

Competitive Programming Reference

TryOmar's Algorithm Collection

A comprehensive collection of algorithms, data structures, and templates

August 5, 2025

GENERATED FROM MARKDOWN REFERENCE

Contents

1	Introduction	3
1.1	How to Use This Reference	3
2	Data Structures	4
2.1	STL Basics	4
2.1.1	Important STL Concepts	4
2.1.2	Common STL Operations	4
2.1.3	Performance Considerations	4
2.1.4	Memory Management	4
2.1.5	Vectors and Arrays	5
2.1.6	Sets and Maps	6
2.1.7	Priority Queue and Heaps	7
2.1.8	Stack and Queue	7
2.1.9	Bitset	8
2.1.10	Bit Manipulation	9
2.1.11	Ordered Set Template	12
2.2	Advanced Data Structures	13
2.2.1	Segment Tree (Iterative)	13
2.2.2	Disjoint Set Union (DSU)	15
3	Graph Algorithms	17
3.1	Depth-First Search (DFS)	17
3.2	Breadth-First Search (BFS)	19
3.3	Dijkstra's Algorithm	21
3.4	Floyd-Warshall Algorithm	23
3.5	Topological Sort	24
3.6	Cycle Detection	26
4	Dynamic Programming	28
4.1	Longest Increasing Subsequence (LIS)	28
5	Backtracking	33
5.1	Subsets	33
5.2	Permutations	34
5.3	Combinations	36
6	String Algorithms	37
6.1	C++ STL String Functions	37
6.2	Longest Substring Without Repeating Characters	38
6.3	Trie (Prefix Tree)	39
7	Mathematics	40
7.1	Fast Power (Binary Exponentiation)	40
7.2	GCD and LCM Functions	41
7.3	Combinatorics	42
7.4	Sieve of Eratosthenes	44

8	Notes & Utilities	45
8.1	Binary Conversions	45
8.2	Coordinate Compression	46
8.3	Performance Utilities	47
8.4	Random Number Generation	48
8.5	String Utilities	49
8.6	Custom Comparators	50
9	Searching Algorithms	51
9.1	Binary Search	51
9.2	Lower Bound / Upper Bound	51
9.3	Binary Search on Answer	53
9.4	Ternary Search	54
10	Geometry (CP Basics)	55
10.1	Points & Vectors	55
10.2	Lines & Segments	56
10.3	Polygons & Areas	57
10.4	Circles and Advanced Geometry	60
10.5	3D Geometry	62

1 Introduction

This document contains a comprehensive collection of algorithms, data structures, and templates for competitive programming. Each section includes implementation details, time complexity analysis, and usage examples. This reference is open source and available on GitHub: <https://github.com/TryOmar/TryOmar-CP-Reference/>. Feel free to contribute improvements or report issues.

1.1 How to Use This Reference

This competitive programming reference is designed to be your go-to resource during contests and practice sessions.

Quick Navigation

- **Table of Contents:** Jump directly to any section
- **Code Blocks:** Ready-to-use implementations
- **Notes:** Implementation details after code blocks
- **Examples:** Practical usage examples

During Contests

- **Copy-Paste Ready:** All code blocks are tested and ready for immediate use
- **Template Approach:** Use provided templates as starting points
- **Time Complexity:** Check notes for complexity analysis
- **Edge Cases:** Notes include important edge cases

Sections Overview

- **Data Structures:** STL containers, advanced structures, and custom implementations
- **Graph Algorithms:** Traversal, shortest paths, and graph analysis
- **Dynamic Programming:** Classic problems and optimization techniques
- **Backtracking:** Combinatorial algorithms and constraint satisfaction
- **String Algorithms:** Pattern matching and string processing
- **Mathematics:** Number theory, combinatorics, and mathematical tools
- **Searching Algorithms:** Binary search variants and search patterns
- **Geometry:** Computational geometry algorithms and formulas
- **Notes & Utilities:** Helper functions and implementation tricks

2 Data Structures

2.1 STL Basics

This section covers the essential C++ Standard Template Library (STL) data structures commonly used in competitive programming.

2.1.1 Important STL Concepts

- **Containers:** Data structures that hold objects (vector, set, map, etc.)
- **Iterators:** Objects that point to elements in containers
- **Algorithms:** Functions that operate on containers (sort, find, etc.)
- **Function Objects:** Objects that can be called like functions
- **Allocators:** Manage memory allocation for containers

2.1.2 Common STL Operations

- **Insertion:** `insert()`, `push_back()`, `emplace()`
- **Deletion:** `erase()`, `pop_back()`, `clear()`
- **Access:** `at()`, `operator[]`, `front()`, `back()`
- **Size:** `size()`, `empty()`, `capacity()`
- **Iteration:** Range-based for loops, iterators, `begin()`, `end()`

2.1.3 Performance Considerations

- **Vector:** $O(1)$ amortized insertion at end, $O(n)$ insertion in middle
- **Set/Map:** $O(\log n)$ for insert, delete, search (Red-Black tree)
- **Unordered Set/Map:** $O(1)$ average case, $O(n)$ worst case (hash table)
- **Stack/Queue:** $O(1)$ for push/pop operations
- **Priority Queue:** $O(\log n)$ for push/pop operations

2.1.4 Memory Management

- **Vector:** Automatically grows, use `reserve()` to pre-allocate
- **Set/Map:** Memory allocated per node, efficient for sparse data
- **Unordered:** Memory allocated in buckets, good for dense data
- **Stack/Queue:** Memory allocated as needed, efficient for LIFO/FIFO

2.1.5 Vectors and Arrays

1: Basic Vector Operations

```
1 // Vector initialization
2 vector<int> v; // Empty vector
3 vector<int> v(5); // Size 5, initialized with 0s
4 vector<int> v(5, 2); // Size 5, initialized with 2s
5 vector<int> v = {1, 2, 3}; // Direct initialization
6
7 // Basic operations
8 v.push_back(4); // Add element to end
9 v.pop_back(); // Remove last element
10 v.size(); // Get current size
11 v.empty(); // Check if empty
12 v.front(); // First element
13 v.back(); // Last element
14 v.clear(); // Remove all elements
15
16 // Access and iteration
17 for(int i = 0; i < v.size(); i++) {
18     cout << v[i] << " "; // Using index
19 }
20 for(int x : v) { // Range-based for loop
21     cout << x << " ";
22 }
```

2: 2D Vector Operations

```
1 // 2D vector initialization
2 vector<vector<int>> grid(n, vector<int>(m)); // n x m grid
3 vector<vector<int>> grid = { // Direct init
4     {1, 2, 3},
5     {4, 5, 6},
6     {7, 8, 9}
7 };
8
9 // Access elements
10 grid[i][j] = value; // Set value
11 int value = grid[i][j]; // Get value
12
13 // Common operations
14 for(int i = 0; i < grid.size(); i++) {
15     for(int j = 0; j < grid[i].size(); j++) {
16         cout << grid[i][j] << " ";
17     }
18     cout << "\n";
19 }
```

2.1.6 Sets and Maps

3: Set and Unordered Set

```
1 // Set (ordered)
2 set<int> s; // Ordered unique elements
3 s.insert(5); // O(log n) insertion
4 s.erase(5); // O(log n) deletion
5 auto it = s.find(5); // O(log n) search
6 auto it = s.lower_bound(5); // First element >= 5
7 auto it = s.upper_bound(5); // First element > 5
8
9 // Unordered Set (hash table)
10 unordered_set<int> us; // Unordered unique elements
11 us.insert(5); // O(1) average case
12 us.erase(5); // O(1) average case
13 auto it = us.find(5); // O(1) average case
```

4: Map and Unordered Map

```
1 // Map (ordered)
2 map<string, int> m; // Key-value pairs
3 m["apple"] = 5; // O(log n) insertion
4 m.erase("apple"); // O(log n) deletion
5 auto it = m.find("apple"); // O(log n) search
6
7 // Unordered Map (hash table)
8 unordered_map<string, int> um; // Hash table
9 um["apple"] = 5; // O(1) average case
10 um.erase("apple"); // O(1) average case
11 auto it = um.find("apple"); // O(1) average case
```

5: Multiset and Multimap Operations

```
1 // Multiset (allows duplicates)
2 multiset<int> ms;
3 ms.insert(5); // Can insert multiple 5s
4 ms.erase(5); // Erases all 5s
5 ms.erase(ms.find(5)); // Erases one occurrence
6
7 // Multimap (multiple values per key)
8 multimap<string, int> mm;
9 mm.insert({"key", 1});
10 mm.insert({"key", 2}); // Same key, different value
11 auto range = mm.equal_range("key"); // Get all values
```

2.1.7 Priority Queue and Heaps

Priority queues in C++ use comparators with reversed logic. By default, `priority_queue<int>` creates a max-heap.

6: Basic Priority Queue

```
1 // Max heap (default)
2 priority_queue<int> maxHeap;
3 // Min heap using greater<int>
4 priority_queue<int, vector<int>, greater<int>> minHeap;
5 // Custom comparator for complex types
6 struct Compare {
7     bool operator()(const Point& a, const Point& b) {
8         // Note: reversed logic compared to set/map
9         if (a.x != b.x) return a.x > b.x;
10        return a.y > b.y;
11    }
12 };
13 priority_queue<Point, vector<Point>, Compare> pq;
```

2.1.8 Stack and Queue

7: Stack and Queue Operations

```
1 // Stack (LIFO)
2 stack<int> s;
3 s.push(5);           // Add element
4 s.pop();             // Remove top element
5 s.top();             // Access top element
6 s.empty();           // Check if empty
7 s.size();            // Get size
8 // Queue (FIFO)
9 queue<int> q;
10 q.push(5);           // Add element
11 q.pop();             // Remove front element
12 q.front();           // Access front element
13 q.back();            // Access back element
14 q.empty();           // Check if empty
15 q.size();            // Get size
16 // Deque (double-ended queue)
17 deque<int> dq;
18 dq.push_front(5);    // Add to front
19 dq.push_back(5);     // Add to back
20 dq.pop_front();      // Remove from front
21 dq.pop_back();       // Remove from back
22 dq.front();          // Access front
23 dq.back();           // Access back
```


2.1.9 Bitset

Bitset provides space-efficient storage for boolean values.

8: Bitset Operations

```
1 // Bitset initialization
2 bitset<32> bs;           // 32-bit bitset
3 bitset<32> bs("1010");   // From binary string
4 bitset<32> bs(42);       // From integer
5
6 // Basic operations
7 bs.set(5);              // Set bit at position 5
8 bs.reset(5);            // Reset bit at position 5
9 bs.flip(5);             // Flip bit at position 5
10 bs.test(5);             // Check if bit is set
11 bs.count();             // Count set bits
12 bs.size();              // Total number of bits
13
14 // Bitwise operations
15 bitset<32> a("1010"), b("1100");
16 auto c = a & b;         // AND
17 auto d = a | b;         // OR
18 auto e = a ^ b;         // XOR
19 auto f = ~a;            // NOT
20
21 // Useful for competitive programming
22 bs.set();               // Set all bits
23 bs.reset();             // Reset all bits
24 bs.flip();              // Flip all bits
```

2.1.10 Bit Manipulation

Advanced bit manipulation techniques and tricks for competitive programming.

9: Basic Bit Operations

```
1 bool getBit(long long n, int i) { return (n >> i) & 1; }
2 long long setBit(long long n, int i) { return n | (1LL << i); }
3 long long clearBit(long long n, int i) { return n & ~(1LL << i); }
4 long long flipBit(long long n, int i) { return n ^ (1LL << i); }
5 long long updateBit(long long n, int i, bool val) {
6     return val ? setBit(n, i) : clearBit(n, i);
7 }
```

10: Bit Tricks

```
1 long long rightmostBit(long long n) { return n & -n; }
2 long long turnOffRightmost(long long n) { return n & (n - 1); }
3 long long turnOnRightmost(long long n) { return n | (n + 1); }
4 bool isPowerOfTwo(long long n) { return n > 0 && (n & (n - 1)) == 0; }
5 long long fastMod(long long n, long long mod) { return n & (mod - 1); }
6 int popcount(long long n) { return __builtin_popcountll(n); }
7 int leadingZeros(long long n) { return __builtin_clzll(n); }
8 int trailingZeros(long long n) { return __builtin_ctzll(n); }
9 int log2Floor(long long n) { return 63 - __builtin_clzll(n); }
```

11: Bitmask Patterns

```
1 long long createMask(int n) { return (1LL << n) - 1; }
2 long long extractBits(long long n, int i, int j) {
3     return (n >> i) & createMask(j - i + 1);
4 }
5 long long setRange(long long n, int i, int j) {
6     return n | (createMask(j - i + 1) << i);
7 }
8 long long clearRange(long long n, int i, int j) {
9     return n & ~(createMask(j - i + 1) << i);
10 }
11 long long swapBits(long long n, int i, int j) {
12     if (getBit(n, i) != getBit(n, j)) n = flipBit(flipBit(n, i), j);
13     return n;
14 }
15 long long reverseBits(long long n, int bits = 64) {
16     long long result = 0;
17     for (int i = 0; i < bits; i++)
18         if (getBit(n, i)) result = setBit(result, bits - 1 - i);
19     return result;
20 }
```

12: Subset Generation

```

1 // Generate all subsets:
2 // for(int mask = 0; mask < (1 << n); mask++)
3 // Generate all submasks:
4 // for(int sub = mask; ; sub = (sub - 1) & mask) { if(!sub) break; }
5 // Generate k-bit subsets:
6 // if(__builtin_popcount(mask) == k)
7
8 long long nextPermutation(long long n) {
9     long long c = n, c0 = 0, c1 = 0;
10    while (((c & 1) == 0) && c != 0) { c0++; c >>= 1; }
11    while ((c & 1) == 1) { c1++; c >>= 1; }
12    if (c0 + c1 >= 31) return -1;
13
14    long long pos = c0 + c1;
15    n = setBit(n, pos);
16    n = clearBit(n, pos - 1);
17    n = n & (~((1LL << (pos - 1)) - 1));
18    n = n | ((1LL << (c1 - 1)) - 1);
19    return n;
20 }

```

13: XOR Range

```

1 long long xorRange(long long n) {
2     int mod = n % 4;
3     return mod == 1 ? 1 : mod == 2 ? n + 1 : mod == 3 ? 0 : n;
4 }
5
6 long long xorRange(long long l, long long r) {
7     return xorRange(r) ^ xorRange(l - 1);
8 }

```

14: Find Two Unique Numbers

```

1 pair<int, int> findTwoUnique(vector<int>& arr) {
2     int xorAll = 0;
3     for (int x : arr) xorAll ^= x;
4     int rightmost = xorAll & -xorAll;
5     int x = 0, y = 0;
6     for (int num : arr) {
7         if (num & rightmost) x ^= num;
8         else y ^= num;
9     }
10    return {x, y};
11 }

```

15: Max XOR Subset

```

1 int maxXorSubset(vector<int> arr) {
2     for (int bit = 30; bit >= 0; bit--) {
3         int pivot = -1;
4         for (int i = 0; i < arr.size(); i++) {
5             if (getBit(arr[i], bit)) { pivot = i; break; }
6         }
7         if (pivot == -1) continue;
8
9         swap(arr[0], arr[pivot]);
10        for (int i = 1; i < arr.size(); i++) {
11            if (getBit(arr[i], bit)) arr[i] ^= arr[0];
12        }
13        arr.erase(arr.begin());
14    }
15    int result = 0;
16    for (int x : arr) result ^= x;
17    return result;
18 }

```

16: Bitmask DP Helpers

```

1
2 bool hasAdjacent(int mask, int n) {
3     return (mask & (mask << 1)) || (getBit(mask, 0) && getBit(mask, n - 1))
4     ;
5 }
6
7 int addIfValid(int mask, int pos, int n) {
8     if ((pos > 0 && getBit(mask, pos - 1)) ||
9         (pos < n - 1 && getBit(mask, pos + 1)) ||
10        (pos == 0 && n > 1 && getBit(mask, n - 1)) ||
11        (pos == n - 1 && n > 1 && getBit(mask, 0)))
12        return -1;
13    return setBit(mask, pos);
14 }
15
16 // Additional useful one-liners:
17 // Check if all bits in range [i,j] are set: ((n >> i) & createMask(j-i+1))
18 // == createMask(j-i+1)
19 // Toggle all bits: n ^ createMask(totalBits)
20 // Isolate rightmost n bits: n & createMask(n)
21 // Check if n has exactly k bits set: __builtin_popcount(n) == k
22 // Get position of rightmost set bit: __builtin_ctz(n)
23 // Get position of leftmost set bit: 31 - __builtin_clz(n) (for 32-bit)
24 // Set all bits from position i to end: n | (~0 << i)
25 // Clear all bits from position i to end: n & ((1 << i) - 1)

```

2.1.11 Ordered Set Template

C++ ordered sets using Policy-Based Data Structures (PBDS) for advanced operations.

17: Ordered Set Template

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4 // Ordered set (unique elements, ascending)
5 template<class T> using ordered_set = tree<T, null_type, less<T>,
    rb_tree_tag, tree_order_statistics_node_update>;
6
7 // Ordered multiset (allows duplicates, ascending)
8 template<class T> using ordered_multiset = tree<T, null_type, less_equal<T>,
    rb_tree_tag, tree_order_statistics_node_update>;
9
10 // Ordered set (unique elements, descending)
11 template<class T> using ordered_set_desc = tree<T, null_type, greater<T>,
    rb_tree_tag, tree_order_statistics_node_update>;
12
13 // Ordered multiset (allows duplicates, descending)
14 template<class T> using ordered_multiset_desc = tree<T, null_type,
    greater_equal<T>, rb_tree_tag, tree_order_statistics_node_update>;

```

18: Ordered Set Functions

```

1 // order_of_key(k): #elements < k; find_by_order(k): kth element (0-indexed)
2 // insert(), erase(), find() as in set
3 ordered_multiset<int> ss; ss.insert(5); ss.insert(2); ss.insert(7); ss.
    insert(2);
4 cout << ss.order_of_key(5) << endl; // 3
5 cout << *ss.find_by_order(1) << endl; // 2
6 ss.erase(ss.find_by_order(ss.order_of_key(2))); // erase one '2'

```

19: Custom Comparator for Ordered Set

```

1 template<class T>
2 struct custom_compare {
3     bool operator()(const T& a, const T& b) const {
4         if (a == b) return true; // Keep duplicates
5         return a > b; // Sort descending
6     }
7 };
8 template<class T> using ordered_multiset_custom = tree<T, null_type,
    custom_compare<T>, rb_tree_tag, tree_order_statistics_node_update>;

```

2.2 Advanced Data Structures

2.2.1 Segment Tree (Iterative)

Efficient range query data structure supporting point updates and range queries.

20: Segment Tree for Range Sum

```
1 struct SegmentTree {
2     int n;
3     vector<int> tree;
4
5     SegmentTree(const vector<int>& v) {
6         n = v.size();
7         tree.resize(n << 1);
8         for (int i = 0; i < n; i++)
9             tree[i + n] = v[i];
10        for (int i = n - 1; i > 0; i--)
11            tree[i] = tree[i << 1] + tree[i << 1 | 1];
12    }
13
14    void update(int pos, int value) {
15        tree[pos += n] = value;
16        for (pos >>= 1; pos > 0; pos >>= 1)
17            tree[pos] = tree[pos << 1] + tree[pos << 1 | 1];
18    }
19
20    int query(int l, int r) { // inclusive range [l, r]
21        int res = 0;
22        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
23            if (l & 1) res += tree[l++];
24            if (r & 1) res += tree[--r];
25        }
26        return res;
27    }
28};
```

21: Segment Tree Example Usage

```
1 int main() {
2     vector<int> a = {2, 1, 5, 3, 4};
3     SegmentTree st(a);
4
5     cout << st.query(1, 3) << "\n"; // 1 + 5 + 3 = 9
6     st.update(2, 0);                // a[2] = 0
7     cout << st.query(1, 3) << "\n"; // 1 + 0 + 3 = 4
8 }
```

22: Segment Tree for Range Maximum

```

1 struct SegmentTree {
2     int n;
3     vector<int> tree;
4
5     SegmentTree(const vector<int>& v) {
6         n = v.size();
7         tree.resize(n << 1);
8         for (int i = 0; i < n; i++)
9             tree[i + n] = v[i];
10        for (int i = n - 1; i > 0; i--)
11            tree[i] = max(tree[i << 1], tree[i << 1 | 1]);
12    }
13
14    void update(int pos, int value) {
15        tree[pos += n] = value;
16        for (pos >>= 1; pos > 0; pos >>= 1)
17            tree[pos] = max(tree[pos << 1], tree[pos << 1 | 1]);
18    }
19
20    int query(int l, int r) { // inclusive range [l, r]
21        int res = INT_MIN;
22        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
23            if (l & 1) res = max(res, tree[l++]);
24            if (r & 1) res = max(res, tree[--r]);
25        }
26        return res;
27    }
28 };

```

23: Segment Tree Max Example Usage

```

1 int main() {
2     vector<int> a = {2, 1, 5, 3, 4};
3     SegmentTree st(a);
4
5     cout << st.query(1, 3) << "\n"; // max(1, 5, 3) = 5
6     st.update(2, 0);                // a[2] = 0
7     cout << st.query(1, 3) << "\n"; // max(1, 0, 3) = 3
8 }

```

2.2.2 Disjoint Set Union (DSU)

Optimized union-find data structure with path compression and union by size.

24: DSU with Vector

```
1 struct DSU {
2     vector<int> parent, size;
3
4     DSU(int n) {
5         parent.resize(n);
6         size.resize(n);
7         for (int i = 0; i < n; i++) {
8             parent[i] = i;
9             size[i] = 1;
10        }
11    }
12
13    int findParent(int x) {
14        if (parent[x] == x) return x;
15        return parent[x] = findParent(parent[x]);
16    }
17
18    bool sameGroup(int x, int y) {
19        return findParent(x) == findParent(y);
20    }
21
22    void merge(int x, int y) {
23        int rootX = findParent(x);
24        int rootY = findParent(y);
25        if (rootX == rootY) return;
26        if (size[rootX] < size[rootY]) swap(rootX, rootY);
27        parent[rootY] = rootX;
28        size[rootX] += size[rootY];
29    }
30 };
```

25: DSU Example Usage

```
1 int main() {
2     DSU dsu(10);
3
4     dsu.merge(1, 2);
5     dsu.merge(2, 3);
6     dsu.merge(4, 5);
7
8     cout << (dsu.sameGroup(1, 3)) << "\n"; // 1 (true)
9     cout << (dsu.sameGroup(1, 5)) << "\n"; // 0 (false)
10 }
```


26: DSU with Unordered Map

```
1 struct DSUMap {
2     unordered_map<int, int> parent, size;
3
4     void makeSet(int x) {
5         if (!parent.count(x)) {
6             parent[x] = x;
7             size[x] = 1;
8         }
9     }
10
11     int findParent(int x) {
12         makeSet(x);
13         if (parent[x] == x) return x;
14         return parent[x] = findParent(parent[x]);
15     }
16
17     bool sameGroup(int x, int y) {
18         return findParent(x) == findParent(y);
19     }
20
21     void merge(int x, int y) {
22         int rootX = findParent(x);
23         int rootY = findParent(y);
24         if (rootX == rootY) return;
25         if (size[rootX] < size[rootY]) swap(rootX, rootY);
26         parent[rootY] = rootX;
27         size[rootX] += size[rootY];
28     }
29 };
```

27: DSU Map Example Usage

```
1 int main() {
2     DSUMap dsu;
3     dsu.merge(100, 200);
4     dsu.merge(200, 300);
5     dsu.merge(400, 500);
6
7     cout << dsu.sameGroup(100, 300) << "\n"; // 1 (true)
8     cout << dsu.sameGroup(100, 500) << "\n"; // 0 (false)
9 }
```

3 Graph Algorithms

3.1 Depth-First Search (DFS)

Depth-First Search is a graph traversal algorithm that explores as far as possible along each branch before backtracking.

28: DFS Implementation

```
1 vector<vector<int>> graph; // Adjacency list
2 vector<bool> visited;
3
4 void dfs(int node) {
5     visited[node] = true;
6     cout << node << " "; // Process node
7
8     for (int neighbor : graph[node]) {
9         if (!visited[neighbor]) {
10             dfs(neighbor);
11         }
12     }
13 }
14
15 // Initialize and run DFS
16 void runDFS(int start, int n) {
17     graph.resize(n);
18     visited.resize(n, false);
19     dfs(start);
20 }
```

DFS Notes

- **Time Complexity:** $O(V + E)$ where V = vertices, E = edges
- **Space Complexity:** $O(V)$ for recursion stack
- **Use Cases:** Exploring all possibilities, backtracking, connected components
- **Recursive Nature:** Uses recursion, can cause stack overflow for very deep graphs

29: DFS with Connected Components

```
1 vector<vector<int>> graph;
2 vector<bool> visited;
3
4 void dfs(int node) {
5     visited[node] = true;
6
7     for (int neighbor : graph[node]) {
8         if (!visited[neighbor]) {
9             dfs(neighbor);
10        }
11    }
12 }
13
14 int countComponents(int n) {
15     visited.resize(n, false);
16     int components = 0;
17
18     for (int i = 0; i < n; i++) {
19         if (!visited[i]) {
20             dfs(i);
21             components++;
22         }
23     }
24     return components;
25 }
```

Connected Components Notes

- **Application:** Finding number of disconnected subgraphs
- **Algorithm:** Run DFS from each unvisited node
- **Result:** Each DFS call discovers one connected component
- **Complexity:** Still $O(V + E)$ as each node/edge visited once

3.2 Breadth-First Search (BFS)

Breadth-First Search explores all vertices at the present depth before moving to vertices at the next depth level.

30: BFS Implementation

```
1 vector<vector<int>> graph; // Adjacency list
2 vector<bool> visited;
3
4 void bfs(int start) {
5     queue<int> q;
6     q.push(start);
7     visited[start] = true;
8
9     while (!q.empty()) {
10         int node = q.front();
11         q.pop();
12         cout << node << " "; // Process node
13
14         for (int neighbor : graph[node]) {
15             if (!visited[neighbor]) {
16                 visited[neighbor] = true;
17                 q.push(neighbor);
18             }
19         }
20     }
21 }
22
23 // Initialize and run BFS
24 void runBFS(int start, int n) {
25     graph.resize(n);
26     visited.resize(n, false);
27     bfs(start);
28 }
```

BFS Notes

- **Time Complexity:** $O(V + E)$ where V = vertices, E = edges
- **Space Complexity:** $O(V)$ for queue
- **Use Cases:** Shortest path in unweighted graphs, level-order traversal
- **Queue-based:** Uses queue, explores level by level

31: BFS with Distance Calculation

```
1 vector<vector<int>> graph;  
2 vector<int> distance;  
3  
4 void bfsWithDistance(int start, int n) {  
5     queue<int> q;  
6     distance.resize(n, -1);  
7  
8     q.push(start);  
9     distance[start] = 0;  
10  
11     while (!q.empty()) {  
12         int node = q.front();  
13         q.pop();  
14  
15         for (int neighbor : graph[node]) {  
16             if (distance[neighbor] == -1) {  
17                 distance[neighbor] = distance[node] + 1;  
18                 q.push(neighbor);  
19             }  
20         }  
21     }  
22 }
```

Distance BFS Notes

- **Shortest Path:** Guarantees shortest path in unweighted graphs
- **Distance Array:** Stores minimum distance from start to each node
- **Level Order:** Nodes at same distance processed together
- **Application:** Network routing, social network analysis

3.3 Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph.

32: Dijkstra's Algorithm

```
1 vector<vector<pair<int, int>>> graph; // {neighbor, weight}
2 vector<int> distance;
3
4 void dijkstra(int start, int n) {
5     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
6         int>>> pq;
7     distance.resize(n, INT_MAX);
8
9     distance[start] = 0;
10    pq.push({0, start});
11
12    while (!pq.empty()) {
13        int dist = pq.top().first;
14        int node = pq.top().second;
15        pq.pop();
16
17        if (dist > distance[node]) continue;
18
19        for (auto [neighbor, weight] : graph[node]) {
20            if (distance[node] + weight < distance[neighbor]) {
21                distance[neighbor] = distance[node] + weight;
22                pq.push({distance[neighbor], neighbor});
23            }
24        }
25    }
```

Dijkstra Notes

- **Time Complexity:** $O((V + E) \log V)$ with priority queue
- **Space Complexity:** $O(V)$ for distance array and priority queue
- **Requirement:** All edge weights must be non-negative
- **Greedy Algorithm:** Always picks the closest unvisited node

33: Dijkstra with Path Reconstruction

```
1 vector<vector<pair<int, int>>> graph;
2 vector<int> distance, parent;
3
4 void dijkstraWithPath(int start, int n) {
5     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
6         int>>> pq;
7     distance.resize(n, INT_MAX);
8     parent.resize(n, -1);
9
10    distance[start] = 0;
11    pq.push({0, start});
12
13    while (!pq.empty()) {
14        int dist = pq.top().first;
15        int node = pq.top().second;
16        pq.pop();
17
18        if (dist > distance[node]) continue;
19
20        for (auto [neighbor, weight] : graph[node]) {
21            if (distance[node] + weight < distance[neighbor]) {
22                distance[neighbor] = distance[node] + weight;
23                parent[neighbor] = node;
24                pq.push({distance[neighbor], neighbor});
25            }
26        }
27    }
28
29    vector<int> getPath(int end) {
30        vector<int> path;
31        for (int node = end; node != -1; node = parent[node]) {
32            path.push_back(node);
33        }
34        reverse(path.begin(), path.end());
35        return path;
36    }
```

Path Reconstruction Notes

- **Parent Array:** Stores predecessor of each node in shortest path
- **Path Recovery:** Backtrack from destination to source
- **Reverse Order:** Path is built backwards, then reversed
- **Application:** Navigation systems, network routing

3.4 Floyd-Warshall Algorithm

Floyd-Warshall finds shortest paths between all pairs of vertices in a weighted graph.

34: Floyd-Warshall Algorithm

```
1 int main() {
2     int INF = 1e9;
3     int n = 4;
4     vector<vector<int>> mat = {
5         {0, 3, INF, 7},
6         {8, 0, 2, INF},
7         {5, INF, 0, 1},
8         {2, INF, INF, 0}
9     };
10
11     for (int mid = 0; mid < n; mid++)
12         for (int from = 0; from < n; from++)
13             for (int to = 0; to < n; to++)
14                 mat[from][to] = min(mat[from][to], mat[from][mid] + mat[mid][to]);
15
16     for (int from = 0; from < n; from++) {
17         for (int to = 0; to < n; to++)
18             cout << (mat[from][to] == INF ? -1 : mat[from][to]) << " ";
19         cout << "\n";
20     }
21 }
```

Floyd-Warshall Notes

- **Time Complexity:** $O(V^3)$ - cubic time complexity
- **Space Complexity:** $O(V^2)$ for distance matrix
- **All Pairs:** Finds shortest path between every pair of vertices
- **Handles Negatives:** Can detect negative cycles

3.5 Topological Sort

Topological sort orders vertices in a directed acyclic graph (DAG) so that all edges point forward.

35: Topological Sort with DFS

```
1 vector<vector<int>> graph;
2 vector<bool> visited;
3 vector<int> topoOrder;
4
5 void dfs(int node) {
6     visited[node] = true;
7
8     for (int neighbor : graph[node]) {
9         if (!visited[neighbor]) {
10             dfs(neighbor);
11         }
12     }
13
14     topoOrder.push_back(node);
15 }
16
17 vector<int> topologicalSort(int n) {
18     visited.resize(n, false);
19     topoOrder.clear();
20
21     for (int i = 0; i < n; i++) {
22         if (!visited[i]) {
23             dfs(i);
24         }
25     }
26
27     reverse(topoOrder.begin(), topoOrder.end());
28     return topoOrder;
29 }
```

DFS Topological Sort Notes

- **Post-order DFS:** Add node after visiting all neighbors
- **Reverse Result:** Final order is reversed DFS post-order
- **Requirement:** Graph must be a DAG (no cycles)
- **Application:** Build order, dependency resolution

36: Topological Sort with Kahn's Algorithm

```
1 vector<vector<int>> graph;
2 vector<int> inDegree;
3
4 vector<int> kahnTopologicalSort(int n) {
5     queue<int> q;
6     vector<int> result;
7
8     // Calculate in-degrees
9     inDegree.resize(n, 0);
10    for (int i = 0; i < n; i++) {
11        for (int neighbor : graph[i]) {
12            inDegree[neighbor]++;
13        }
14    }
15
16    // Add nodes with in-degree 0
17    for (int i = 0; i < n; i++) {
18        if (inDegree[i] == 0) {
19            q.push(i);
20        }
21    }
22
23    while (!q.empty()) {
24        int node = q.front();
25        q.pop();
26        result.push_back(node);
27
28        for (int neighbor : graph[node]) {
29            inDegree[neighbor]--;
30            if (inDegree[neighbor] == 0) {
31                q.push(neighbor);
32            }
33        }
34    }
35
36    return result;
37 }
```

Kahn's Algorithm Notes

- **In-degree Tracking:** Count incoming edges for each node
- **Queue-based:** Process nodes with zero in-degree
- **Multiple Orders:** Can have multiple valid topological orders
- **Cycle Detection:** If result size < n, graph has cycle

3.6 Cycle Detection

Detecting cycles in directed and undirected graphs.

37: Cycle Detection in Undirected Graph

```
1 vector<vector<int>> graph;
2 vector<bool> visited;
3
4 bool hasCycleUndirected(int node, int parent) {
5     visited[node] = true;
6
7     for (int neighbor : graph[node]) {
8         if (!visited[neighbor]) {
9             if (hasCycleUndirected(neighbor, node)) {
10                 return true;
11             }
12         } else if (neighbor != parent) {
13             return true;
14         }
15     }
16     return false;
17 }
18
19 bool detectCycleUndirected(int n) {
20     visited.resize(n, false);
21
22     for (int i = 0; i < n; i++) {
23         if (!visited[i]) {
24             if (hasCycleUndirected(i, -1)) {
25                 return true;
26             }
27         }
28     }
29     return false;
30 }
```

Undirected Cycle Detection Notes

- **Parent Tracking:** Avoid revisiting parent node
- **Back Edge:** Cycle if neighbor is visited but not parent
- **DFS-based:** Uses DFS to explore graph
- **Application:** Validating trees, network topology

38: Cycle Detection in Directed Graph

```
1 vector<vector<int>> graph;
2 vector<bool> visited, recStack;
3
4 bool hasCycleDirected(int node) {
5     visited[node] = true;
6     recStack[node] = true;
7
8     for (int neighbor : graph[node]) {
9         if (!visited[neighbor]) {
10             if (hasCycleDirected(neighbor)) {
11                 return true;
12             }
13         } else if (recStack[neighbor]) {
14             return true;
15         }
16     }
17
18     recStack[node] = false;
19     return false;
20 }
21
22 bool detectCycleDirected(int n) {
23     visited.resize(n, false);
24     recStack.resize(n, false);
25
26     for (int i = 0; i < n; i++) {
27         if (!visited[i]) {
28             if (hasCycleDirected(i)) {
29                 return true;
30             }
31         }
32     }
33     return false;
34 }
```

Directed Cycle Detection Notes

- **Recursion Stack:** Track nodes in current recursion path
- **Back Edge:** Cycle if neighbor is in recursion stack
- **Two Arrays:** visited for all nodes, recStack for current path
- **Application:** Deadlock detection, DAG validation

4 Dynamic Programming

4.1 Longest Increasing Subsequence (LIS)

The Longest Increasing Subsequence problem finds the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

39: LIS - 2D DP Bottom-Up Implementation

```

1 int lengthOfLIS(vector<int>& nums) {
2     int n = nums.size();
3     vector<vector<int>> dp(n + 2, vector<int>(n + 2));
4
5     for (int i = n - 1; i >= 0; --i) {
6         for (int j = i - 1; j >= -1; --j) {
7             int curr = i + 1, prev = j + 1;
8             if (j == -1 || nums[i] > nums[j])
9                 dp[curr][prev] = dp[curr + 1][curr] + 1;
10            dp[curr][prev] = max(dp[curr][prev], dp[curr + 1][prev]);
11        }
12    }
13
14    // Reconstruct the LIS
15    vector<int> lis;
16    int i = 0, j = -1;
17    while (i < n) {
18        int curr = i + 1, prev = j + 1;
19        if (dp[curr][prev] == dp[curr + 1][curr] + 1 && (j == -1 || nums[i]
20            > nums[j])) {
21            lis.push_back(nums[i]);
22            j = i;
23        }
24        i++;
25    }
26    return dp[1][0];
27 }

```

LIS 2D DP Notes

- **Time Complexity:** $O(n^2)$ - quadratic time
- **Space Complexity:** $O(n^2)$ for 2D DP table
- **State Definition:** $dp[i+1][j+1]$ represents LIS from index i with last element at j
- **Reconstruction:** Can reconstruct the actual LIS sequence
- **Usage:** Use for understanding and simple cases

40: LIS - 1D DP Bottom-Up Implementation

```
1 int lengthOfLIS(vector<int>& nums) {  
2     int n = nums.size();  
3     vector<int> dp(n, 1);  
4     for (int i = n - 1; i >= 0; --i)  
5         for (int j = i + 1; j < n; ++j)  
6             if (nums[j] > nums[i])  
7                 dp[i] = max(dp[i], dp[j] + 1);  
8  
9     // Reconstruct the LIS  
10    int maxLen = *max_element(dp.begin(), dp.end());  
11    vector<int> lis;  
12    for (int i = 0; i < n && maxLen; ++i)  
13        if (dp[i] == maxLen) {  
14            lis.push_back(nums[i]);  
15            --maxLen;  
16        }  
17  
18    return *max_element(dp.begin(), dp.end());  
19 }
```

LIS 1D DP Notes

- **Time Complexity:** $O(n^2)$ with memoization
- **Space Complexity:** $O(n)$ for 1D DP array
- **State Definition:** $dp[i]$ is length of LIS ending at index i
- **Base Case:** $dp[i] = 1$ for all i (single element is valid LIS)
- **Advantage:** More space efficient than 2D approach

41: LIS - Recursive Implementation

```
1 int lengthOfLIS(const vector<int>& nums) {  
2     int n = nums.size();  
3     vector<vector<int>> dp(n + 1, vector<int>(n + 1, -1));  
4  
5     function<int(int, int)> calculateLIS = [&](int cur, int prev) {  
6         if (cur == n) return 0;  
7         int i = cur + 1, j = prev + 1;  
8         int& res = dp[i][j];  
9         if (res != -1) return res;  
10  
11         if (prev == -1 || nums[cur] > nums[prev])  
12             res = max(res, 1 + calculateLIS(cur + 1, cur));  
13  
14         res = max(res, calculateLIS(cur + 1, prev));  
15  
16         return res;  
17     };  
18  
19     return calculateLIS(0, -1);  
20 }
```

LIS Recursive Notes

- **Time Complexity:** $O(n^2)$ with memoization
- **Space Complexity:** $O(n^2)$ for DP table and recursion stack
- **Top-down DP:** Recursive approach with memoization
- **Base Case:** When $cur == n$, return 0
- **Memoization:** Stores results to avoid redundant calculations

42: LIS - Binary Search Implementation

```
1 int lengthOfLIS(const vector<int>& a) {
2     vector<int> lis;
3     for (int i = 0; i < a.size(); ++i) {
4         auto it = lower_bound(begin(lis), end(lis), a[i]);
5         it != end(lis) ? *it = a[i] : lis.push_back(a[i]);
6     }
7     return lis.size();
8 }
9
10 // Reconstruct the actual LIS sequence
11 vector<int> getLIS(const vector<int>& a) {
12     vector<int> lis, prev(a.size(), -1);
13     for (int i = 0; i < a.size(); ++i) {
14         auto it = lower_bound(begin(lis), end(lis), i, [&](int j, int k) {
15             return a[j] < a[k];
16         });
17         it != end(lis) ? *it = i : lis.push_back(i);
18         if (it != begin(lis)) prev[i] = *(it - 1);
19     }
20     vector<int> res;
21     for (int i = lis.back(); i != -1; i = prev[i]) {
22         res.push_back(a[i]);
23     }
24     reverse(begin(res), end(res));
25     return res;
26 }
```

LIS Binary Search Notes

- **Time Complexity:** $O(n \log n)$ - optimal approach
- **Space Complexity:** $O(n)$ for LIS array and prev array
- **Binary Search:** Uses `lower_bound` for efficient insertion
- **Optimal Solution:** Best time complexity for LIS problem
- **Usage:** Use for optimal time complexity in practice
- **Reconstruction:** Can reconstruct the actual LIS sequence

43: LIS - Segment Tree Implementation

```

1 struct SegmentTree {
2     int n;
3     vector<int> tree;
4
5     SegmentTree(int _n) {
6         n = _n;
7         tree.resize(2 * _n);
8     }
9
10    void update(int pos, int value) {
11        tree[pos += n] = value;
12        for (pos >>= 1; pos > 0; pos >>= 1)
13            tree[pos] = max(tree[pos << 1], tree[pos << 1 | 1]);
14    }
15
16    int query(int l, int r) {
17        int res = 0;
18        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
19            if (l & 1) res = max(res, tree[l++]);
20            if (r & 1) res = max(res, tree[--r]);
21        }
22        return res;
23    }
24 };
25
26 int lengthOfLIS(vector<int>& nums) {
27     SegmentTree seg(1e5 + 1);
28     int res = 0;
29     for (auto i : nums) {
30         i += 2e4; // Offset to handle negative numbers
31         int val = seg.query(0, i - 1) + 1; // Find max LIS ending before i
32         res = max(res, val);
33         seg.update(i, val); // Update the LIS at position i
34     }
35     return res;
36 }

```

LIS Segment Tree Notes

- **Time Complexity:** $O(n \log M)$ where M is the range of values
- **Space Complexity:** $O(M)$ for segment tree
- **Advanced Approach:** Uses segment tree for range queries
- **Coordinate Compression:** Can handle large value ranges
- **Usage:** Use when you need range queries or advanced applications
- **Offset:** $+2e4$ handles negative numbers

5 Backtracking

5.1 Subsets

Generate all possible subsets of a given array.

44: Subsets Implementation

```
1 #include <vector>
2 using namespace std;
3
4 vector<vector<int>> subsets(vector<int>& nums) {
5     vector<vector<int>> result;
6     vector<int> subset;
7
8     function<void(int)> generate = [&](int start) {
9         // Add the current subset to the result
10        result.push_back(subset);
11
12        // Try adding each remaining element to the current subset
13        for (int i = start; i < nums.size(); i++) {
14            subset.push_back(nums[i]);
15            generate(i + 1);
16            subset.pop_back();
17        }
18    };
19
20    generate(0);
21    return result;
22 }
```

Subsets Notes

- **Time Complexity:** $O(2^n)$ where n is the number of elements
- **Space Complexity:** $O(2^n)$ to store all subsets
- **Backtracking Pattern:** Choose \rightarrow Recurse \rightarrow Unchoose
- **Natural Generation:** Each recursive call decides whether to include each element
- **Empty Set:** Includes the empty set as a valid subset
- **No Duplicates:** Avoids duplicates by only considering elements from current index forward

5.2 Permutations

Generate all possible permutations of a given array.

45: Permutations Without Duplicates

```
1 #include <vector>
2 using namespace std;
3
4 vector<vector<int>> permuteUnique(vector<int>& nums) {
5     vector<vector<int>> result;
6     vector<int> comb;
7     vector<bool> visited(nums.size(), false);
8
9     function<void()> permute = [&]() {
10         if (comb.size() == nums.size()) {
11             result.push_back(comb);
12             return;
13         }
14         for (int i = 0; i < nums.size(); i++) {
15             if (visited[i]) continue;
16             visited[i] = true;
17             comb.push_back(nums[i]);
18             permute();
19             comb.pop_back();
20             visited[i] = false;
21         }
22     };
23
24     permute();
25     return result;
26 }
```

Permutations Without Duplicates Notes

- **Time Complexity:** $O(n!)$ where n is the number of elements
- **Space Complexity:** $O(n!)$ to store all permutations
- **Visited Array:** Tracks which elements have been used
- **Perfect for Unique Elements:** Arrays with unique elements
- **All Orderings:** Generates all possible orderings of input array
- **Backtracking:** Uses visited array to prevent reusing elements

46: Permutations With Duplicates

```
1 #include <vector>
2 #include <unordered_map>
3 using namespace std;
4
5 vector<vector<int>> permuteWithDuplicates(vector<int>& nums) {
6     vector<vector<int>> result;
7     unordered_map<int, int> counter;
8     for (int num : nums) counter[num]++;
9
10    vector<int> comb;
11
12    function<void()> permute = [&]() {
13        if (comb.size() == nums.size()) {
14            result.push_back(comb);
15            return;
16        }
17        for (auto& item : counter) {
18            int num = item.first;
19            int count = item.second;
20            if (count == 0) continue;
21            comb.push_back(num);
22            counter[num]--;
23            permute();
24            comb.pop_back();
25            counter[num]++;
26        }
27    };
28
29    permute();
30    return result;
31 }
```

Permutations With Duplicates Notes

- **Time Complexity:** $O(n! \times n)$ due to factorial permutations and element checking
- **Space Complexity:** $O(n!)$ to store the resulting permutations
- **Unordered Map:** Tracks frequency of each element
- **Prevents Duplicates:** More efficient for inputs with repeated elements
- **Counter Management:** Decrements and increments counter during backtracking
- **Usage:** Use when input array contains duplicate elements

5.3 Combinations

Generate all possible combinations of k elements from an array.

47: Combinations Implementation

```
1 #include <vector>
2 using namespace std;
3
4 vector<vector<int>> combinations(vector<int>& nums, int k) {
5     vector<vector<int>> result;
6     vector<int> comb;
7
8     function<void(int)> combine = [&](int start) {
9         if (comb.size() == k) {
10             result.push_back(comb);
11             return;
12         }
13         for (int i = start; i < nums.size(); i++) {
14             comb.push_back(nums[i]);
15             combine(i + 1);
16             comb.pop_back();
17         }
18     };
19
20     combine(0);
21     return result;
22 }
```

Combinations Notes

- **Time Complexity:** $O(C(n,k))$ or $O(n!/(k!(n-k)!))$ where n is number of elements and k is size of each combination
- **Space Complexity:** $O(C(n,k))$ to store all combinations
- **Starting Index:** Uses start parameter to avoid duplicates
- **Size Constraint:** Generates combinations of exactly size k
- **No Reuse:** No element is used more than once in each combination
- **Order Independent:** Unlike permutations, order doesn't matter in combinations

6 String Algorithms

6.1 C++ STL String Functions

Essential string manipulation functions from the C++ Standard Library.

48: STL String Functions

```
1 #include <string>
2 #include <algorithm>
3
4 string s = "Hello World";
5 // Basic operations
6 s.length();           // Get string length
7 s.size();             // Same as length()
8 s.empty();            // Check if empty
9 s.clear();            // Clear string
10 // Access elements
11 s[0];                // Access character
12 s.at(0);             // Bounds-checked access
13 s.front();           // First character
14 s.back();            // Last character
15 // String manipulation
16 s.substr(0, 5);       // Substring
17 s.find("World");      // Find substring
18 s.replace(0, 5, "Hi"); // Replace substring
19 s.insert(5, " ");     // Insert at position
20 // String algorithms
21 reverse(s.begin(), s.end()); // Reverse string
22 sort(s.begin(), s.end());    // Sort characters
23 transform(s.begin(), s.end(), s.begin(), ::tolower); // To lowercase
24 transform(s.begin(), s.end(), s.begin(), ::toupper); // To uppercase
25 // String concatenation
26 string s1 = "Hello";
27 string s2 = "World";
28 string result = s1 + " " + s2; // Concatenation
29 s1.append(s2);                // Append to string
30 s1 += s2;                    // Append operator
```

STL String Notes

- **Time Complexity:** Most operations $O(1)$ or $O(n)$
- **Memory Efficient:** String uses dynamic allocation
- **STL Algorithms:** Can use all STL algorithms on strings
- **Character Access:** Direct indexing and bounds-checked access

6.2 Longest Substring Without Repeating Characters

Find the length of the longest substring without repeating characters.

49: Longest Substring Without Repeating Characters

```
1 int lengthOfLongestSubstring(string s) {  
2     vector<int> charIndex(128, -1); // ASCII characters  
3     int maxLength = 0;  
4     int start = 0;  
5  
6     for (int end = 0; end < s.length(); end++) {  
7         char currentChar = s[end];  
8  
9         // If character already seen, update start  
10        if (charIndex[currentChar] >= start) {  
11            start = charIndex[currentChar] + 1;  
12        }  
13  
14        charIndex[currentChar] = end;  
15        maxLength = max(maxLength, end - start + 1);  
16    }  
17  
18    return maxLength;  
19 }
```

Longest Substring Notes

- **Sliding Window:** Uses two pointers technique
- **Time Complexity:** $O(n)$ where n is string length
- **Space Complexity:** $O(1)$ for fixed alphabet size
- **Character Tracking:** Uses array to track last position

6.3 Trie (Prefix Tree)

A trie is a tree-like data structure used to store a dynamic set of strings.

50: Trie Implementation

```

1 struct TrieNode {
2     unordered_map<char, TrieNode*> child;
3     bool word = false;
4 };
5 struct Trie {
6     TrieNode* root = new TrieNode();
7     void insert(const string& word) {
8         TrieNode* node = root;
9         for (char ch : word) {
10             if (!node->child.count(ch)) {
11                 node->child[ch] = new TrieNode();
12             }
13             node = node->child[ch];
14         }
15         node->word = true;
16     }
17     bool search(const string& word) {
18         TrieNode* node = root;
19         for (char ch : word) {
20             if (!node->child.count(ch)) return false;
21             node = node->child[ch];
22         }
23         return node->word;
24     }
25     bool startsWith(const string& prefix) {
26         TrieNode* node = root;
27         for (char ch : prefix) {
28             if (!node->child.count(ch)) return false;
29             node = node->child[ch];
30         }
31         return true;
32     }
33 };

```

Trie Notes

- **Time Complexity:** $O(m)$ where m is string length for insertion and search
- **Space Complexity:** $O(\text{ALPHABET_SIZE} \times N \times M)$
- **Applications:** Prefix matching, autocomplete, spell checking
- **Memory Usage:** Can be memory intensive for large datasets
- **Node Structure:** Each node stores children in `unordered_map` and word flag

7 Mathematics

7.1 Fast Power (Binary Exponentiation)

Efficiently compute large powers using binary exponentiation.

51: Binary Exponentiation - Iterative

```
1 int64_t power(int64_t base, int64_t exp) {
2     int64_t result = 1;
3     while (exp > 0) {
4         if (exp & 1) result *= base;
5         base *= base;
6         exp >>= 1;
7     }
8     return result;
9 }
```

52: Modular Exponentiation

```
1 int64_t modPower(int64_t base, int64_t exp, int64_t mod) {
2     int64_t result = 1;
3     base = base % mod;
4     while (exp > 0) {
5         if (exp & 1) result = (result * base) % mod;
6         base = (base * base) % mod;
7         exp >>= 1;
8     }
9     return result;
10 }
```

Modular Exponentiation Notes

- **Time Complexity:** $O(\log \text{exp})$ - logarithmic time
- **Space Complexity:** $O(1)$ constant space
- **Modulo Arithmetic:** Handles large numbers with modulo
- **Overflow Prevention:** Essential for competitive programming
- **Applications:** Cryptography, number theory problems

7.2 GCD and LCM Functions

Greatest Common Divisor and Least Common Multiple functions.

53: GCD and LCM Functions

```
1 int gcd(int a, int b) {  
2     while (b != 0) {  
3         a %= b;  
4         swap(a, b);  
5     }  
6     return a;  
7 }  
8  
9 int lcm(int a, int b) {  
10    return (a / gcd(a, b)) * b;  
11 }
```

GCD/LCM Notes

- **Time Complexity:** $O(\log \min(a,b))$ for GCD
- **Space Complexity:** $O(1)$ constant space
- **Euclidean Algorithm:** Efficient GCD calculation
- **LCM Formula:** $\text{LCM}(a,b) = (a \times b) / \text{GCD}(a,b)$
- **Applications:** Number theory, fraction simplification

7.3 Combinatorics

Basic combinatorial functions with modular arithmetic support.

54: Standard nCr and nPr

```
1 // Don't use for n > 67 (int64_t overflow)
2 int64_t nCr(int n, int r) {
3     if (r < 0 || r > n) return 0;
4     if (r > n - r) r = n - r;
5     int64_t res = 1;
6     for (int i = 0; i < r; ++i) {
7         res *= (n - i);
8         res /= (i + 1);
9     }
10    return res;
11 }
12
13 // Don't use for n > 20 or large r (int64_t overflow)
14 int64_t nPr(int n, int r) {
15     if (r < 0 || r > n) return 0;
16     int64_t res = 1;
17     for (int i = 0; i < r; ++i)
18         res *= (n - i);
19     return res;
20 }
```

Standard Combinatorics Notes

- **Time Complexity:** $O(r)$ for both nCr and nPr
- **Space Complexity:** $O(1)$ constant space
- **Limits:** $n \leq 67$ for nCr, $n \leq 20$ for nPr
- **Optimization:** nCr uses symmetry $C(n,r) = C(n,n-r)$
- **Applications:** Probability, counting problems

55: Combinatorics with Modular Arithmetic

```

1 #include <vector>
2 using namespace std;
3
4 class Combinatorics {
5 private:
6     static const int MOD = 1000000007;
7     vector<int64_t> f, inv;
8
9     int64_t pow(int64_t b, int64_t e) const {
10         int64_t r = 1;
11         while (e) {
12             if (e & 1) r = r * b % MOD;
13             b = b * b % MOD;
14             e >>= 1;
15         }
16         return r;
17     }
18
19 public:
20     Combinatorics(int n) : f(n + 1), inv(n + 1) {
21         f[0] = 1;
22         for (int i = 1; i <= n; ++i)
23             f[i] = f[i - 1] * i % MOD;
24         inv[n] = pow(f[n], MOD - 2);
25         for (int i = n - 1; i >= 0; --i)
26             inv[i] = inv[i + 1] * (i + 1) % MOD;
27     }
28     int64_t nCr(int n, int r) const {
29         if (r < 0 || r > n) return 0;
30         return f[n] * inv[r] % MOD * inv[n - r] % MOD;
31     }
32     int64_t nPr(int n, int r) const {
33         if (r < 0 || r > n) return 0;
34         return f[n] * inv[n - r] % MOD;
35     }
36 };

```

Modular Combinatorics Notes

- **Preprocessing:** $O(n)$ time and space for setup
- **Query Time:** $O(1)$ per nCr/nPr call
- **Limits:** n up to 10^6 (uses 16MB for $n=10^6$)
- **Features:** Handles large n , fast for many queries
- **Fermat's Little Theorem:** Uses for modular inverse
- **Applications:** Large combinatorial problems

7.4 Sieve of Eratosthenes

Efficient algorithm to find all prime numbers up to a given limit.

56: Sieve of Eratosthenes

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 // Time:  $O(n \log \log n)$ , Space:  $O(n)$ 
4 // Range: n up to  $10^7$  (typical CP limit)
5 // Memory: ~40MB for  $n=10^7$ 
6 class Sieve {
7 public:
8     vector<int> prime_factor, primes;
9
10    Sieve(int n) {
11        prime_factor.resize(n + 1);
12        for (int i = 0; i <= n; i++) prime_factor[i] = i;
13        for (int i = 2; i <= n; i++) {
14            if (prime_factor[i] == i) {
15                primes.push_back(i);
16                for (int j = i * i; j <= n; j += i)
17                    if (prime_factor[j] == j) prime_factor[j] = i;
18            }
19        }
20    }
21 };
22
23 int main() {
24     Sieve sieve(100);
25     for (int p : sieve.primes) cout << p << " ";
26     cout << "\n";
27     for (int i = 12; i <= 15; i++) {
28         cout << i << ": prime_factor=" << sieve.prime_factor[i] << "\n";
29     }
30     return 0;
31 }

```

Sieve Notes

- **Time Complexity:** $O(n \log \log n)$ - nearly linear
- **Space Complexity:** $O(n)$ for boolean array
- **Prime Factors:** `sieve.prime_factor[x]` gives smallest prime factor
- **Prime List:** `sieve.primes` contains all primes up to n
- **Memory Usage:** 40MB for $n=10^7$
- **Applications:** Prime factorization, number theory

8 Notes & Utilities

8.1 Binary Conversions

Convert numbers between different bases.

57: Binary to Decimal Conversion

```
1 // Convert binary string to decimal integer
2 string binaryStr = "1010";
3 int decimal = stoll(binaryStr, nullptr, 2);
4 // Result: 10
5
6 // Using bitset for larger binary strings
7 #include <bitset>
8 const int N = 32; // Enough for standard integers
9 int decimal = bitset<N>("1010").to_ulong();
10 // Result: 10
11
12 // For longer binary strings
13 const int LARGE_N = 10000; // For very large binary strings
14 unsigned long long largeDecimal = bitset<LARGE_N>(longBinaryStr).to_ullong()
    ;
```

58: Decimal to Binary Conversion

```
1 #include <bitset>
2
3 // Convert decimal to binary string
4 int decimal = 10;
5 const int N = 8; // Number of bits to represent
6 string binaryStr = bitset<N>(decimal).to_string();
7 // Result: "00001010"
8
9 // Remove leading zeros if needed
10 binaryStr = binaryStr.substr(binaryStr.find('1') != string::npos ? binaryStr
    .find('1') : N-1);
11 // Result: "1010"
12
13 // Using std::format (C++20)
14 #include <format>
15 string binaryStr = format("{:b}", decimal);
16 // Result: "1010"
```

8.2 Coordinate Compression

Efficiently map large values to smaller ranges for data structures.

59: Coordinate Compression Template

```

1  template <typename T>
2  class Compress {
3      vector<T> vals;
4      unordered_map<T, int> idx;
5
6  public:
7      Compress(const vector<T>& input) {
8          vals = input;
9          sort(vals.begin(), vals.end());
10         vals.erase(unique(vals.begin(), vals.end()), vals.end());
11         for (int i = 0; i < vals.size(); i++)
12             idx[vals[i]] = i;
13     }
14
15     int operator[](const T& x) const { return idx.at(x); }
16     T orig(int i) const { return vals.at(i); }
17     int size() const { return vals.size(); }
18 };

```

60: Coordinate Compression Example

```

1  // Basic usage
2  vector<int> data = {1000000, 5, 10000, 6, 7, 1000};
3  Compress<int> comp(data);
4
5  // Convert original value to compressed index
6  for (int x : data) {
7      cout << x << " -> " << comp[x] << endl;
8  }
9  // Output: 1000000->5, 5->0, 10000->3, 6->1, 7->2, 1000->4
10
11 // Get original value from compressed index
12 for (int i = 0; i < comp.size(); i++) {
13     cout << i << " -> " << comp.orig(i) << endl;
14 }
15 // Output: 0->5, 1->6, 2->7, 3->1000, 4->10000, 5->1000000

```

Coordinate Compression Notes

- **Time Complexity:** $O(N \log N)$ for construction, $O(1)$ for lookup
- **Space Complexity:** $O(N)$ for sorted list and hashmap

8.3 Performance Utilities

Tools for measuring and optimizing code performance.

61: Measure Time Utility

```
1 #include <iostream>
2 #include <chrono>
3 #include <cstdint>
4 #include <iomanip>
5 using namespace std;
6
7 template<typename Func, typename... Args>
8 double measure(Func&& f, Args&&... args) {
9     auto start = chrono::high_resolution_clock::now();
10    forward<Func>(f)(forward<Args>(args)...);
11    auto end = chrono::high_resolution_clock::now();
12    chrono::duration<double, milli> elapsed = end - start;
13    return elapsed.count();
14 }
15
16 int main() {
17     cout << fixed << setprecision(4);
18
19     double t1 = measure(funcVoid);
20     cout << "funcVoid took " << t1 << " ms\n";
21
22     int64_t res = 0;
23     auto wrapper = [&](int n) { res = funcInt(n); };
24     double t2 = measure(wrapper, 1000000);
25     cout << "funcInt took " << t2 << " ms, sum = " << res << "\n";
26
27     return 0;
28 }
```

Measure Time Notes

- **Template Function:** Works with any callable and arguments
- **High Resolution:** Uses `high_resolution_clock` for precision
- **Millisecond Precision:** Returns time in milliseconds
- **Applications:** Performance analysis, algorithm comparison
- **Wrapper Usage:** Use lambda wrapper for functions with return values

8.4 Random Number Generation

Generate random numbers for testing and simulation.

62: Random Number Generator

```
1 #include <iostream>
2 #include <random>
3 #include <ctime>
4 using namespace std;
5
6 mt19937_64 ran(time(nullptr));
7
8 int r(int a, int b) {
9     return ran() % (abs(b - a) + 1) + min(a, b);
10 }
```

63: Random Number Generator Example Usage

```
1 // Generate 5 random numbers between 1 and 100
2 for (int i = 0; i < 5; ++i) {
3     cout << r(1, 100) << " ";
4 }
5 // Output: e.g. 42 17 89 3 76
```

Number Generator Notes

- **High Quality:** Uses mt19937_64 for 64-bit random numbers
- **Range Function:** $r(a, b)$ returns random integer in $[\min(a,b), \max(a,b)]$
- **Time Seeding:** Seeded with current time
- **Applications:** Test case generation, competitive programming
- **Note:** Not cryptographically secure

8.5 String Utilities

Common string manipulation and parsing utilities.

64: String Split Utility

```
1 template<typename T>
2 vector<T> split(const string& line, char delimiter = ' ') {
3     vector<T> result;
4     stringstream ss(line);
5     string token;
6
7     while (getline(ss, token, delimiter)) {
8         stringstream convert(token);
9         T value;
10        convert >> value;
11        if (!convert.fail()) {
12            result.push_back(value);
13        }
14    }
15
16    return result;
17 }
18
19 // Basic string split to vector<string>
20 vector<string> split(const string& line, char delimiter = ' ') {
21     vector<string> result;
22     stringstream ss(line);
23     string token;
24     while (getline(ss, token, delimiter)) {
25         result.push_back(token);
26     }
27     return result;
28 }
```

65: String Split Examples

```
1 // Split string to vector<int>
2 vector<int> ints = split<int>("10 20 30"); // [10, 20, 30]
3
4 // Split string to vector<double> with comma delimiter
5 vector<double> doubles = split<double>("3.14,2.71,1.41", ','); // [3.14,
6     2.71, 1.41]
7
8 // Split input line to vector<int>
9 string input; getline(cin, input);
10 vector<int> values = split<int>(input);
```

8.6 Custom Comparators

Custom comparators for sets, maps, and priority queues in C++.

66: Custom Comparator Approaches

```
1 // Struct comparator (descending):
2 struct Desc { bool operator()(int a, int b) const { return a > b; } };
3 set<int, Desc> s;
4 map<int, int, Desc> m;
5 priority_queue<int, vector<int>, Desc> pq;
6
7 // Lambda comparator (descending):
8 auto cmp = [](int a, int b) { return a > b; };
9 set<int, decltype(cmp)> s2(cmp);
10 map<int, int, decltype(cmp)> m2(cmp);
11 priority_queue<int, vector<int>, decltype(cmp)> pq2(cmp);
12
13 // Priority queues:
14 priority_queue<int> maxHeap; // max-heap (default)
15 priority_queue<int, vector<int>, greater<int>> minHeap; // min-heap
```

Custom Comparators Notes

- **Struct Approach:** No need to pass comparator instance to constructor
- **Lambda Approach:** Must pass comparator instance to constructor
- **decltype:** Use to deduce lambda function type
- **Priority Queue Logic:** Comparator logic is reversed compared to set/map
- **Applications:** Reverse ordering, custom object sorting, specialized sorting
- **Important:** For priority_queue, comparator returns true if first argument should come after second

9 Searching Algorithms

9.1 Binary Search

67: Binary Search Implementation

```
1 // Standard binary search
2 int binarySearch(vector<int>& arr, int target) {
3     int low = 0, high = arr.size() - 1;
4     while (low <= high) {
5         int mid = (low + high) / 2;
6         if (arr[mid] == target) return mid;
7         else if (arr[mid] < target) low = mid + 1;
8         else high = mid - 1;
9     }
10    return -1;
11 }
12
13 // Using STL binary_search
14 bool found = binary_search(arr.begin(), arr.end(), target);
```

9.2 Lower Bound / Upper Bound

68: Lower Bound Implementation

```
1 // Manual lower bound
2 int lowerBound(vector<int>& arr, int target) {
3     int low = 0, high = arr.size() - 1, index = -1;
4     while (low <= high) {
5         int mid = (low + high) / 2;
6         if (arr[mid] >= target) {
7             index = mid;
8             high = mid - 1;
9         } else {
10            low = mid + 1;
11        }
12    }
13    return index;
14 }
15
16 // Using STL lower_bound
17 int index = lower_bound(arr.begin(), arr.end(), target) - arr.begin();
```

69: Upper Bound Implementation

```
1 // Manual upper bound
2 int upperBound(vector<int>& arr, int target) {
3     int low = 0, high = arr.size() - 1, index = -1;
4     while (low <= high) {
5         int mid = (low + high) / 2;
6         if (arr[mid] > target) {
7             index = mid;
8             high = mid - 1;
9         } else {
10             low = mid + 1;
11         }
12     }
13     return index;
14 }
15
16 // Using STL upper_bound
17 int index = upper_bound(arr.begin(), arr.end(), target) - arr.begin();
```

9.3 Binary Search on Answer

```
1 // Binary search on answer when we need to find minimum/maximum
2 // that satisfies some condition
3 long long binarySearchOnAnswer(long long left, long long right, function<
4     bool(long long)> check) {
5     long long ans = right;
6     while (left <= right) {
7         long long mid = left + (right - left) / 2;
8         if (check(mid)) {
9             ans = mid;
10            right = mid - 1; // For minimum
11            // left = mid + 1; // For maximum
12        } else {
13            left = mid + 1; // For minimum
14            // right = mid - 1; // For maximum
15        }
16    }
17    return ans;
18 }
19 // Example: Find minimum time to complete a task
20 bool canComplete(vector<int>& tasks, long long time) {
21     long long total = 0;
22     for (int task : tasks) {
23         total += (time + task - 1) / task; // Ceiling division
24     }
25     return total <= time;
26 }
```

9.4 Ternary Search

```
1 // Integer ternary search for unimodal function
2 int ternarySearchInt(int left, int right, function<int(int)> f) {
3     while (right - left > 3) {
4         int mid1 = left + (right - left) / 3;
5         int mid2 = right - (right - left) / 3;
6
7         if (f(mid1) < f(mid2)) {
8             left = mid1;
9         } else {
10             right = mid2;
11         }
12     }
13
14     // Check remaining points
15     int best = left;
16     for (int i = left; i <= right; i++) {
17         if (f(i) < f(best)) best = i;
18     }
19
20     return best;
21 }
22
23 // Floating point ternary search
24 double ternarySearchDouble(double left, double right, function<double(double)> f, double eps = 1e-9) {
25     while (right - left > eps) {
26         double mid1 = left + (right - left) / 3;
27         double mid2 = right - (right - left) / 3;
28
29         if (f(mid1) < f(mid2)) {
30             left = mid1;
31         } else {
32             right = mid2;
33         }
34     }
35
36     return left;
37 }
```

10 Geometry (CP Basics)

10.1 Points & Vectors

70: Point and Vector Structure

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 const double EPS = 1e-9;
5 const double PI = acos(-1.0);
6
7 struct Point {
8     double x, y;
9     Point(double x = 0, double y = 0) : x(x), y(y) {}
10
11     Point operator+(Point p) { return Point(x + p.x, y + p.y); }
12     Point operator-(Point p) { return Point(x - p.x, y - p.y); }
13     Point operator*(double t) { return Point(x * t, y * t); }
14
15     double dot(Point p) { return x * p.x + y * p.y; }
16     double cross(Point p) { return x * p.y - y * p.x; }
17     double norm() { return sqrt(x * x + y * y); }
18     Point rotate(double a) {
19         return Point(x*cos(a) - y*sin(a), x*sin(a) + y*cos(a));
20     }
21 };
22
23 double dist(Point a, Point b) { return (a - b).norm(); }
```


10.2 Lines & Segments

71: Line and Segment Operations

```
1 // Distance point to line
2 double distPointLine(Point p, Point a, Point b) {
3     return abs((b - a).cross(p - a)) / (b - a).norm();
4 }
5
6 // Distance point to segment
7 double distPointSeg(Point p, Point a, Point b) {
8     if ((b - a).dot(p - a) < 0) return (p - a).norm();
9     if ((a - b).dot(p - b) < 0) return (p - b).norm();
10    return distPointLine(p, a, b);
11 }
12
13 // Line intersection
14 bool lineIntersect(Point a1, Point b1, Point a2, Point b2, Point& res) {
15     Point d1 = b1 - a1, d2 = b2 - a2;
16     double cross = d1.cross(d2);
17     if (abs(cross) < EPS) return false;
18     double t = (a2 - a1).cross(d2) / cross;
19     res = a1 + d1 * t;
20     return true;
21 }
22
23 // Segment intersection
24 bool segIntersect(Point a1, Point b1, Point a2, Point b2) {
25     Point d1 = b1 - a1, d2 = b2 - a2;
26     double cross = d1.cross(d2);
27     if (abs(cross) < EPS) return false;
28     double t1 = (a2 - a1).cross(d2) / cross;
29     double t2 = (a2 - a1).cross(d1) / cross;
30     return t1 >= 0 && t1 <= 1 && t2 >= 0 && t2 <= 1;
31 }
```

10.3 Polygons & Areas

72: Polygon Area Calculations

```
1 // Polygon area (signed)
2 double polyArea(vector<Point>& poly) {
3     double area = 0;
4     int n = poly.size();
5     for (int i = 0; i < n; i++)
6         area += poly[i].cross(poly[(i + 1) % n]);
7     return area / 2.0;
8 }
9
10 // Alternative polygon area (triangulation from first vertex)
11 double polyAreaAlt(vector<Point>& poly) {
12     double area = 0;
13     for (int i = 1; i < poly.size() - 1; i++)
14         area += (poly[i] - poly[0]).cross(poly[i + 1] - poly[0]);
15     return abs(area / 2.0);
16 }
17
18 // Triangle area using cross product
19 double triArea(Point a, Point b, Point c) {
20     return abs((b - a).cross(c - a)) / 2.0;
21 }
22
23 // Triangle area using Heron's formula (given side lengths)
24 double triAreaHeron(double a, double b, double c) {
25     double s = (a + b + c) * 0.5;
26     return sqrt(s * (s - a) * (s - b) * (s - c));
27 }
28
29 // Triangle area using coordinate formula
30 double triAreaCoord(Point a, Point b, Point c) {
31     return abs(a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y)) /
32         2.0;
33 }
```

73: Polygon Centroid and Lattice Points

```

1 // Polygon centroid
2 Point polyCentroid(vector<Point>& poly) {
3     Point centroid(0, 0);
4     double area = 0;
5     int n = poly.size();
6     for (int i = 0; i < n; i++) {
7         int j = (i + 1) % n;
8         double cross = poly[i].cross(poly[j]);
9         area += cross;
10        centroid = centroid + (poly[i] + poly[j]) * cross;
11    }
12    area /= 2.0;
13    return centroid * (1.0 / (6.0 * area));
14 }
15
16 // Lattice points (Pick's theorem:  $A = I + B/2 - 1$ )
17 int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }
18
19 // Boundary lattice points on segment
20 int boundaryPoints(Point a, Point b) {
21     return gcd(abs((int)(b.x - a.x)), abs((int)(b.y - a.y))) + 1;
22 }
23
24 // Interior lattice points using Pick's theorem
25 int interiorPoints(vector<Point>& poly) {
26     int boundary = 0;
27     int n = poly.size();
28     for (int i = 0; i < n; i++) {
29         boundary += boundaryPoints(poly[i], poly[(i + 1) % n]) - 1;
30     }
31     return (int)abs(polyArea(poly)) - boundary / 2 + 1;
32 }

```

74: Point in Polygon and Convex Hull

```
1 bool pointInPoly(Point p, vector<Point>& poly) {
2     int n = poly.size();
3     bool inside = false;
4     for (int i = 0, j = n - 1; i < n; j = i++) {
5         if (((poly[i].y > p.y) != (poly[j].y > p.y)) &&
6             (p.x < (poly[j].x - poly[i].x) * (p.y - poly[i].y) / (poly[j].y
7                 - poly[i].y) + poly[i].x))
8             inside = !inside;
9     }
10    return inside;
11 }
12
13 // Convex hull
14 vector<Point> convexHull(vector<Point> pts) {
15     sort(pts.begin(), pts.end(), [](Point a, Point b) {
16         return a.x < b.x || (a.x == b.x && a.y < b.y);
17     });
18     vector<Point> hull;
19     // Lower hull
20     for (Point p : pts) {
21         while (hull.size() >= 2 && (hull[hull.size()-1] - hull[hull.size()-2]).cross(p - hull[hull.size()-2]) <= 0)
22             hull.pop_back();
23         hull.push_back(p);
24     }
25
26     // Upper hull
27     int t = hull.size() + 1;
28     for (int i = pts.size() - 2; i >= 0; i--) {
29         while (hull.size() >= t && (hull[hull.size()-1] - hull[hull.size()-2]).cross(pts[i] - hull[hull.size()-2]) <= 0)
30             hull.pop_back();
31         hull.push_back(pts[i]);
32     }
33
34     hull.pop_back();
35     return hull;
36 }
```

10.4 Circles and Advanced Geometry

75: Circle Operations and Properties

```

1 // Cosine rule and triangle properties
2 double cosineRule(double a, double b, double c) {
3     return (a*a + b*b - c*c) / (2*a*b);
4 }
5
6 // Regular polygon properties
7 double regPolyArea(int n, double side) {
8     return n * side * side / (4 * tan(PI/n));
9 }
10
11 double regPolyRadius(int n, double side) {
12     return side / (2 * sin(PI/n));
13 }
14
15 struct Circle {
16     Point c; double r;
17     Circle(Point c, double r) : c(c), r(r) {}
18     bool contains(Point p) { return dist(c, p) <= r + EPS; }
19 };
20
21 // Circle from 3 points
22 Circle circumcircle(Point a, Point b, Point c) {
23     double d = 2 * (a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y));
24     double ux = ((a.x*a.x + a.y*a.y) * (b.y - c.y) + (b.x*b.x + b.y*b.y) * (c.y - a.y) + (c.x*c.x + c.y*c.y) * (a.y - b.y)) / d;
25     double uy = ((a.x*a.x + a.y*a.y) * (c.x - b.x) + (b.x*b.x + b.y*b.y) * (a.x - c.x) + (c.x*c.x + c.y*c.y) * (b.x - a.x)) / d;
26     Point center(ux, uy);
27     return Circle(center, dist(center, a));
28 }

```

76: Circle Intersection and Transformations

```
1 // Circle-circle intersection
2 int circleIntersect(Circle c1, Circle c2, Point& p1, Point& p2) {
3     double d = dist(c1.c, c2.c);
4     if (d > c1.r + c2.r || d < abs(c1.r - c2.r)) return 0;
5
6     double a = (c1.r*c1.r - c2.r*c2.r + d*d) / (2*d);
7     Point p = c1.c + (c2.c - c1.c) * (a/d);
8
9     if (abs(d - c1.r - c2.r) < EPS || abs(d - abs(c1.r - c2.r)) < EPS) {
10         p1 = p; return 1;
11     }
12
13     double h = sqrt(c1.r*c1.r - a*a);
14     Point perp = Point(-(c2.c.y - c1.c.y), c2.c.x - c1.c.x) * (h/d);
15     p1 = p + perp; p2 = p - perp;
16     return 2;
17 }
18
19 Point rotate(Point p, Point center, double angle) {
20     return center + (p - center).rotate(angle);
21 }
22
23 Point reflect(Point p, Point a, Point b) {
24     Point v = (b - a) * (1.0 / (b - a).norm());
25     Point foot = a + v * ((p - a).dot(v));
26     return foot * 2 - p;
27 }
```

10.5 3D Geometry

77: 3D Point and Vector Operations

```
1
2 struct Point3D {
3     double x, y, z;
4     Point3D(double x = 0, double y = 0, double z = 0) : x(x), y(y), z(z) {}
5
6     Point3D operator+(Point3D p) {
7         return Point3D(x + p.x, y + p.y, z + p.z);
8     }
9
10    Point3D operator-(Point3D p) {
11        return Point3D(x - p.x, y - p.y, z - p.z);
12    }
13
14    Point3D operator*(double t) {
15        return Point3D(x * t, y * t, z * t);
16    }
17
18    double dot(Point3D p) { return x * p.x + y * p.y + z * p.z; }
19    Point3D cross(Point3D p) {
20        return Point3D(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
21    }
22    double norm() { return sqrt(x*x + y*y + z*z); }
23 };
24
25 double dist3D(Point3D a, Point3D b) { return (a - b).norm(); }
26
27 // Volume of tetrahedron
28 double tetVolume(Point3D a, Point3D b, Point3D c, Point3D d) {
29     return abs((b - a).cross(c - a).dot(d - a)) / 6.0;
30 }
```