) {
    int count = 0;
    if (node->isEndOfWord) count++;
    for (auto& pair : node->children) {
        count += countWords(pair.second);
    }
    return count;
}
```

57.2.3 Autocomplete Suggestions

```
void getSuggestions(TrieNode* node, string prefix, vector<string>& suggestions) {
    if (node->isEndOfWord) {
        suggestions.push_back(prefix);
    }
    for (auto& pair : node->children) {
        getSuggestions(pair.second, prefix + pair.first, suggestions);
    }
}

vector<string> autocomplete(const string& prefix) {
    vector<string> suggestions;
    TrieNode* current = root;

    for (char c : prefix) {
        if (current->children.find(c) == current->children.end()) {
            return suggestions;
        }
        current = current->children[c];
    }

    getSuggestions(current, prefix, suggestions);
    return suggestions;
}
```

57.3 Optimized Trie Variants

57.3.1 Compressed Trie (Radix Tree)

```
class RadixNode {
public:
    string fragment;
    unordered_map<char, RadixNode*> children;
    bool isEndOfWord;

    RadixNode(const string& f = "") : fragment(f), isEndOfWord(false) {}
};

class RadixTrie {
private:
    RadixNode* root;

    int commonPrefixLength(const string& s1, const string& s2) {
        int minLen = min(s1.length(), s2.length());
        for (int i = 0; i < minLen; i++) {
            if (s1[i] != s2[i]) return i;
        }
        return minLen;
    }

public:
    RadixTrie() {
        root = new RadixNode();
    }

    void insert(const string& word) {
        RadixNode* current = root;
        int i = 0;

        while (i < word.length()) {
            char c = word[i];
            if (current->children.find(c) == current->children.end()) {
                current->children[c] = new RadixNode(word.substr(i));
                current->children[c]->isEndOfWord = true;
                return;
            }

            RadixNode* child = current->children[c];
            int commonLen = commonPrefixLength(child->fragment, word.substr(i));

            if (commonLen < child->fragment.length()) {
                RadixNode* splitNode = new RadixNode(child->fragment.substr(commonLen));
                splitNode->children = child->children;
                splitNode->isEndOfWord = child->isEndOfWord;

                child->fragment = child->fragment.substr(0, commonLen);
                child->children.clear();
                child->children[splitNode->fragment[0]] = splitNode;
                child->isEndOfWord = false;
            }

            i += commonLen;
            current = child;
        }
        current->isEndOfWord = true;
    }
};
```

57.3.2 Ternary Search Trie (TST)

```
class TSTNode {
public:
    char data;
    bool isEndOfWord;
    TSTNode *left, *middle, *right;
    TSTNode(char c) : data(c), isEndOfWord(false), left(nullptr),
                     middle(nullptr), right(nullptr) {}
};

class TernarySearchTrie {
private:
    TSTNode* root;

    TSTNode* insertHelper(TSTNode* node, const string& word, int index) {
        char c = word[index];

        if (!node) {
            node = new TSTNode(c);
        }

        if (c < node->data) {
            node->left = insertHelper(node->left, word, index);
        } else if (c > node->data) {
            node->right = insertHelper(node->right, word, index);
        } else {
            if (index < word.length() - 1) {
                node->middle = insertHelper(node->middle, word, index + 1);
            } else {
                node->isEndOfWord = true;
            }
        }
        return node;
    }

public:
    TernarySearchTrie() : root(nullptr) {}

    void insert(const string& word) {
        root = insertHelper(root, word, 0);
    }

    bool search(const string& word) {
        TSTNode* current = root;
        int i = 0;

        while (current && i < word.length()) {
            char c = word[i];

            if (c < current->data) {
                current = current->left;
            } else if (c > current->data) {
                current = current->right;
            } else {
                if (i == word.length() - 1) {
                    return current->isEndOfWord;
                }
                current = current->middle;
                i++;
            }
        }
        return false;
    }
};
```

57.4 Applications of Tries

57.4.1 Spell Checker

```
class SpellChecker {
private:
    Trie dictionary;
    unordered_set<string> knownWords;

public:
    void buildDictionary(const vector<string>& words) {
        for (const string& word : words) {
            dictionary.insert(word);
            knownWords.insert(word);
        }
    }

    bool isCorrect(const string& word) {
        return knownWords.find(word) != knownWords.end();
    }

    vector<string> suggestCorrections(const string& word, int maxDistance) {
        vector<string> suggestions;
        queue<pair<string, TrieNode*>> q;
        q.push({"", dictionary.getRoot()});

        while (!q.empty()) {
            auto current = q.front();
            q.pop();
            string currentWord = current.first;
            TrieNode* currentNode = current.second;

            if (currentWord.length() == word.length()) {
                if (currentNode->isEndOfWord &&
                    editDistance(currentWord, word) <= maxDistance) {
                    suggestions.push_back(currentWord);
                }
                continue;
            }

            char nextChar = word[currentWord.length()];
            for (auto& child : currentNode->children) {
                string newWord = currentWord + child.first;
                int dist = editDistance(newWord, word.substr(0, newWord.length()));
                if (dist <= maxDistance) {
                    q.push({newWord, child.second});
                }
            }
        }
        return suggestions;
    }
};
```

57.4.2 IP Routing (Longest Prefix Match)

```
class IPRouter {
private:
    Trie trie;

public:
    void addRoute(const string& prefix, const string& nextHop) {
        trie.insert(prefix);
        // Store nextHop information in the terminal node
    }

    string findBestMatch(const string& ip) {
        string bestMatch;
        TrieNode* current = trie.getRoot();
        string currentPrefix;

        for (char c : ip) {
            if (current->children.find(c) == current->children.end()) {
                break;
            }
            current = current->children[c];
            currentPrefix += c;
            if (current->isEndOfWord) {
                bestMatch = currentPrefix;
            }
        }
        return bestMatch;
    }
};
```

57.5 Performance Characteristics

- **Insertion**: $O(L)$, where L is the length of the word
- **Search**: $O(L)$ for exact search
- **Prefix Search**: $O(L)$ to find the prefix node, then $O(K)$ to collect all matches
- **Space**: $O(N \cdot M)$, where N is number of words and M is average word length

Tries are particularly useful when:

- You need to find all words with a common prefix
- You need to implement autocomplete functionality
- You're working with a dictionary of words
- You need to perform prefix-based searches efficiently

58 Suffix Structures in String Algorithms

Suffix structures are powerful tools for solving complex string processing problems efficiently. Here's a comprehensive guide to suffix arrays, suffix trees, and suffix automata with optimized C++ implementations.

58.1 Suffix Array

58.1.1 Basic Construction ($O(n^2 \log n)$)

```
vector<int> buildSuffixArray(const string& s) {
    int n = s.size();
    vector<int> sa(n);
    iota(sa.begin(), sa.end(), 0);

    vector<int> rank(s.begin(), s.end());

    for (int k = 1; k < n; k *= 2) {
        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;
        };
        sort(sa.begin(), sa.end(), cmp);

        vector<int> new_rank(n);
        new_rank[sa[0]] = 0;
        for (int i = 1; i < n; i++) {
            new_rank[sa[i]] = new_rank[sa[i-1]] + (cmp(sa[i-1], sa[i]) ? 1 : 0);
        }
        rank = move(new_rank);
    }
    return sa;
}
```

58.1.2 Kasai's Algorithm for LCP Array ($O(n)$)

```
vector<int> buildLCP(const string& s, const vector<int>& sa) {
    int n = s.size();
    vector<int> lcp(n, 0);
    vector<int> rank(n, 0);

    for (int i = 0; i < n; i++)
        rank[sa[i]] = i;

    int h = 0;
    for (int i = 0; i < n; i++) {
        if (rank[i] > 0) {
            int j = sa[rank[i] - 1];
            while (i + h < n && j + h < n && s[i+h] == s[j+h])
                h++;
            lcp[rank[i]] = h;
            if (h > 0) h--;
        }
    }
    return lcp;
}
```

58.2 Suffix Tree

58.2.1 Ukkonen's Algorithm ($O(n)$)

```
class SuffixTreeNode {
public:
    unordered_map<char, SuffixTreeNode*> children;
    SuffixTreeNode* suffixLink = nullptr;
    int start = -1;
    int* end = nullptr;
    int suffixIndex = -1;

    ~SuffixTreeNode() {
        for (auto& child : children)
            delete child.second;
    }
};
...
```

58.3 Suffix Automaton

58.3.1 Efficient Construction (O(n))

```
class State {
public:
    int len;
    int link;
    unordered_map<char, int> next;
};

class SuffixAutomaton {
    vector<State> states;
    int last;

public:
    SuffixAutomaton() {
        states.emplace_back();
        states[0].len = 0;
        states[0].link = -1;
        last = 0;
    }
...
```

58.4 Applications

58.4.1 Longest Common Substring

```
string longestCommonSubstring(const string& s1, const string& s2) {
    SuffixAutomaton sam;
    for (char c : s1)
        sam.sa_extend(c);
    ...
}
```

58.4.2 Pattern Searching

```
vector<int> findPattern(const string& text, const string& pattern,
                        const vector<int>& suffixArray) {
    ...
}
```

58.4.3 Longest Repeated Substring

```
string longestRepeatedSubstring(const string& s) {
    ...
}
```

58.5 Performance Comparison

Structure	Construction Time	Space	Pattern Search	LCS Time
Suffix Array	$O(n \log n)$	$O(n)$	$O(m \log n)$	$O(n)$
Suffix Tree	$O(n)$	$O(n)$	$O(m)$	$O(n)$
Suffix Automaton	$O(n)$	$O(n)$	$O(m)$	$O(n)$

Key Points:

- Suffix arrays are simpler to implement and more space-efficient.
- Suffix trees provide fastest pattern searches but use more memory.
- Suffix automata offer a good balance and additional functionality.
- All structures enable solving complex string problems efficiently.

59 Manacher's Algorithm for Longest Palindromic Substring

Manacher's algorithm is the most efficient method to find the longest palindromic substring in a string, with linear time complexity $O(n)$. Here's a comprehensive implementation and explanation.

59.1 Algorithm Overview

1. **Transform the string** to handle even-length palindromes by inserting special characters (usually #) between each character.
2. **Maintain a palindrome radius array** (P) where $P[i]$ represents the radius of the longest palindrome centered at i.
3. **Utilize symmetry properties** to avoid redundant computations.
4. **Track the center and right boundary** of the current rightmost palindrome.

59.2 Complete C++ Implementation

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

using namespace std;

string preprocess(const string& s) {
    string result = "#";
    for (char c : s) {
        result += c;
        result += '#';
    }
    return result;
}

string findLongestPalindrome(const string& s) {
    if (s.empty()) return "";

    string T = preprocess(s);
    int n = T.size();
    vector<int> P(n, 0);

    int C = 0; // center of the current palindrome
    int R = 0; // right boundary of the current palindrome

    for (int i = 1; i < n-1; i++) {
        int mirror = 2*C - i;

        if (i < R) {
            P[i] = min(R - i, P[mirror]);
        }

        while (T[i + (1 + P[i])] == T[i - (1 + P[i])]) {
            P[i]++;
        }

        if (i + P[i] > R) {
            C = i;
            R = i + P[i];
        }
    }

    int max_len = 0;
    int center_index = 0;
    for (int i = 1; i < n-1; i++) {
        if (P[i] > max_len) {
            max_len = P[i];
            center_index = i;
        }
    }
    int start = (center_index - max_len) / 2;
    return s.substr(start, max_len);
}
```

59.3 Optimized Version (Space Efficient)

```
string longestPalindrome(string s) {
    if (s.empty()) return "";

    string T = "^#";
    for (char c : s) {
        T += c;
        T += '#';
    }
    T += "$";

    int n = T.size();
    vector<int> P(n, 0);
    int C = 0, R = 0;

    for (int i = 1; i < n-1; i++) {
        int i_mirror = 2*C - i;

        P[i] = (R > i) ? min(R-i, P[i_mirror]) : 0;

        while (T[i + 1 + P[i]] == T[i - 1 - P[i]]) {
            P[i]++;
        }

        if (i + P[i] > R) {
            C = i;
            R = i + P[i];
        }
    }

    int max_len = 0;
    int center_index = 0;
    for (int i = 1; i < n-1; i++) {
        if (P[i] > max_len) {
            max_len = P[i];
            center_index = i;
        }
    }

    int start = (center_index - max_len) / 2;
    return s.substr(start, max_len);
}
```

59.4 Key Components Explained

- **Preprocessing**: Converts `"abc"` into `"^#a#b#c#$"` to handle even and odd length palindromes uniformly.
- **Palindrome Array (P)**: $P[i]$ stores the length of the palindrome centered at $T[i]$.
- **Mirror Property**: For $i < R$, we can reuse information from the mirrored position $2C - i$.
- **Expansion**: When symmetry cannot be used, expand around the center and update C and R if a longer palindrome is found.

59.5 Applications

59.5.1 Count All Palindromic Substrings

```
int countPalindromicSubstrings(const string& s) {
    string T = preprocess(s);
    int n = T.size();
    vector<int> P(n, 0);
    int C = 0, R = 0;
    int count = 0;

    for (int i = 1; i < n-1; i++) {
        int mirror = 2*C - i;

        P[i] = (R > i) ? min(R-i, P[mirror]) : 0;

        while (T[i + 1 + P[i]] == T[i - 1 - P[i]]) {
            P[i]++;
        }

        if (i + P[i] > R) {
            C = i;
            R = i + P[i];
        }

        count += (P[i] + 1) / 2;
    }
    return count;
}
```

59.5.2 Find All Distinct Palindromic Substrings

```
unordered_set<string> allPalindromicSubstrings(const string& s) {
    unordered_set<string> palindromes;
    string T = preprocess(s);
    int n = T.size();
    vector<int> P(n, 0);
    int C = 0, R = 0;

    for (int i = 1; i < n-1; i++) {
        int mirror = 2*C - i;

        P[i] = (R > i) ? min(R-i, P[mirror]) : 0;

        while (T[i + 1 + P[i]] == T[i - 1 - P[i]]) {
            P[i]++;
        }

        if (i + P[i] > R) {
            C = i;
            R = i + P[i];
        }

        int start = (i - P[i]) / 2;
        for (int len = 1; len <= P[i]; len++) {
            palindromes.insert(s.substr(start, len));
        }
    }
    return palindromes;
}
```

59.6 Performance Analysis

- **Time Complexity**: $O(n)$ — Each character is compared at most twice.
- **Space Complexity**: $O(n)$ — For the transformed string and the radius array.
- **Advantages**:
  - Most efficient known algorithm for this problem.
  - Handles even and odd palindromes uniformly.
  - Easily adapted to related problems.

60 Greedy Algorithms: Theory and Applications

Greedy algorithms make **locally optimal choices at each step** with the hope of finding a **globally optimal solution**. They are simple, efficient, and widely used in optimization problems where a **greedy choice leads to the best solution**.

60.1 Key Properties of Greedy Algorithms

Greedy Choice Property
A *locally optimal choice* leads to a globally optimal solution.
No reconsideration of previous choices is needed.

Optimal Substructure
The problem can be broken into smaller subproblems.
The optimal solution to the problem contains optimal solutions to subproblems.

60.2 When to Use Greedy Algorithms?

- Coin Change Problem (with certain coin denominations)
- Fractional Knapsack Problem
- Interval Scheduling (Activity Selection)
- Huffman Coding (Data Compression)
- Minimum Spanning Tree (Prim's & Kruskal's Algorithms)
- Dijkstra's Shortest Path Algorithm

60.3 Classic Greedy Problems and C++ Implementations

Activity Selection Problem
Problem: Select the maximum number of non-overlapping activities.

```cpp
#include <vector>
#include <algorithm>
using namespace std;

struct Activity {
    int start, finish;
};

bool compare(Activity a1, Activity a2) {
    return a1.finish < a2.finish;
}

vector<Activity> selectMaxActivities(vector<Activity>& activities) {
    sort(activities.begin(), activities.end(), compare);
    vector<Activity> selected;
    selected.push_back(activities[0]);

    for (int i = 1; i < activities.size(); i++) {
        if (activities[i].start >= selected.back().finish) {
            selected.push_back(activities[i]);
        }
    }
    return selected;
}
```

Time Complexity: $O(n \log n)$

Fractional Knapsack Problem
Problem: Maximize value in a knapsack with fractional items.

```cpp
#include <vector>
#include <algorithm>
using namespace std;

struct Item {
    int weight, value;
    double ratio; // value per unit weight
};

bool compare(Item a, Item b) {
    return a.ratio > b.ratio;
}

double fractionalKnapsack(int W, vector<Item>& items) {
    for (auto &item : items) {
        item.ratio = (double)item.value / item.weight;
    }
    sort(items.begin(), items.end(), compare);

    double maxValue = 0.0;
    for (const auto &item : items) {
        if (W >= item.weight) {
            maxValue += item.value;
            W -= item.weight;
        } else {
            maxValue += item.ratio * W;
            break;
        }
    }
    return maxValue;
}
```

Time Complexity: $O(n \log n)$

Huffman Coding (Data Compression)

Problem: Assign variable-length codes to minimize total encoded size.

```cpp
#include <queue>
#include <unordered_map>
using namespace std;

struct HuffmanNode {
    char data;
    int freq;
    HuffmanNode *left, *right;
    HuffmanNode(char d, int f) : data(d), freq(f), left(nullptr), right(nullptr) {}
};

struct Compare {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->freq > b->freq;
    }
};

void generateCodes(HuffmanNode* root, string code, unordered_map<char, string>& huffmanCodes) {
    if (!root) return;
    if (!root->left && !root->right) {
        huffmanCodes[root->data] = code;
    }
    generateCodes(root->left, code + "0", huffmanCodes);
    generateCodes(root->right, code + "1", huffmanCodes);
}

unordered_map<char, string> buildHuffmanCodes(string text) {
    unordered_map<char, int> freq;
    for (char c : text) freq[c]++;

    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> minHeap;
    for (auto &pair : freq) {
        minHeap.push(new HuffmanNode(pair.first, pair.second));
    }
    while (minHeap.size() > 1) {
        HuffmanNode* left = minHeap.top(); minHeap.pop();
        HuffmanNode* right = minHeap.top(); minHeap.pop();
        HuffmanNode* newNode = new HuffmanNode('$', left->freq + right->freq);
        newNode->left = left;
        newNode->right = right;
        minHeap.push(newNode);
    }

    HuffmanNode* root = minHeap.top();
    unordered_map<char, string> huffmanCodes;
    generateCodes(root, "", huffmanCodes);
    return huffmanCodes;
}
```

Time Complexity: $O(n \log n)$

Minimum Number of Coins (Coin Change Problem)
Problem: Find the minimum number of coins to make a given amount.

```
#include <vector>
#include <algorithm>
using namespace std;

int minCoins(vector<int>& coins, int amount) {
    sort(coins.begin(), coins.end(), greater<int>());
    int count = 0;
    for (int coin : coins) {
        while (amount >= coin) {
            amount -= coin;
            count++;
        }
    }
    return (amount == 0) ? count : -1;
}
```

Note: Works only for certain coin systems (e.g., US coins).
Time Complexity: $O(n \log n)$

60.4 When Greedy Fails

Greedy algorithms **do not always work**—they fail when:

- The problem requires **reconsidering past choices** (e.g., 0/1 Knapsack).
- The greedy choice does **not guarantee global optimality** (e.g., Coin Change with arbitrary denominations).

Alternative: Use **Dynamic Programming** when greedy fails.

60.5 Conclusion

- **Greedy algorithms are fast and intuitive** but require proof of correctness.
- **Best for problems with optimal substructure and greedy-choice property**.
- **Common applications:** Scheduling, Compression, Shortest Path, MST.

61 Mo's Algorithm: An Advanced Technique for Competitive Programming

Mo's Algorithm is an elegant and powerful technique for efficiently answering range queries on arrays or trees when queries can be processed offline. It's particularly useful in competitive programming for problems involving multiple range queries where the order of processing can be adjusted for better performance.

61.1 Key Concepts

Basic Idea: Mo's Algorithm processes range queries in a specific order to minimize the number of operations needed to adjust the current range when moving from one query to the next.

Complexity:

- Time: $\mathcal{O}((N+Q)\sqrt{N})$
- Space: $\mathcal{O}(N)$

61.2 Algorithm Overview

1. Divide the array into blocks of size $\sqrt{N}$
2. Sort all queries:
   - Primary key: block number of the query's left endpoint
   - Secondary key: right endpoint (ascending if left block is even, descending if odd — "wiggle" optimization)
3. Process queries in sorted order, maintaining a current `[L, R]` range and adjusting it incrementally for each query

61.3 Implementation

```cpp
#include <bits/stdc++.h>
using namespace std;

const int BLOCK_SIZE = 317; // ~sqrt(1e5)

struct Query {
    int l, r, idx;
    bool operator<(Query other) const {
        if (l/BLOCK_SIZE != other.l/BLOCK_SIZE)
            return l < other.l;
        return (l/BLOCK_SIZE & 1) ? (r < other.r) : (r > other.r);
    }
};

void mo_algorithm(vector<int>& arr, vector<Query>& queries) {
    sort(queries.begin(), queries.end());

    int curr_l = 0, curr_r = -1;
    // Initialize your answer variable(s) here

    vector<int> answers(queries.size());

    for (Query q : queries) {
        while (curr_l > q.l) {
            curr_l--;
            add_element(arr[curr_l]);
        }
        while (curr_r < q.r) {
            curr_r++;
            add_element(arr[curr_r]);
        }
        while (curr_l < q.l) {
            remove_element(arr[curr_l]);
            curr_l++;
        }
        while (curr_r > q.r) {
            remove_element(arr[curr_r]);
            curr_r--;
        }
        answers[q.idx] = get_answer();
    }

    for (int ans : answers) {
        cout << ans << "\n";
    }
}
```

61.4 Key Components to Implement

- `add_element(x)`: Updates your data structures when expanding the current range to include `x`
- `remove_element(x)`: Updates your data structures when shrinking the current range to exclude `x`
- `get_answer()`: Returns the answer for the current range

61.5 Example: Count Distinct Elements in Range

```
vector<int> freq;
int distinct;

void add_element(int x) {
    if (freq[x] == 0) distinct++;
    freq[x]++;
}

void remove_element(int x) {
    freq[x]--;
    if (freq[x] == 0) distinct--;
}

int get_answer() {
    return distinct;
}

void solve() {
    int n, q;
    cin >> n >> q;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    freq.assign(*max_element(arr.begin(), arr.end()) + 1, 0);
    distinct = 0;

    vector<Query> queries(q);
    for (int i = 0; i < q; i++) {
        cin >> queries[i].l >> queries[i].r;
        queries[i].l--; queries[i].r--;
        queries[i].idx = i;
    }

    mo_algorithm(arr, queries);
}
```

61.6 Optimizations

- **Block Size:** Use $N/\sqrt{2Q}$ for better cache locality
- **Wiggle Order:** Reduces total movement of the right pointer
- **Coordinate Compression:** Compress large value ranges
- **Precomputing:** Use prefix sums or other auxiliary data

61.7 When to Use Mo's Algorithm

- Offline range queries
- Need to maintain statistics over a range
- Counting frequencies or unique elements
- When segment tree ($O(Q \log N)$) is too slow

61.8 Practice Problems

1. DQUERY (SPOJ) – Count distinct elements in a range
2. Powerful Array (Codeforces) – Compute weighted frequency sums
3. XOR and Favorite Number (Codeforces) – Count subarrays by XOR
4. Tree and Queries (Codeforces) – Mo's algorithm on trees

Mastering Mo's Algorithm can give you a significant advantage in contests, especially for problems with many range queries.

62 Segment Trees & Lazy Propagation: Advanced Techniques for Competitive Programming

Segment Trees are one of the most powerful data structures in competitive programming, enabling efficient range queries and updates. When combined with Lazy Propagation, they become even more versatile for handling range update operations.

62.1 Segment Tree Fundamentals

Key Properties

- Full binary tree structure where each node represents an interval
- Leaf nodes represent single elements
- Internal nodes store aggregated information about their children's intervals
- $\mathcal{O}(n)$ space complexity
- $\mathcal{O}(\log n)$ time complexity for both queries and point updates

Basic Operations

1. Build: Construct the segment tree from an array
2. Query: Get aggregate information about a range
3. Update: Modify a single element (point update)

62.2 Basic Segment Tree Implementation (Range Sum Query)

```cpp
class SegmentTree {
private:
    vector<int> tree;
    int n;

    void build(const vector<int>& arr, int node, int start, int end) {
        if (start == end) {
            tree[node] = arr[start];
        } else {
            int mid = (start + end) / 2;
            build(arr, 2*node, start, mid);
            build(arr, 2*node+1, mid+1, end);
            tree[node] = tree[2*node] + tree[2*node+1];
        }
    }

    int query(int node, int node_start, int node_end, int l, int r) {
        if (r < node_start || l > node_end) return 0;
        if (l <= node_start && node_end <= r) return tree[node];

        int mid = (node_start + node_end) / 2;
        return query(2*node, node_start, mid, l, r) +
               query(2*node+1, mid+1, node_end, l, r);
    }

    void update(int node, int node_start, int node_end, int idx, int val) {
        if (node_start == node_end) {
            tree[node] = val;
        } else {
            int mid = (node_start + node_end) / 2;
            if (idx <= mid) {
                update(2*node, node_start, mid, idx, val);
            } else {
                update(2*node+1, mid+1, node_end, idx, val);
            }
            tree[node] = tree[2*node] + tree[2*node+1];
        }
    }
public:
    SegmentTree(const vector<int>& arr) {
        n = arr.size();
        tree.resize(4*n);
        build(arr, 1, 0, n-1);
    }

    int range_query(int l, int r) {
        return query(1, 0, n-1, l, r);
    }

    void point_update(int idx, int val) {
        update(1, 0, n-1, idx, val);
    }
};
```

62.3 Lazy Propagation

Why We Need It

- Basic segment trees handle point updates efficiently
- Range updates would take $\mathcal{O}(n \log n)$ with point updates
- Lazy propagation enables $\mathcal{O}(\log n)$ range updates by deferring updates

How It Works

1. Postpone updates until necessary
2. Store pending updates in lazy arrays
3. Propagate updates only when needed (during queries or further updates)

62.4 Segment Tree with Lazy Propagation (Range Sum with Range Add)

```
class LazySegmentTree {
private:
    vector<int> tree, lazy;
    int n;

    void build(const vector<int>& arr, int node, int start, int end) {
        if (start == end) {
            tree[node] = arr[start];
        } else {
            int mid = (start + end) / 2;
            build(arr, 2*node, start, mid);
            build(arr, 2*node+1, mid+1, end);
            tree[node] = tree[2*node] + tree[2*node+1];
        }
    }

    void push(int node, int node_start, int node_end) {
        if (lazy[node] != 0) {
            tree[node] += (node_end - node_start + 1) * lazy[node];
            if (node_start != node_end) {
                lazy[2*node] += lazy[node];
                lazy[2*node+1] += lazy[node];
            }
            lazy[node] = 0;
        }
    }

    int query(int node, int node_start, int node_end, int l, int r) {
        push(node, node_start, node_end);
        if (r < node_start || l > node_end) return 0;
        if (l <= node_start && node_end <= r) return tree[node];

        int mid = (node_start + node_end) / 2;
        return query(2*node, node_start, mid, l, r) +
               query(2*node+1, mid+1, node_end, l, r);
    }

    void range_update(int node, int node_start, int node_end, int l, int r, int val) {
        push(node, node_start, node_end);
        if (r < node_start || l > node_end) return;
        if (l <= node_start && node_end <= r) {
            lazy[node] += val;
            push(node, node_start, node_end);
            return;
        }

        int mid = (node_start + node_end) / 2;
        range_update(2*node, node_start, mid, l, r, val);
        range_update(2*node+1, mid+1, node_end, l, r, val);
        tree[node] = tree[2*node] + tree[2*node+1];
    }

public:
    LazySegmentTree(const vector<int>& arr) {
        n = arr.size();
        tree.resize(4*n);
        lazy.resize(4*n, 0);
        build(arr, 1, 0, n-1);
    }

    int range_query(int l, int r) {
        return query(1, 0, n-1, l, r);
    }

    void range_add(int l, int r, int val) {
        range_update(1, 0, n-1, l, r, val);
    }
};
```

62.5 Advanced Variations

- **Range Set Updates (with Lazy Propagation)**

```
void range_set(int node, int node_start, int node_end, int l, int r, int val) {
    push(node, node_start, node_end);
    if (r < node_start || l > node_end) return;
    if (l <= node_start && node_end <= r) {
        lazy_set[node] = val;
        lazy_add[node] = 0; // Reset add if set comes after
        push(node, node_start, node_end);
        return;
    }
    // ... similar to range_add implementation
}
```

- **Persistent Segment Trees**: Maintain multiple versions of the segment tree
- **2D Segment Trees**: For matrix/two-dimensional range queries
- **Segment Trees with Other Operations**:
  - Range minimum/maximum queries
  - Range GCD/LCM
  - Range bitwise operations (AND, OR, XOR)

62.6 When to Use Segment Trees

- Need to answer range queries and perform updates efficiently
- Problem involves range sum/min/max/GCD/XOR etc.
- Need to perform range updates (use lazy propagation)

- Need to answer historical queries (persistent segment trees)

62.7 Practice Problems

1. **Basic RSQ:** Range Sum Queries I (CSES)
2. **Range Updates:** Range Sum Queries II (CSES)
3. **RMQ:** Range Minimum Queries (CSES)
4. **Advanced:** Hotel Queries (CSES), Distinct Values Queries (CSES)
5. **Lazy Propagation:** Range Update and Sum (Codeforces)
6. **Persistent:** K-th Number (SPOJ), Distinct Numbers in Range (Codeforces)

62.8 Optimization Tips

- Iterative implementation: often faster than recursive
- Memory optimization: use exact power-of-two sizes when possible
- Bitwise operations: use $(n \gg 1)$ instead of $(n/2)$
- Coordinate compression: for large value ranges
- Disable bounds checking: in performance-critical code

Mastering segment trees and lazy propagation will give you powerful tools to solve a wide range of problems in competitive programming efficiently.

63 Persistent Data Structures: Advanced Techniques for Competitive Programming

Persistent data structures maintain multiple versions of themselves as they undergo modifications, enabling efficient access to historical states. This powerful concept is invaluable for solving problems that require querying past states of data.

63.1 Fundamental Concepts

Types of Persistence

1. **Partially Persistent**
    - All versions are linearly ordered (version 1, 2, 3...)
    - Can query any previous version but only modify the latest version
2. **Fully Persistent**
    - Forms a version tree (branches possible)
    - Can query and modify any version, creating new branches
3. **Confluently Persistent**
    - Allows merging of different versions
    - Most complex and rarely used in competitive programming

Key Properties

- **Immutability**: Previous versions remain unchanged
- **Path Copying**: Only copy modified nodes when updating
- **O(1) version access**: Each version has its own root
- **Time and space complexity**: Typically logarithmic overhead

63.2 Persistent Segment Tree

One of the most useful persistent structures for competitive programming, enabling efficient range queries across multiple versions.

Implementation (Point Updates)

```
struct Node {
    int value;
    Node *left, *right;
    Node(int v) : value(v), left(nullptr), right(nullptr) {}
    Node(Node *l, Node *r) : left(l), right(r) {
        value = (l ? l->value : 0) + (r ? r->value : 0);
    }
};

class PersistentSegTree {
    vector<Node*> versions;
    int n;

    Node* build(const vector<int>& arr, int l, int r) {
        if (l == r) return new Node(arr[l]);
        int mid = (l + r) / 2;
        return new Node(build(arr, l, mid), build(arr, mid+1, r));
    }

    Node* update(Node* prev, int l, int r, int idx, int val) {
        if (l == r) return new Node(val);
        int mid = (l + r) / 2;
        if (idx <= mid)
            return new Node(update(prev->left, l, mid, idx, val), prev->right);
        else
            return new Node(prev->left, update(prev->right, mid+1, r, idx, val));
    }

    int query(Node* node, int node_l, int node_r, int l, int r) {
        if (!node || r < node_l || l > node_r) return 0;
        if (l <= node_l && node_r <= r) return node->value;
        int mid = (node_l + node_r) / 2;
        return query(node->left, node_l, mid, l, r) +
               query(node->right, mid+1, node_r, l, r);
    }

public:
    PersistentSegTree(const vector<int>& arr) {
        n = arr.size();
        versions.push_back(build(arr, 0, n-1));
    }

    void update(int version, int idx, int val) {
        versions.push_back(update(versions[version], 0, n-1, idx, val));
    }

    int query(int version, int l, int r) {
        return query(versions[version], 0, n-1, l, r);
    }

    int latest_version() {
        return versions.size() - 1;
    }
};
```

63.3 Persistent Union-Find (Disjoint Set Union)

```
struct PersistentDSU {
    struct Node {
        int parent, rank;
        Node(int p = 0, int r = 0) : parent(p), rank(r) {}
    };

    vector<vector<Node>> versions;
    int n;

    PersistentDSU(int size) : n(size) {
        versions.emplace_back(n);
        for (int i = 0; i < n; i++)
            versions[0][i] = Node(i, 1);
    }

    int find(int version, int x) {
        if (versions[version][x].parent != x)
            return find(version, versions[version][x].parent);
        return x;
    }

    bool unite(int version, int x, int y) {
        x = find(version, x);
        y = find(version, y);
        if (x == y) {
            versions.push_back(versions[version]);
            return false;
        }

        versions.emplace_back(versions[version]);
        int new_version = versions.size() - 1;
        auto& node_x = versions[new_version][x];
        auto& node_y = versions[new_version][y];

        if (node_x.rank < node_y.rank) {
            node_x.parent = y;
        } else {
            node_y.parent = x;
            if (node_x.rank == node_y.rank)
                node_x.rank++;
        }
        return true;
    }

    bool same(int version, int x, int y) {
        return find(version, x) == find(version, y);
    }
};
```

63.4 Applications in Competitive Programming

- Historical Queries
- Functional Programming
- Temporal Data
- Parallel Processing

63.5 Common Problem Patterns

1. K-th Number in Range (SPOJ MKTHNUM)
2. Distinct Numbers in Range (Codeforces)
3. Version-Controlled Arrays
4. Time-Traveling DSU

63.6 Optimization Techniques

- Fat Nodes
- Path Copying
- Compression
- Lazy Propagation

63.7 Practice Problems

- **Easy**
  - MKTHNUM (SPOJ)
  - DQUERY (SPOJ)
- **Medium**
  - Chef and Array (CodeChef)
  - Travel in History (Codeforces)
- **Hard**
  - Version Controlled IDE (Codeforces)
  - Time Machine (ICPC World Finals)

63.8 Implementation Considerations

- Memory Management
- Version Indexing
- Garbage Collection
- Thread Safety

Mastering persistent data structures will give you powerful tools to solve complex problems involving temporal data, version control, and historical queries efficiently in competitive programming.

64 Grundy Numbers & Nim Game in C++: Game Theory for Competitive Programming

Here's a comprehensive C++ implementation of game theory concepts with Nim Game and Grundy Numbers:

64.1 Standard Nim Game Implementation

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

using namespace std;

bool isWinningPosition(const vector<int>& piles) {
    int nimSum = accumulate(piles.begin(), piles.end(), 0, bit_xor<int>());
    return nimSum != 0;
}

int main() {
    vector<int> piles = {3, 4, 5};
    if (isWinningPosition(piles)) {
        cout << "First player will win with optimal play\n";
    } else {
        cout << "Second player will win with optimal play\n";
    }
    return 0;
}
```

64.2 Grundy Number Calculation with Memoization

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

using namespace std;

vector<int> possibleMoves(int n) {
    vector<int> moves;
    for (int i = 1; i <= 3 && i <= n; i++) {
        moves.push_back(n - i);
    }
    return moves;
}

int calculateGrundy(int n, unordered_map<int, int>& memo) {
    if (n == 0) return 0;
    if (memo.count(n)) return memo[n];

    unordered_set<int> reachable;
    for (int move : possibleMoves(n)) {
        reachable.insert(calculateGrundy(move, memo));
    }

    int mex = 0;
    while (reachable.count(mex)) mex++;

    return memo[n] = mex;
}

int main() {
    unordered_map<int, int> memo;
    for (int n = 0; n <= 10; n++) {
        cout << "Grundy(" << n << ") = " << calculateGrundy(n, memo) << endl;
    }
    return 0;
}
```

64.3 Take-away Game Solution

```cpp
#include <iostream>

using namespace std;

string takeAwayWinner(int n) {
    return (n % 4 != 0) ? "First" : "Second";
}

int main() {
    for (int n = 0; n <= 10; n++) {
        cout << "With " << n << " stones: " << takeAwayWinner(n) << " player wins\n";
    }
    return 0;
}
```

64.4 Composite Game Example

```cpp
#include <iostream>
#include <vector>
#include <numeric>
#include <functional>
#include <unordered_map>
#include <unordered_set>

using namespace std;

vector<int> possibleMoves(int n) {
    vector<int> moves;
    for (int i = 1; i <= 3 && i <= n; i++) {
        moves.push_back(n - i);
    }
    return moves;
}

int calculateGrundy(int n, unordered_map<int, int>& memo) {
    if (n == 0) return 0;
    if (memo.count(n)) return memo[n];

    unordered_set<int> reachable;
    for (int move : possibleMoves(n)) {
        reachable.insert(calculateGrundy(move, memo));
    }

    int mex = 0;
    while (reachable.count(mex)) mex++;

    return memo[n] = mex;
}

int main() {
    vector<int> piles = {3, 4, 5};
    unordered_map<int, int> memo;
    vector<int> grundyNumbers;

    for (int pile : piles) {
        grundyNumbers.push_back(calculateGrundy(pile, memo));
    }

    int totalXOR = accumulate(grundyNumbers.begin(), grundyNumbers.end(), 0, bit_xor<int>());

    if (totalXOR != 0) {
        cout << "First player can win the composite game\n";
    } else {
        cout << "Second player can win the composite game\n";
    }
    return 0;
}
```

64.5 Misère Nim Variant

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

using namespace std;

string misereNimWinner(const vector<int>& piles) {
    if (all_of(piles.begin(), piles.end(), [](int x) { return x == 1; })) {
        return (piles.size() % 2 == 1) ? "Second" : "First";
    }

    int nimSum = accumulate(piles.begin(), piles.end(), 0, bit_xor<int>());
    return (nimSum != 0) ? "First" : "Second";
}

int main() {
    vector<int> piles = {1, 1, 1};
    cout << "Misère Nim winner: " << misereNimWinner(piles) << endl;
    return 0;
}
```

64.6 Wythoff's Game Implementation

```cpp
#include <iostream>
#include <cmath>
#include <utility>

using namespace std;

bool isWinningPosition(int a, int b) {
    if (a > b) swap(a, b);
    const double phi = (1 + sqrt(5)) / 2;
    int n = floor((b - a) * phi);
    return !(a == floor(n * phi) && b == a + n);
}

int main() {
    int a = 3, b = 5;
    if (isWinningPosition(a, b)) {
        cout << "Current player can force a win\n";
    } else {
        cout << "Opponent can force a win\n";
    }
    return 0;
}
```

64.7 Key Features of These Implementations

- **Standard Template Library Usage**: Leveraging `vector`, `unordered_map`, and algorithms.
- **Bitwise Operations**: Using `bit_xor` for efficient Nim-sum calculation.
- **Memoization**: Storing computed Grundy numbers for efficiency.
- **Mathematical Patterns**: Implementing game-specific optimizations.
- **Clear Output**: Demonstrating results with explanatory messages.

These implementations cover the fundamental game theory concepts you'll encounter in competitive programming, with efficient C++ solutions that can handle large inputs typical in programming contests.

65 Sprague-Grundy Theorem in C++: Advanced Game Theory for Competitive Programming

The Sprague-Grundy Theorem is a fundamental result in combinatorial game theory that generalizes the strategy for Nim to all impartial games. Here's a comprehensive C++ implementation covering the theorem and its applications.

65.1 Core Theorem Implementation

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

using namespace std;

// Function to calculate mex (minimum excludant)
int calculateMex(unordered_set<int>& s) {
    int mex = 0;
    while (s.find(mex) != s.end()) mex++;
    return mex;
}

// Function to compute Grundy number for a given position
int calculateGrundy(int n, vector<int>& dp) {
    if (dp[n] != -1) return dp[n];

    unordered_set<int> reachable;
    // Example moves: can take 1, 2, or 3 stones
    for (int i = 1; i <= 3; i++) {
        if (n - i >= 0) {
            reachable.insert(calculateGrundy(n - i, dp));
        }
    }

    dp[n] = calculateMex(reachable);
    return dp[n];
}

// Sprague-Grundy theorem application
bool isWinningPosition(vector<int>& positions) {
    vector<int> dp(*max_element(positions.begin(), positions.end()) + 1, -1);
    dp[0] = 0; // Base case

    int totalXOR = 0;
    for (int pos : positions) {
        totalXOR ^= calculateGrundy(pos, dp);
    }

    return totalXOR != 0;
}

int main() {
    vector<int> gamePositions = {3, 4, 5};
    if (isWinningPosition(gamePositions)) {
        cout << "First player can force a win\n";
    } else {
        cout << "Second player can force a win\n";
    }
    return 0;
}
```

65.2 Advanced Game Analysis with Multiple Move Options

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

```
#include <algorithm>

using namespace std;

class GameAnalyzer {
    vector<int> dp;
    vector<int> allowedMoves;

public:
    GameAnalyzer(int maxPosition, const vector<int>& moves)
        : dp(maxPosition + 1, -1), allowedMoves(moves) {
        dp[0] = 0; // Base case
    }

    int grundyNumber(int position) {
        if (dp[position] != -1) return dp[position];

        unordered_set<int> reachable;
        for (int move : allowedMoves) {
            if (position - move >= 0) {
                reachable.insert(grundyNumber(position - move));
            }
        }

        dp[position] = calculateMex(reachable);
        return dp[position];
    }

    bool isWinningState(const vector<int>& positions) {
        int totalXOR = 0;
        for (int pos : positions) {
            totalXOR ^= grundyNumber(pos);
        }
        return totalXOR != 0;
    }

private:
    int calculateMex(unordered_set<int>& s) {
        int mex = 0;
        while (s.find(mex) != s.end()) mex++;
        return mex;
    }
};

int main() {
    vector<int> allowedMoves = {1, 3, 4}; // Can take 1, 3, or 4 stones
    vector<int> gamePositions = {5, 7, 9};

    GameAnalyzer analyzer(10, allowedMoves); // Max position 10

    cout << "Grundy numbers:\n";
    for (int i = 0; i <= 10; i++) {
        cout << "G(" << i << ") = " << analyzer.grundyNumber(i) << endl;
    }

    if (analyzer.isWinningState(gamePositions)) {
        cout << "Current player can win\n";
    } else {
        cout << "Current player will lose against optimal play\n";
    }

    return 0;
}
```

65.3 Graph Game Implementation (for games represented as DAGs)

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

```
using namespace std;

class GraphGame {
    vector<vector<int>> adj;
    vector<int> grundy;
    vector<bool> visited;

    void dfs(int u) {
        visited[u] = true;
        unordered_set<int> reachable;

        for (int v : adj[u]) {
            if (!visited[v]) dfs(v);
            reachable.insert(grundy[v]);
        }

        grundy[u] = calculateMex(reachable);
    }

public:
    GraphGame(int n) : adj(n), grundy(n, 0), visited(n, false) {}

    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    void computeGrundyNumbers() {
        for (int i = 0; i < adj.size(); i++) {
            if (!visited[i]) dfs(i);
        }
    }

    int getGrundy(int u) { return grundy[u]; }

    bool isWinningPosition(int startNode) {
        return grundy[startNode] != 0;
    }

private:
    int calculateMex(unordered_set<int>& s) {
        int mex = 0;
        while (s.find(mex) != s.end()) mex++;
        return mex;
    }
};

int main() {
    // Example: Game with 5 positions
    GraphGame game(5);

    // Define game moves (directed edges)
    game.addEdge(0, 1); // From position 0 can move to 1
    game.addEdge(1, 2);
    game.addEdge(1, 3);
    game.addEdge(2, 4);
    game.addEdge(3, 4);
    game.addEdge(4, 0); // Looping move

    game.computeGrundyNumbers();

    cout << "Grundy numbers for each position:\n";
    for (int i = 0; i < 5; i++) {
        cout << "Position " << i << ": " << game.getGrundy(i) << endl;
    }

    int startPosition = 0;
    if (game.isWinningPosition(startPosition)) {
        cout << "Starting from position " << startPosition
             << ", first player can win\n";
    } else {
        cout << "Starting from position " << startPosition
             << ", first player will lose against optimal play\n";
    }
    return 0;
}
```

65.4 Multi-Dimensional Game Analysis

```cpp
#include <iostream>
#include <vector>
#include <unordered_set>
#include <tuple>
#include <map>

using namespace std;

class MultiDimensionalGame {
    map<tuple<int, int>, int> grundy;
    vector<pair<int, int>> allowedMoves;

    int calculateMex(unordered_set<int>& s) {
        int mex = 0;
        while (s.find(mex) != s.end()) mex++;
        return mex;
    }

public:
    MultiDimensionalGame(const vector<pair<int, int>>& moves)
        : allowedMoves(moves) {}

    int getGrundy(int x, int y) {
        auto key = make_tuple(x, y);
        if (grundy.count(key)) return grundy[key];

        if (x == 0 && y == 0) return grundy[key] = 0; // Terminal

        unordered_set<int> reachable;
        for (auto& move : allowedMoves) {
            int nx = x - move.first;
            int ny = y - move.second;
            if (nx >= 0 && ny >= 0) {
                reachable.insert(getGrundy(nx, ny));
            }
        }

        return grundy[key] = calculateMex(reachable);
    }

    bool isWinningPosition(int x, int y) {
        return getGrundy(x, y) != 0;
    }
};

int main() {
    // Knight's move game example
    vector<pair<int, int>> moves = {
        {1, 2}, {2, 1},    // Standard knight moves
        {-1, 2}, {2, -1},  // All 8 possible directions
        {1, -2}, {-2, 1},
        {-1, -2}, {-2, -1}
    };

    MultiDimensionalGame game(moves);

    int x = 4, y = 4;
    cout << "Grundy number for (" << x << "," << y << "): "
         << game.getGrundy(x, y) << endl;

    if (game.isWinningPosition(x, y)) {
        cout << "First player can win from this position\n";
    } else {
        cout << "Second player can force a win from this position\n";
    }
    // Print grundy table for small coordinates
    cout << "\nGrundy table for x,y up to 4:\n";
    for (int i = 0; i <= 4; i++) {
        for (int j = 0; j <= 4; j++) {
            cout << game.getGrundy(i, j) << " ";
        }
        cout << endl;
    }

    return 0;
}
```

65.5 Key Features of These Implementations

- **Comprehensive Coverage**: From basic to advanced game scenarios
- **Efficient Computation**: Using memoization and DP to avoid recomputation
- **Flexible Design**: Adaptable to various game rules through move definitions
- **Mathematically Sound**: Proper implementation of mex function and XOR principle
- **Real-world Applicable**: Ready-to-use code for programming competitions

These implementations demonstrate how to apply the Sprague-Grundy theorem to analyze impartial games, compute Grundy numbers, and determine winning strategies in C++. The code is optimized for competitive programming with clear structure and efficient algorithms.

66 Minimax with Alpha-Beta Pruning in C++

Here's a complete C++ implementation of the **Minimax algorithm with Alpha-Beta Pruning** for a simple **Tic-Tac-Toe** game. The AI will play optimally, ensuring it never loses.

66.1 Game Board Representation

We represent the Tic-Tac-Toe board as a 3×3 grid (`vector<vector<char>>`), where:

- `'X'` is the maximizing player (AI).
- `'O'` is the minimizing player (human).
- `'.'` denotes an empty cell.

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

using namespace std;

// Game board
vector<vector<char>> board(3, vector<char>(3, '.'));

// Print the board
void printBoard() {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}
```

66.2 Check for Win or Draw

We need helper functions to check if the game is over:

- `isWinner()` checks if a player has won.
- `isBoardFull()` checks if the game is a draw.

```cpp
// Check if a player has won
bool isWinner(char player) {
    // Check rows and columns
    for (int i = 0; i < 3; i++) {
        if (board[i][0] == player && board[i][1] == player && board[i][2] == player) return true;
        if (board[0][i] == player && board[1][i] == player && board[2][i] == player) return true;
    }
    // Check diagonals
    if (board[0][0] == player && board[1][1] == player && board[2][2] == player) return true;
    if (board[0][2] == player && board[1][1] == player && board[2][0] == player) return true;
    return false;
}

// Check if the board is full (draw)
bool isBoardFull() {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i][j] == '.') return false;
        }
    }
    return true;
}
```

66.3 Minimax with Alpha-Beta Pruning

The core of the AI logic. It recursively evaluates all possible moves and prunes irrelevant branches.

```cpp
// Minimax with Alpha-Beta Pruning
int minimax(int depth, bool isMax, int alpha, int beta) {
    // Terminal conditions
    if (isWinner('X')) return 10 - depth;  // AI wins
    if (isWinner('O')) return depth - 10;  // Human wins
    if (isBoardFull()) return 0;           // Draw

    if (isMax) {  // AI's turn (maximize)
        int bestScore = -numeric_limits<int>::max();
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == '.') {
                    board[i][j] = 'X';
                    int score = minimax(depth + 1, false, alpha, beta);
                    board[i][j] = '.';
                    bestScore = max(bestScore, score);
                    alpha = max(alpha, bestScore);
                    if (beta <= alpha) break;  // Alpha-Beta pruning
                }
            }
        }
        return bestScore;
    } else {  // Human's turn (minimize)
        int bestScore = numeric_limits<int>::max();
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == '.') {
                    board[i][j] = 'O';
                    int score = minimax(depth + 1, true, alpha, beta);
                    board[i][j] = '.';
                    bestScore = min(bestScore, score);
                    beta = min(beta, bestScore);
                    if (beta <= alpha) break;  // Alpha-Beta pruning
                }
            }
        }
        return bestScore;
    }
}
```

66.4 AI Move Selection

The AI selects the best move using `minimax()`.

```cpp
// AI makes the best move
void aiMove() {
    int bestScore = -numeric_limits<int>::max();
    int bestMoveRow = -1, bestMoveCol = -1;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i][j] == '.') {
                board[i][j] = 'X';
                int score = minimax(0, false, -numeric_limits<int>::max(), numeric_limits<int>::max());
                board[i][j] = '.';
                if (score > bestScore) {
                    bestScore = score;
                    bestMoveRow = i;
                    bestMoveCol = j;
                }
            }
        }
    }

    if (bestMoveRow != -1 && bestMoveCol != -1) {
        board[bestMoveRow][bestMoveCol] = 'X';
    }
}
```

66.5 Main Game Loop

The game alternates between the human ('O') and AI ('X') until someone wins or the game ends in a draw.

```
int main() {
    cout << "Tic-Tac-Toe (AI: X, Human: O)" << endl;

    while (true) {
        printBoard();

        // Human move
        int row, col;
        cout << "Enter row (0-2) and column (0-2): ";
        cin >> row >> col;

        if (row < 0 || row > 2 || col < 0 || col > 2 || board[row][col] != '.') {
            cout << "Invalid move! Try again." << endl;
            continue;
        }

        board[row][col] = 'O';

        if (isWinner('O')) {
            printBoard();
            cout << "You win!" << endl;
            break;
        }

        if (isBoardFull()) {
            printBoard();
            cout << "It's a draw!" << endl;
            break;
        }

        aiMove();

        if (isWinner('X')) {
            printBoard();
            cout << "AI wins!" << endl;
            break;
        }

        if (isBoardFull()) {
            printBoard();
            cout << "It's a draw!" << endl;
            break;
        }
    }

    return 0;
}
```

66.6 How It Works

1. The AI ('X') uses `minimax()` with **Alpha-Beta pruning** to evaluate all possible moves.
2. It assigns scores:
   - `+10` if AI wins.
   - `-10` if human wins.
   - `0` for a draw.
3. The AI selects the move with the **highest minimax score**.
4. The human ('O') inputs moves, and the game continues until a terminal state is reached.

66.7 Example Output

```
Tic-Tac-Toe (AI: X, Human: O)
. . .
. . .
. . .
Enter row (0-2) and column (0-2): 1 1
. . .
. O .
. . .
X . .
. O .
. . .
Enter row (0-2) and column (0-2): 0 2
X . O
. O .
. . .
X . O
. O .
X . .
Enter row (0-2) and column (0-2): 2 2
X . O
. O .
X . O
AI wins!
```

66.8 Key Optimizations

- **Alpha-Beta Pruning** reduces unnecessary computations.
- **Depth-based scoring** (`10 - depth`) ensures the AI wins in the fewest moves possible.

This implementation guarantees that the AI **never loses** (it either wins or forces a draw). You can extend this to more complex games like Chess or Checkers by modifying the evaluation function and move generation.

67 Convex Hull Algorithms: Graham Scan and Andrew's Algorithm

A **convex hull** is the smallest convex polygon that contains all given points in a plane. Two popular algorithms to compute it are:

- **Graham Scan** (1972) – Uses angular sorting and a stack.
- **Andrew's Algorithm (Monotone Chain)** – Sorts points by coordinates and builds upper/lower hulls.

67.1 Graham Scan Algorithm

Idea: Sort points by polar angle relative to the lowest point, then traverse while maintaining convexity using a stack.

Steps:

- Find the pivot point (lowest y-coordinate, leftmost if tie).
- Sort points by polar angle relative to the pivot.
- Iterate through sorted points, using a stack to discard non-convex points.

C++ Implementation

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>
using namespace std;

struct Point {
    int x, y;
};

Point pivot; // Lowest y-coordinate (leftmost if tie)

// Cross product to check orientation (ccw, collinear, cw)
int crossProduct(Point a, Point b, Point c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

// Compare polar angles relative to pivot
bool compare(Point a, Point b) {
    int orient = crossProduct(pivot, a, b);
    if (orient == 0) // Collinear, pick the closest one
        return (a.x - pivot.x) * (a.x - pivot.x) + (a.y - pivot.y) * (a.y - pivot.y)
            < (b.x - pivot.x) * (b.x - pivot.x) + (b.y - pivot.y) * (b.y - pivot.y);
    return orient > 0; // Counter-clockwise order
}

vector<Point> grahamScan(vector<Point>& points) {
    int n = points.size();
    if (n <= 3) return points; // Already convex if 3 points

    // Find the pivot (lowest y, leftmost if tie)
    pivot = points[0];
    for (int i = 1; i < n; i++) {
        if (points[i].y < pivot.y || (points[i].y == pivot.y && points[i].x < pivot.x))
            pivot = points[i];
    }

    // Sort points by polar angle
    sort(points.begin(), points.end(), compare);

    // Build convex hull using a stack
    vector<Point> hull;
    hull.push_back(points[0]);
    hull.push_back(points[1]);
    for (int i = 2; i < n; i++) {
        while (hull.size() >= 2 && crossProduct(hull[hull.size()-2], hull.back(), points[i]) <= 0) {
            hull.pop_back(); // Remove non-convex points
        }
        hull.push_back(points[i]);
    }

    return hull;
}

int main() {
    vector<Point> points = {{0, 3}, {1, 1}, {2, 2}, {4, 4}, {0, 0}, {1, 2}, {3, 1}, {3, 3}};
    vector<Point> convexHull = grahamScan(points);

    cout << "Convex Hull Points:\n";
    for (auto p : convexHull) {
        cout << "(" << p.x << ", " << p.y << ")\n";
    }

    return 0;
}
```

Output:

```
Convex Hull Points:
(0, 0)
(3, 1)
(4, 4)
(0, 3)
```

67.2 Andrew's Algorithm (Monotone Chain)

Idea: Sort points by x-coordinate (break ties by y), then construct upper and lower hulls.

Steps:

- Sort points lexicographically (by x, then y).
- Build lower hull (left to right, keeping right turns).
- Build upper hull (right to left, keeping right turns).
- Merge hulls (remove duplicate endpoints).

C++ Implementation

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

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

// Cross product (Orient: +ve=ccw, 0=collinear, -ve=cw)
int cross(Point a, Point b, Point c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

vector<Point> convexHull(vector<Point>& points) {
    int n = points.size();
    if (n <= 3) return points;

    sort(points.begin(), points.end());

    vector<Point> hull;
    hull.reserve(2 * n);

    // Lower hull (left to right)
    for (int i = 0; i < n; i++) {
        while (hull.size() >= 2 && cross(hull[hull.size()-2], hull.back(), points[i]) <= 0) {
            hull.pop_back();
        }
        hull.push_back(points[i]);
    }

    // Upper hull (right to left)
    for (int i = n-2, t = hull.size()+1; i >= 0; i--) {
        while (hull.size() >= t && cross(hull[hull.size()-2], hull.back(), points[i]) <= 0) {
            hull.pop_back();
        }
        hull.push_back(points[i]);
    }

    hull.pop_back(); // Remove duplicate start point
    return hull;
}

int main() {
    vector<Point> points = {{0, 3}, {1, 1}, {2, 2}, {4, 4}, {0, 0}, {1, 2}, {3, 1}, {3, 3}};
    vector<Point> hull = convexHull(points);

    cout << "Convex Hull Points:\n";
    for (auto p : hull) {
        cout << "(" << p.x << ", " << p.y << ")\n";
    }

    return 0;
}
```

Output:

```
Convex Hull Points:
(0, 0)
(3, 1)
(4, 4)
(0, 3)
```

67.3 Comparison

Algorithm	Time Complexity	Space Complexity	Key Idea
Graham Scan	$O(n \log n)$	$O(n)$	Sort by angle, stack-based
Andrew's	$O(n \log n)$	$O(n)$	Sort by x, upper/lower hulls

When to Use Which?

- Graham Scan is simpler but **sensitive to collinear points**.
- Andrew's Algorithm is more **numerically stable** and often preferred.

67.4 Applications

- Collision detection (smallest enclosing shape).
- Path planning (robot navigation).
- Computer graphics (shape simplification).

68 Line Intersection & Sweep Line Algorithm

The **Sweep Line Algorithm** is a computational geometry technique used to efficiently solve problems like:

- **Line segment intersection**
- **Closest pair of points**
- **Convex hull construction**

Here, we focus on **finding all intersection points** among a set of line segments in $O((n + k) \log n)$ time, where:

- n = number of segments,
- k = number of intersections.

68.1 Sweep Line Algorithm for Line Intersection

Key Idea

- A **vertical sweep line** moves from left to right.
- **Active segments** (those intersecting the sweep line) are stored in a **balanced BST** (e.g., `std::set` in C++).
- **Events** (segment endpoints and intersections) are processed in order.

Steps

- Sort all endpoints and possible intersections (event points) by x-coordinate.
- Process events:
  - **Left endpoint**: Add segment to active set and check for new intersections with neighbors.
  - **Right endpoint**: Remove segment from active set and check for intersections between its former neighbors.
  - **Intersection point**: Swap the two intersecting segments in the active set and check for new intersections.

68.2 C++ Implementation

Data Structures

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include <cmath>
using namespace std;

struct Point {
    double x, y;
    bool operator<(const Point& p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

struct Segment {
    Point p1, p2;
    int id;
};

// Event structure: (x-coordinate, isLeftEndpoint, segment)
struct Event {
    double x;
    bool isLeft;
    Segment seg;
    bool operator<(const Event& e) const {
        return x < e.x;
    }
};
```

Intersection Check

```
// Check if segments s1 and s2 intersect
bool intersect(const Segment& s1, const Segment& s2) {
    auto cross = [](Point a, Point b, Point c) {
        return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    };

    double d1 = cross(s1.p1, s1.p2, s2.p1);
    double d2 = cross(s1.p1, s1.p2, s2.p2);
    double d3 = cross(s2.p1, s2.p2, s1.p1);
    double d4 = cross(s2.p1, s2.p2, s1.p2);

    // Proper intersection
    if ((d1 * d2 < 0) && (d3 * d4 < 0)) return true;

    // Collinear cases (optional, if needed)
    return false;
}
```

Sweep Line Algorithm

```
vector<Point> findIntersections(vector<Segment>& segments) {
    vector<Event> events;
    for (auto& seg : segments) {
        if (seg.p2 < seg.p1) swap(seg.p1, seg.p2); // Ensure p1 is left of p2
        events.push_back({seg.p1.x, true, seg});
        events.push_back({seg.p2.x, false, seg});
    }
    sort(events.begin(), events.end());

    auto cmp = [](const Segment& a, const Segment& b) {
        if (a.p1.x == b.p1.x) return a.p2.x < b.p2.x;
        return a.p1.x < b.p1.x;
    };
    set<Segment, decltype(cmp)> activeSegments(cmp);

    vector<Point> intersections;

    for (auto& event : events) {
        if (event.isLeft) {
            auto it = activeSegments.insert(event.seg).first;
            auto prev = it, next = it;
            if (prev != activeSegments.begin()) {
                prev--;
                if (intersect(*prev, *it)) {
                    // Compute intersection point (omitted for brevity)
                    intersections.push_back({...});
                }
            }
            next++;
            if (next != activeSegments.end()) {
                if (intersect(*it, *next)) {
                    intersections.push_back({...});
                }
            }
        } else {
            auto it = activeSegments.find(event.seg);
            if (it == activeSegments.end()) continue;
            auto prev = it, next = it;
            if (prev != activeSegments.begin() && next != activeSegments.end()) {
                prev--;
                next++;
                if (intersect(*prev, *next)) {
                    intersections.push_back({...});
                }
            }
            activeSegments.erase(it);
        }
    }

    return intersections;
}
```

68.3 Time Complexity

Step	**Complexity**
Sorting events	$O(n \log n)$
Processing events	$O((n+k) \log n)$
Total	$O((n+k) \log n)$

- **Worst case** ($k = O(n^2)$): $O(n^2 \log n)$
- **Best case** ($k = 0$): $O(n \log n)$

68.4 Applications

- **Computer graphics** (clipping, hidden line removal)
- **VLSI design** (wire routing)
- **GIS systems** (map overlay)

68.5 Alternative: Bentley-Ottmann Algorithm

A more advanced version of the sweep line algorithm:

- Uses a **priority queue** for events.
- Handles **degenerate cases** (e.g., overlapping segments).
- Guarantees $O((n+k) \log n)$ time.

68.6 Example

Input:

```
Segment 1: (1,1) → (4,4)
Segment 2: (1,4) → (4,1)
Segment 3: (2,2) → (5,5)
```

Output:

```
Intersection at (2.5, 2.5)
Intersection at (3.0, 3.0)
```

68.7 Conclusion

The **Sweep Line Algorithm** efficiently finds line intersections by:

- Sorting events (left/right endpoints and intersections)
- Maintaining active segments in a BST
- Checking neighboring segments for new intersections

69 Closest Pair of Points (Divide and Conquer Algorithm)

The **Closest Pair of Points** problem involves finding the two points with the smallest Euclidean distance in a given set of n points. A naive approach checks all pairs ($O(n^2)$), but **Divide and Conquer** solves it in $O(n \log n)$.

69.1 Algorithm Overview

69.1.1 Key Steps

- Sort points by x-coordinate.
- Divide the set into two equal halves.
- Recursively find the closest pairs in left and right halves.
- Merge results by checking points near the split line (strip of width $2d$, where $d = \min(\text{left\_min}, \text{right\_min})$).

69.1.2 Time Complexity

Step	Complexity
Sorting	$O(n \log n)$
Divide & Conquer	$O(n \log n)$
Total	$O(n \log n)$

69.2 C++ Implementation

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <limits>
using namespace std;

struct Point {
    double x, y;
};

bool compareX(const Point& a, const Point& b) {
    return a.x < b.x;
}

bool compareY(const Point& a, const Point& b) {
    return a.y < b.y;
}

double distance(const Point& a, const Point& b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

double bruteForce(const vector<Point>& points, int left, int right) {
    double minDist = numeric_limits<double>::max();
    for (int i = left; i <= right; i++) {
        for (int j = i + 1; j <= right; j++) {
            minDist = min(minDist, distance(points[i], points[j]));
        }
    }
    return minDist;
}

double stripClosest(vector<Point>& strip, double d) {
    double minDist = d;
    sort(strip.begin(), strip.end(), compareY);
    for (int i = 0; i < strip.size(); i++) {
        for (int j = i + 1; j < strip.size() && (strip[j].y - strip[i].y) < minDist; j++) {
            minDist = min(minDist, distance(strip[i], strip[j]));
        }
    }
    return minDist;
}

double closestUtil(vector<Point>& points, int left, int right) {
    if (right - left <= 3) {
        return bruteForce(points, left, right);
    }

    int mid = (left + right) / 2;
    Point midPoint = points[mid];

    double dl = closestUtil(points, left, mid);
    double dr = closestUtil(points, mid + 1, right);
    double d = min(dl, dr);

    vector<Point> strip;
    for (int i = left; i <= right; i++) {
        if (abs(points[i].x - midPoint.x) < d) {
            strip.push_back(points[i]);
        }
    }

    return min(d, stripClosest(strip, d));
}

double closestPair(vector<Point>& points) {
    sort(points.begin(), points.end(), compareX);
    return closestUtil(points, 0, points.size() - 1);
}

int main() {
    vector<Point> points = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    double minDist = closestPair(points);
    cout << "The smallest distance is: " << minDist << endl;
    return 0;
}
```

Output:

```
The smallest distance is: 1.41421
```

69.3 Explanation

- Sorting: Points are sorted by x-coordinate.
- Divide: Split into left and right halves.
- Conquer: Recursively find the smallest distances d_L (left) and d_R (right).
- Merge: Check the strip (points within $d = \min(d_L, d_R)$ of the midline). Only 7 points per strip need checking due to geometric properties.

69.4 Why Only 7 Points in the Strip?

- **Pigeonhole Principle**: In a $d \times 2d$ rectangle, at most 8 points can exist without being closer than d.
- **Proof**: Divide the strip into 8 squares of side $d/2$. Only one point per square is allowed, ensuring $O(1)$ checks per point.

69.5 Applications

- Computational biology (DNA sequence alignment)
- Networking (closest server selection)
- Computer vision (feature matching)

69.6 Optimizations

- Avoid repeated sorting: Pre-sort by y-coordinate once.
- Parallelization: Divide steps can run in parallel.

69.7 Comparison with Naive Approach

Method	Time Complexity	Space Complexity
Brute Force	$O(n^2)$	$O(1)$
Divide & Conquer	$O(n \log n)$	$O(n)$

69.8 Extensions

- Higher dimensions (3D closest pair)
- Approximate algorithms (for very large datasets)

70 Two Pointers Technique: A Powerful Optimization Tool

The **Two Pointers Technique** is an algorithmic pattern that efficiently solves problems by using two pointers (indices) to traverse data structures (arrays, linked lists, strings) in a single pass. It's widely used to optimize nested loops from $O(n^2)$ to $O(n)$.

70.1 When to Use Two Pointers?

- ✓ **Sorted arrays/lists** (e.g., pair sum, triplets)
- ✓ **Sliding window problems** (e.g., subarrays with a given sum)
- ✓ **In-place modifications** (e.g., removing duplicates)
- ✓ **Palindrome checks** (e.g., string reversal)

70.2 Common Two-Pointer Patterns

70.2.1 (1) Opposite Direction (Converging Pointers)

- **Use Case**: Searching pairs in a **sorted array** (e.g., two-sum).
- **Pointer Movement**:
  - `left` starts at **beginning**, `right` at **end**.
  - Move inward based on conditions.

Example: Two Sum in Sorted Array

```
vector<int> twoSum(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) return {left, right};
        else if (sum < target) left++;  // Need larger sum
        else right--;                   // Need smaller sum
    }
    return {};
}
```

Time Complexity: $O(n)$

70.2.2 (2) Same Direction (Fast & Slow Pointers)

- **Use Case**:
  - Linked list cycle detection (Floyd's algorithm)
  - Removing duplicates in-place
- **Pointer Movement**:
  - `slow` processes data
  - `fast` explores ahead

Example: Remove Duplicates from Sorted Array

```
int removeDuplicates(vector<int>& nums) {
    if (nums.empty()) return 0;
    int slow = 0;
    for (int fast = 1; fast < nums.size(); fast++) {
        if (nums[fast] != nums[slow]) {
            slow++;
            nums[slow] = nums[fast];
        }
    }
    return slow + 1;
}
```

Time Complexity: $O(n)$

70.2.3 (3) Sliding Window (Subarray Problems)

- **Use Case**: Find subarrays meeting a condition (e.g., max sum, shortest/longest window).
- **Pointer Movement**:
  - `left` and `right` expand/shrink the window dynamically.

Example: Minimum Size Subarray Sum

```
int minSubArrayLen(int target, vector<int>& nums) {
    int left = 0, sum = 0, minLen = INT_MAX;
    for (int right = 0; right < nums.size(); right++) {
        sum += nums[right];
        while (sum >= target) {
            minLen = min(minLen, right - left + 1);
            sum -= nums[left++];  // Shrink window
        }
    }
    return minLen == INT_MAX ? 0 : minLen;
}
```

Time Complexity: $O(n)$

70.3 Key Problems Solved by Two Pointers

Problem	Brute Force	Two Pointers
Two Sum (sorted array)	$O(n^2)$	$O(n)$
Remove Duplicates	$O(n^2)$	$O(n)$
Container With Most Water	$O(n^2)$	$O(n)$
Longest Substring Without Repeats	$O(n^2)$	$O(n)$

70.4 Why Two Pointers?

- **Space Efficiency**: Often uses $O(1)$ extra space.
- **Time Efficiency**: Converts $O(n^2)$ to $O(n)$.
- **Readability**: Cleaner than nested loops.

70.5 Advanced Variations

1. **Three Pointers** (e.g., Dutch National Flag problem)
2. **Four Pointers** (e.g., merging two sorted arrays in-place)
3. **Non-adjacent Pointers** (e.g., trapping rainwater)

70.6 Practice Problems

1. Two Sum II
2. Remove Duplicates
3. Container With Most Water
4. Longest Substring Without Repeating Characters

70.7 7. C++ Template for Two Pointers

```cpp
int twoPointers(vector<int>& nums) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        if (condition) {
            // Process left and/or right
            left++;
        } else {
            right--;
        }
    }
    return result;
}
```

70.8 Conclusion

The **Two Pointers Technique** is a must-know for coding interviews. It optimizes problems by **eliminating redundant checks** and is especially powerful for **sorted data** and **sliding windows**.

71 Meet-in-the-Middle: A Powerful Technique for NP Problems

The **Meet-in-the-Middle (MITM)** technique is an optimization strategy that splits a problem into two halves, solves each independently, and then combines the results. It's particularly useful for **NP-hard problems** where brute force is infeasible.

71.1 When to Use Meet-in-the-Middle?

- ✓ Subset Sum (Find subsets with a target sum)
- ✓ Knapsack Problems (Optimize weight/value constraints)
- ✓ Closest Pair in High Dimensions
- ✓ Breaking Symmetric Encryption (e.g., DES brute-force attacks)

Why MITM?

- Reduces brute-force time from $O(2^n) \rightarrow O(2^{n/2})$.
- Works well when combining two halves is efficient (e.g., using hash maps or binary search).

71.2 Classic Example: Subset Sum Problem

Problem Statement: Given an array `nums` and a target `T`, does any subset sum to `T`?
Brute Force ($O(2^n)$): Check all possible subsets.
MITM Optimization ($O(2^{n/2})$):

1. Split the array into two halves A and B.
2. Generate all subset sums for A and B separately.
3. Check combinations:
   - If `sumA + sumB == T`, return `true`.
   - Use hashing or sorting + binary search for efficient lookup.

C++ Implementation:

```cpp
#include <vector>
#include <unordered_set>
using namespace std;

bool subsetSumMITM(vector<int>& nums, int target) {
    int n = nums.size();
    int half = n / 2;

    vector<int> sumA;
    for (int mask = 0; mask < (1 << half); mask++) {
        int sum = 0;
        for (int i = 0; i < half; i++) {
            if (mask & (1 << i)) sum += nums[i];
        }
        sumA.push_back(sum);
    }

    unordered_set<int> sumB;
    for (int mask = 0; mask < (1 << (n - half)); mask++) {
        int sum = 0;
        for (int i = 0; i < (n - half); i++) {
            if (mask & (1 << i)) sum += nums[half + i];
        }
        sumB.insert(sum);
    }

    for (int s : sumA) {
        if (sumB.count(target - s)) {
            return true;
        }
    }
    return false;
}
```

Time Complexity:

- Generating subset sums: $O(2^{n/2})$
- Checking combinations: $O(2^{n/2})$
- Total: $O(2^{n/2})$

71.3 MITM for 0-1 Knapsack Problem

Problem Statement: Given weights `w`, values `v`, and max capacity `W`, maximize value without exceeding `W`.

MITM Approach:

1. Split items into two halves.
2. Generate all possible weight-value pairs for each half.
3. Sort one half and use binary search to find best feasible combination.

C++ Snippet:

```cpp
vector<pair<int, int>> generatePairs(vector<int>& w, vector<int>& v, int l, int r) {
    vector<pair<int, int>> res;
    int n = r - l;
    for (int mask = 0; mask < (1 << n); mask++) {
        int weight = 0, value = 0;
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                weight += w[l + i];
                value += v[l + i];
            }
        }
        res.push_back({weight, value});
    }
    return res;
}

int knapsackMITM(vector<int>& w, vector<int>& v, int W) {
    int n = w.size();
    int half = n / 2;

    auto left = generatePairs(w, v, 0, half);
    auto right = generatePairs(w, v, half, n);

    sort(right.begin(), right.end());
    vector<int> maxValRight;
    int maxSoFar = 0;
    for (auto [weight, val] : right) {
        if (val > maxSoFar) maxSoFar = val;
        maxValRight.push_back(maxSoFar);
    }

    int maxValue = 0;
    for (auto [weightL, valL] : left) {
        if (weightL > W) continue;
        int remaining = W - weightL;

        auto it = upper_bound(right.begin(), right.end(), make_pair(remaining, INT_MAX));
        if (it != right.begin()) {
            int idx = it - right.begin() - 1;
            maxValue = max(maxValue, valL + maxValRight[idx]);
        }
    }
    return maxValue;
}
```

Time Complexity: $O(n \cdot 2^{n/2})$

71.4 MITM vs. Dynamic Programming (DP)

Aspect	**MITM**	**DP (Knapsack)**
Time Complexity	$O(n \cdot 2^{n/2})$	$O(nW)$ (pseudo-poly)
Space Complexity	$O(2^{n/2})$	$O(W)$
Best For	Medium-sized $n \approx$ 40-60	Small W

71.5 Advanced Applications

1. Cryptanalysis (e.g., breaking DES with MITM + time-memory trade-offs)
2. Closest Pair in High Dimensions (split dimensions)
3. Traveling Salesman Problem (TSP) (split cities into two sets)

71.6 Limitations

- Not suitable for very large n (still exponential)
- Requires problem to be splittable (not all NP problems work)

71.7 Key Takeaways

- ✓ MITM reduces $O(2^n) \to O(2^{n/2})$, making some problems feasible.
- ✓ Works best when combining halves is efficient (e.g., hashing, sorting).
- ✓ Ideal for medium-sized NP problems (n 40–60).

Want a deeper dive into MITM for TSP or cryptography? Let me know!

72 Randomized Algorithms & Heuristics in C++

This section covers practical implementations of Monte Carlo methods, Karger's min-cut, simulated annealing, and genetic algorithms.

72.1 Monte Carlo Estimation

Estimate by sampling points in the unit square and counting those inside the unit circle.

```cpp
double estimate_pi(int n) {
    std::mt19937 gen(std::random_device{}());
    std::uniform_real_distribution<> dis(0,1);
    int inside = 0;
    for(int i=0; i<n; ++i) {
        double x = dis(gen), y = dis(gen);
        if(x*x + y*y <= 1) inside++;
    }
    return 4.0 * inside / n;
}
```

72.2 Karger's Min-Cut Algorithm

Randomly contracts edges until two nodes remain, approximating the minimum cut.

```cpp
using Graph = std::unordered_map<int, std::vector<int>>;

void contract(Graph& g, int u, int v) {
    g[u].insert(g[u].end(), g[v].begin(), g[v].end());
    for(int n : g[v]) {
        auto& edges = g[n];
        edges.erase(std::remove(edges.begin(), edges.end(), v), edges.end());
        edges.push_back(u);
    }
    g.erase(v);
}

int karger_min_cut(Graph g) {
    std::mt19937 gen(std::random_device{}());
    while(g.size() > 2) {
        auto it = g.begin();
        std::advance(it, std::uniform_int_distribution<>(0, g.size()-1)(gen));
        int u = it->first;
        auto& edges = it->second;
        int v = edges[std::uniform_int_distribution<>(0, edges.size()-1)(gen)];
        contract(g, u, v);
    }
    return g.begin()->second.size();
}
```

72.3 Simulated Annealing (TSP)

Improves a route by swapping cities; occasionally accepts worse routes to escape local minima.

```cpp
double path_length(const auto& cities, const std::vector<int>& path) {
    double dist = 0;
    for(size_t i=0; i<path.size(); ++i) {
        int j = (i+1) % path.size();
        auto [x1,y1] = cities[path[i]];
        auto [x2,y2] = cities[path[j]];
        dist += std::hypot(x2-x1, y2-y1);
    }
    return dist;
}

std::vector<int> simulated_annealing_tsp(const auto& cities, double T=10000, double cooling=0.99) {
    std::mt19937 gen(std::random_device{}());
    std::uniform_int_distribution<> idx_dist(0, cities.size()-1);
    std::uniform_real_distribution<> prob(0,1);

    std::vector<int> path(cities.size());
    std::iota(path.begin(), path.end(), 0);
    std::shuffle(path.begin(), path.end(), gen);

    auto best = path;
    double best_dist = path_length(cities, path);

    while(T > 1e-3) {
        auto neighbor = path;
        std::swap(neighbor[idx_dist(gen)], neighbor[idx_dist(gen)]);
        double curr_dist = path_length(cities, path);
        double neigh_dist = path_length(cities, neighbor);
        double delta = neigh_dist - curr_dist;

        if(delta < 0 || prob(gen) < std::exp(-delta / T)) {
            path = neighbor;
            if(neigh_dist < best_dist) {
                best = neighbor;
                best_dist = neigh_dist;
            }
        }
        T *= cooling;
    }
    return best;
}
```

72.4 Genetic Algorithm (Knapsack)

Evolves a population of solutions with selection, crossover, and mutation.

```
struct Item { int weight, value; };

int evaluate(const std::vector<bool>& sol, const std::vector<Item>& items, int cap) {
    int val=0, wt=0;
    for(size_t i=0; i<sol.size(); ++i)
        if(sol[i]) { val+=items[i].value; wt+=items[i].weight; }
    return (wt <= cap) ? val : 0;
}

std::vector<bool> crossover(const std::vector<bool>& p1, const std::vector<bool>& p2) {
    std::mt19937 gen(std::random_device{}());
    int cp = std::uniform_int_distribution<>(0, p1.size()-1)(gen);
    std::vector<bool> child(p1.size());
    std::copy(p1.begin(), p1.begin()+cp, child.begin());
    std::copy(p2.begin()+cp, p2.end(), child.begin()+cp);
    return child;
}

void mutate(std::vector<bool>& sol, double rate=0.01) {
    std::mt19937 gen(std::random_device{}());
    std::uniform_real_distribution<> prob(0,1);
    for(auto& bit : sol)
        if(prob(gen) < rate) bit = !bit;
}
```

72.5 Summary

- Monte Carlo: Random sampling for approximation.
- Karger's: Probabilistic graph min-cut.
- Simulated Annealing: Escape local optima in optimization.
- Genetic Algorithm: Evolutionary combinatorial search.

Balance speed and accuracy for hard problems.