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

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.

1.1 How to Use This Reference

- **Code Templates**: Ready-to-use implementations
- **Complexity Analysis**: Time and space complexity for each algorithm
- **Usage Examples**: Practical examples and edge cases
- **Notes**: Important implementation details and optimizations

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

```cpp
// Vector initialization
vector<int> v;                    // Empty vector
vector<int> v(5);                // Size 5, initialized with 0s
vector<int> v(5, 2);             // Size 5, initialized with 2s
vector<int> v = {1, 2, 3};       // Direct initialization

// Basic operations
v.push_back(4);                  // Add element to end
v.pop_back();                    // Remove last element
v.size();                        // Get current size
v.empty();                       // Check if empty
v.front();                       // First element
v.back();                        // Last element
v.clear();                       // Remove all elements

// Access and iteration
for(int i = 0; i < v.size(); i++) {
    cout << v[i] << " ";         // Using index
}
for(int x : v) {                 // Range-based for loop
    cout << x << " ";
}
```

2: 2D Vector Operations

```cpp
// 2D vector initialization
vector<vector<int>> grid(n, vector<int>(m));     // n x m grid
vector<vector<int>> grid = {                     // Direct init
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Access elements
grid[i][j] = value;              // Set value
int value = grid[i][j];          // Get value

// Common operations
for(int i = 0; i < grid.size(); i++) {
    for(int j = 0; j < grid[i].size(); j++) {
        cout << grid[i][j] << " ";
    }
    cout << "\n";
}
```

2.1.6 Sets and Maps

3: Set and Unordered Set

```
// Set (ordered)
set<int> s;                    // Ordered unique elements
s.insert(5);                   // O(log n) insertion
s.erase(5);                   // O(log n) deletion
auto it = s.find(5);          // O(log n) search
auto it = s.lower_bound(5); // First element >= 5
auto it = s.upper_bound(5); // First element > 5

// Unordered Set (hash table)
unordered_set<int> us;        // Unordered unique elements
us.insert(5);                 // O(1) average case
us.erase(5);                 // O(1) average case
auto it = us.find(5);        // O(1) average case
```

4: Map and Unordered Map

```
// Map (ordered)
map<string, int> m;           // Key-value pairs
m["apple"] = 5;               // O(log n) insertion
m.erase("apple");            // O(log n) deletion
auto it = m.find("apple"); // O(log n) search

// Unordered Map (hash table)
unordered_map<string, int> um; // Hash table
um["apple"] = 5;                // O(1) average case
um.erase("apple");             // O(1) average case
auto it = um.find("apple");  // O(1) average case
```

5: Multiset and Multimap Operations

```
// Multiset (allows duplicates)
multiset<int> ms;
ms.insert(5);                 // Can insert multiple 5s
ms.erase(5);                 // Erases all 5s
ms.erase(ms.find(5));        // Erases one occurrence

// Multimap (multiple values per key)
multimap<string, int> mm;
mm.insert({"key", 1});
mm.insert({"key", 2});       // Same key, different value
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

```cpp
// Max heap (default)
priority_queue<int> maxHeap;
// Min heap using greater<int>
priority_queue<int, vector<int>, greater<int>> minHeap;
// Custom comparator for complex types
struct Compare {
    bool operator()(const Point& a, const Point& b) {
        // Note: reversed logic compared to set/map
        if (a.x != b.x) return a.x > b.x;
        return a.y > b.y;
    }
};
priority_queue<Point, vector<Point>, Compare> pq;
```

2.1.8 Stack and Queue

7: Stack and Queue Operations

```cpp
// Stack (LIFO)
stack<int> s;
s.push(5);                  // Add element
s.pop();                    // Remove top element
s.top();                    // Access top element
s.empty();                  // Check if empty
s.size();                   // Get size
// Queue (FIFO)
queue<int> q;
q.push(5);                  // Add element
q.pop();                    // Remove front element
q.front();                  // Access front element
q.back();                   // Access back element
q.empty();                  // Check if empty
q.size();                   // Get size
// Deque (double-ended queue)
deque<int> dq;
dq.push_front(5);           // Add to front
dq.push_back(5);            // Add to back
dq.pop_front();             // Remove from front
dq.pop_back();              // Remove from back
dq.front();                 // Access front
dq.back();                  // Access back
```

2.1.9 Bitset

Bitset provides space-efficient storage for boolean values.

8: Bitset Operations

```
// Bitset initialization
bitset<32> bs;                  // 32-bit bitset
bitset<32> bs("1010");          // From binary string
bitset<32> bs(42);              // From integer

// Basic operations
bs.set(5);                      // Set bit at position 5
bs.reset(5);                    // Reset bit at position 5
bs.flip(5);                     // Flip bit at position 5
bs.test(5);                     // Check if bit is set
bs.count();                     // Count set bits
bs.size();                      // Total number of bits

// Bitwise operations
bitset<32> a("1010"), b("1100");
auto c = a & b;                 // AND
auto d = a | b;                 // OR
auto e = a ^ b;                 // XOR
auto f = ~a;                    // NOT

// Useful for competitive programming
bs.set();                       // Set all bits
bs.reset();                     // Reset all bits
bs.flip();                      // Flip all bits
```

2.2 Advanced Data Structures

2.2.1 Segment Tree (Iterative)

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

9: Segment Tree for Range Sum

```cpp
struct SegmentTree {
    int n;
    vector<int> tree;

    SegmentTree(const vector<int>& v) {
        n = v.size();
        tree.resize(n << 1);
        for (int i = 0; i < n; i++)
            tree[i + n] = v[i];
        for (int i = n - 1; i > 0; i--)
            tree[i] = tree[i << 1] + tree[i << 1 | 1];
    }

    void update(int pos, int value) {
        tree[pos += n] = value;
        for (pos >>= 1; pos > 0; pos >>= 1)
            tree[pos] = tree[pos << 1] + tree[pos << 1 | 1];
    }

    int query(int l, int r) { // inclusive range [l, r]
        int res = 0;
        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
            if (l & 1) res += tree[l++];
            if (r & 1) res += tree[--r];
        }
        return res;
    }
};
```

10: Segment Tree Example Usage

```cpp
int main() {
    vector<int> a = {2, 1, 5, 3, 4};
    SegmentTree st(a);

    cout << st.query(1, 3) << "\n"; // 1 + 5 + 3 = 9
    st.update(2, 0);                 // a[2] = 0
    cout << st.query(1, 3) << "\n"; // 1 + 0 + 3 = 4
}
```

11: Segment Tree for Range Maximum

```
struct SegmentTree {
    int n;
    vector<int> tree;

    SegmentTree(const vector<int>& v) {
        n = v.size();
        tree.resize(n << 1);
        for (int i = 0; i < n; i++)
            tree[i + n] = v[i];
        for (int i = n - 1; i > 0; i--)
            tree[i] = max(tree[i << 1], tree[i << 1 | 1]);
    }

    void update(int pos, int value) {
        tree[pos += n] = value;
        for (pos >>= 1; pos > 0; pos >>= 1)
            tree[pos] = max(tree[pos << 1], tree[pos << 1 | 1]);
    }

    int query(int l, int r) { // inclusive range [l, r]
        int res = INT_MIN;
        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
            if (l & 1) res = max(res, tree[l++]);
            if (r & 1) res = max(res, tree[--r]);
        }
        return res;
    }
};
```

12: Segment Tree Max Example Usage

```
int main() {
    vector<int> a = {2, 1, 5, 3, 4};
    SegmentTree st(a);

    cout << st.query(1, 3) << "\n"; // max(1, 5, 3) = 5
    st.update(2, 0);                // a[2] = 0
    cout << st.query(1, 3) << "\n"; // max(1, 0, 3) = 3
}
```

2.2.2 Disjoint Set Union (DSU)

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

13: DSU with Vector

```
struct DSU {
    vector<int> parent, size;

    DSU(int n) {
        parent.resize(n);
        size.resize(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findParent(int x) {
        if (parent[x] == x) return x;
        return parent[x] = findParent(parent[x]);
    }

    bool sameGroup(int x, int y) {
        return findParent(x) == findParent(y);
    }

    void merge(int x, int y) {
        int rootX = findParent(x);
        int rootY = findParent(y);
        if (rootX == rootY) return;
        if (size[rootX] < size[rootY]) swap(rootX, rootY);
        parent[rootY] = rootX;
        size[rootX] += size[rootY];
    }
};
```

14: DSU Example Usage

```
int main() {
    DSU dsu(10);

    dsu.merge(1, 2);
    dsu.merge(2, 3);
    dsu.merge(4, 5);

    cout << (dsu.sameGroup(1, 3)) << "\n";  // 1 (true)
    cout << (dsu.sameGroup(1, 5)) << "\n";  // 0 (false)
}
```

15: DSU with Unordered Map

```cpp
struct DSUMap {
    unordered_map<int, int> parent, size;

    void makeSet(int x) {
        if (!parent.count(x)) {
            parent[x] = x;
            size[x] = 1;
        }
    }

    int findParent(int x) {
        makeSet(x);
        if (parent[x] == x) return x;
        return parent[x] = findParent(parent[x]);
    }

    bool sameGroup(int x, int y) {
        return findParent(x) == findParent(y);
    }

    void merge(int x, int y) {
        int rootX = findParent(x);
        int rootY = findParent(y);
        if (rootX == rootY) return;
        if (size[rootX] < size[rootY]) swap(rootX, rootY);
        parent[rootY] = rootX;
        size[rootX] += size[rootY];
    }
};
```

16: DSU Map Example Usage

```cpp
int main() {
    DSUMap dsu;
    dsu.merge(100, 200);
    dsu.merge(200, 300);
    dsu.merge(400, 500);

    cout << dsu.sameGroup(100, 300) << "\n"; // 1 (true)
    cout << dsu.sameGroup(100, 500) << "\n"; // 0 (false)
}
```

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.

17: DFS Implementation

```
vector<vector<int>> graph;  // Adjacency list
vector<bool> visited;

void dfs(int node) {
    visited[node] = true;
    cout << node << " ";  // Process node

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

// Initialize and run DFS
void runDFS(int start, int n) {
    graph.resize(n);
    visited.resize(n, false);
    dfs(start);
}
```

3.1.1 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

18: DFS with Connected Components

```cpp
vector<vector<int>> graph;
vector<bool> visited;

void dfs(int node) {
    visited[node] = true;

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

int countComponents(int n) {
    visited.resize(n, false);
    int components = 0;

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i);
            components++;
        }
    }
    return components;
}
```

3.1.2 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.

19: BFS Implementation

```cpp
vector<vector<int>> graph;  // Adjacency list
vector<bool> visited;

void bfs(int start) {
    queue<int> q;
    q.push(start);
    visited[start] = true;

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

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

// Initialize and run BFS
void runBFS(int start, int n) {
    graph.resize(n);
    visited.resize(n, false);
    bfs(start);
}
```

3.2.1 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

20: BFS with Distance Calculation

```
vector<vector<int>> graph;
vector<int> distance;

void bfsWithDistance(int start, int n) {
    queue<int> q;
    distance.resize(n, -1);

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

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

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

3.2.2 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.

21: Dijkstra's Algorithm

```cpp
vector<vector<pair<int, int>>> graph;  // {neighbor, weight}
vector<int> distance;

void dijkstra(int start, int n) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
        int>>> pq;
    distance.resize(n, INT_MAX);

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

    while (!pq.empty()) {
        int dist = pq.top().first;
        int node = pq.top().second;
        pq.pop();

        if (dist > distance[node]) continue;

        for (auto [neighbor, weight] : graph[node]) {
            if (distance[node] + weight < distance[neighbor]) {
                distance[neighbor] = distance[node] + weight;
                pq.push({distance[neighbor], neighbor});
            }
        }
    }
}
```

3.3.1 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

22: Dijkstra with Path Reconstruction

```
vector<vector<pair<int, int>>> graph;
vector<int> distance, parent;

void dijkstraWithPath(int start, int n) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
        int>>> pq;
    distance.resize(n, INT_MAX);
    parent.resize(n, -1);

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

    while (!pq.empty()) {
        int dist = pq.top().first;
        int node = pq.top().second;
        pq.pop();

        if (dist > distance[node]) continue;

        for (auto [neighbor, weight] : graph[node]) {
            if (distance[node] + weight < distance[neighbor]) {
                distance[neighbor] = distance[node] + weight;
                parent[neighbor] = node;
                pq.push({distance[neighbor], neighbor});
            }
        }
    }
}

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

3.3.2 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.

23: Floyd-Warshall Algorithm

```cpp
int main() {
    int INF = 1e9;
    int n = 4;
    vector<vector<int>> mat = {
        {0, 3, INF, 7},
        {8, 0, 2, INF},
        {5, INF, 0, 1},
        {2, INF, INF, 0}
    };

    for (int mid = 0; mid < n; mid++)
        for (int from = 0; from < n; from++)
            for (int to = 0; to < n; to++)
                mat[from][to] = min(mat[from][to], mat[from][mid] + mat[mid
                    ][to]);

    for (int from = 0; from < n; from++) {
        for (int to = 0; to < n; to++)
            cout << (mat[from][to] == INF ? -1 : mat[from][to]) << " ";
        cout << "\n";
    }
}
```

3.4.1 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.

24: Topological Sort with DFS

```
vector<vector<int>> graph;
vector<bool> visited;
vector<int> topoOrder;

void dfs(int node) {
    visited[node] = true;

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

    topoOrder.push_back(node);
}

vector<int> topologicalSort(int n) {
    visited.resize(n, false);
    topoOrder.clear();

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i);
        }
    }

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

3.5.1 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

25: Topological Sort with Kahn's Algorithm

```
vector<vector<int>> graph;
vector<int> inDegree;

vector<int> kahnTopologicalSort(int n) {
    queue<int> q;
    vector<int> result;

    // Calculate in-degrees
    inDegree.resize(n, 0);
    for (int i = 0; i < n; i++) {
        for (int neighbor : graph[i]) {
            inDegree[neighbor]++;
        }
    }

    // Add nodes with in-degree 0
    for (int i = 0; i < n; i++) {
        if (inDegree[i] == 0) {
            q.push(i);
        }
    }

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        result.push_back(node);

        for (int neighbor : graph[node]) {
            inDegree[neighbor]--;
            if (inDegree[neighbor] == 0) {
                q.push(neighbor);
            }
        }
    }

    return result;
}
```

3.5.2 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.

26: Cycle Detection in Undirected Graph

```
vector<vector<int>> graph;
vector<bool> visited;

bool hasCycleUndirected(int node, int parent) {
    visited[node] = true;

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

bool detectCycleUndirected(int n) {
    visited.resize(n, false);

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            if (hasCycleUndirected(i, -1)) {
                return true;
            }
        }
    }
    return false;
}
```

3.6.1 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

27: Cycle Detection in Directed Graph

```
vector<vector<int>> graph;
vector<bool> visited, recStack;

bool hasCycleDirected(int node) {
    visited[node] = true;
    recStack[node] = true;

    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            if (hasCycleDirected(neighbor)) {
                return true;
            }
        } else if (recStack[neighbor]) {
            return true;
        }
    }

    recStack[node] = false;
    return false;
}

bool detectCycleDirected(int n) {
    visited.resize(n, false);
    recStack.resize(n, false);

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

3.6.2 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.

28: LIS - 2D DP Bottom-Up Implementation

```cpp
int lengthOfLIS(vector<int>& nums) {
    int n = nums.size();
    vector<vector<int>> dp(n + 2, vector<int>(n + 2));

    for (int i = n - 1; i >= 0; --i) {
        for (int j = i - 1; j >= -1; --j) {
            int curr = i + 1, prev = j + 1;
            if (j == -1 || nums[i] > nums[j])
                dp[curr][prev] = dp[curr + 1][curr] + 1;
            dp[curr][prev] = max(dp[curr][prev], dp[curr + 1][prev]);
        }
    }

    // Reconstruct the LIS
    vector<int> lis;
    int i = 0, j = -1;
    while (i < n) {
        int curr = i + 1, prev = j + 1;
        if (dp[curr][prev] == dp[curr + 1][curr] + 1 && (j == -1 || nums[i] > nums[j])) {
            lis.push_back(nums[i]);
            j = i;
        }
        i++;
    }

    return dp[1][0];
}
```

4.1.1 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

29: LIS - 1D DP Bottom-Up Implementation

```cpp
int lengthOfLIS(vector<int>& nums) {
    int n = nums.size();
    vector<int> dp(n, 1);
    for (int i = n - 1; i >= 0; --i)
        for (int j = i + 1; j < n; ++j)
            if (nums[j] > nums[i])
                dp[i] = max(dp[i], dp[j] + 1);

    // Reconstruct the LIS
    int maxLen = *max_element(dp.begin(), dp.end());
    vector<int> lis;
    for (int i = 0; i < n && maxLen; ++i)
        if (dp[i] == maxLen) {
            lis.push_back(nums[i]);
            --maxLen;
        }

    return *max_element(dp.begin(), dp.end());
}
```

4.1.2 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

30: LIS - Recursive Implementation

```
int lengthOfLIS(const vector<int>& nums) {
    int n = nums.size();
    vector<vector<int>> dp(n + 1, vector<int>(n + 1, -1));

    function<int(int, int)> calculateLIS = [&](int cur, int prev) {
        if (cur == n) return 0;
        int i = cur + 1, j = prev + 1;
        int& res = dp[i][j];
        if (res != -1) return res;

        if (prev == -1 || nums[cur] > nums[prev])
            res = max(res, 1 + calculateLIS(cur + 1, cur));

        res = max(res, calculateLIS(cur + 1, prev));

        return res;
    };

    return calculateLIS(0, -1);
}
```

4.1.3 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

31: LIS - Binary Search Implementation

```
int lengthOfLIS(const vector<int>& a) {
    vector<int> lis;
    for (int i = 0; i < a.size(); ++i) {
        auto it = lower_bound(begin(lis), end(lis), a[i]);
        it != end(lis) ? *it = a[i] : lis.push_back(a[i]);
    }
    return lis.size();
}

// Reconstruct the actual LIS sequence
vector<int> getLIS(const vector<int>& a) {
    vector<int> lis, prev(a.size(), -1);
    for (int i = 0; i < a.size(); ++i) {
        auto it = lower_bound(begin(lis), end(lis), i, [&](int j, int k) {
            return a[j] < a[k];
        });
        it != end(lis) ? *it = i : lis.push_back(i);
        if (it != begin(lis)) prev[i] = *(it - 1);
    }
    vector<int> res;
    for (int i = lis.back(); i != -1; i = prev[i]) {
        res.push_back(a[i]);
    }
    reverse(begin(res), end(res));
    return res;
}
```

4.1.4 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

32: LIS - Segment Tree Implementation

```
struct SegmentTree {
    int n;
    vector<int> tree;

    SegmentTree(int _n) {
        n = _n;
        tree.resize(2 * _n);
    }

    void update(int pos, int value) {
        tree[pos += n] = value;
        for (pos >>= 1; pos > 0; pos >>= 1)
            tree[pos] = max(tree[pos << 1], tree[pos << 1 | 1]);
    }

    int query(int l, int r) {
        int res = 0;
        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
            if (l & 1) res = max(res, tree[l++]);
            if (r & 1) res = max(res, tree[--r]);
        }
        return res;
    }
};

int lengthOfLIS(vector<int>& nums) {
    SegmentTree seg(1e5 + 1);
    int res = 0;
    for (auto i : nums) {
        i += 2e4;  // Offset to handle negative numbers
        int val = seg.query(0, i - 1) + 1;  // Find max LIS ending before i
        res = max(res, val);
        seg.update(i, val);  // Update the LIS at position i
    }
    return res;
}
```

4.1.5 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.

33: Subsets Implementation

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

vector<vector<int>> subsets(vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> subset;

    function<void(int)> generate = [&](int start) {
        // Add the current subset to the result
        result.push_back(subset);

        // Try adding each remaining element to the current subset
        for (int i = start; i < nums.size(); i++) {
            subset.push_back(nums[i]);
            generate(i + 1);
            subset.pop_back();
        }
    };

    generate(0);
    return result;
}
```

5.1.1 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 → Recurse → 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.

34: Permutations Without Duplicates

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

vector<vector<int>> permuteUnique(vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> comb;
    vector<bool> visited(nums.size(), false);

    function<void()> permute = [&]() {
        if (comb.size() == nums.size()) {
            result.push_back(comb);
            return;
        }
        for (int i = 0; i < nums.size(); i++) {
            if (visited[i]) continue;
            visited[i] = true;
            comb.push_back(nums[i]);
            permute();
            comb.pop_back();
            visited[i] = false;
        }
    };

    permute();
    return result;
}
```

5.2.1 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

35: Permutations With Duplicates

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

vector<vector<int>> permuteWithDuplicates(vector<int>& nums) {
    vector<vector<int>> result;
    unordered_map<int, int> counter;
    for (int num : nums) counter[num]++;

    vector<int> comb;

    function<void()> permute = [&]() {
        if (comb.size() == nums.size()) {
            result.push_back(comb);
            return;
        }
        for (auto& item : counter) {
            int num = item.first;
            int count = item.second;
            if (count == 0) continue;
            comb.push_back(num);
            counter[num]--;
            permute();
            comb.pop_back();
            counter[num]++;
        }
    };

    permute();
    return result;
}
```

5.2.2 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.

36: Combinations Implementation

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

vector<vector<int>> combinations(vector<int>& nums, int k) {
    vector<vector<int>> result;
    vector<int> comb;

    function<void(int)> combine = [&](int start) {
        if (comb.size() == k) {
            result.push_back(comb);
            return;
        }
        for (int i = start; i < nums.size(); i++) {
            comb.push_back(nums[i]);
            combine(i + 1);
            comb.pop_back();
        }
    };

    combine(0);
    return result;
}
```

5.3.1 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.

37: STL String Functions

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

string s = "Hello World";
// Basic operations
s.length();                    // Get string length
s.size();                      // Same as length()
s.empty();                     // Check if empty
s.clear();                     // Clear string
// Access elements
s[0];                         // Access character
s.at(0);                      // Bounds-checked access
s.front();                     // First character
s.back();                      // Last character
// String manipulation
s.substr(0, 5);               // Substring
s.find("World");               // Find substring
s.replace(0, 5, "Hi");        // Replace substring
s.insert(5, " ");             // Insert at position
// String algorithms
reverse(s.begin(), s.end());   // Reverse string
sort(s.begin(), s.end());      // Sort characters
transform(s.begin(), s.end(), s.begin(), ::tolower); // To lowercase
transform(s.begin(), s.end(), s.begin(), ::toupper); // To uppercase
// String concatenation
string s1 = "Hello";
string s2 = "World";
string result = s1 + " " + s2; // Concatenation
s1.append(s2);                 // Append to string
s1 += s2;                      // Append operator
```

6.1.1 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.

38: Longest Substring Without Repeating Characters

```cpp
int lengthOfLongestSubstring(string s) {
    vector<int> charIndex(128, -1);  // ASCII characters
    int maxLength = 0;
    int start = 0;

    for (int end = 0; end < s.length(); end++) {
        char currentChar = s[end];

        // If character already seen, update start
        if (charIndex[currentChar] >= start) {
            start = charIndex[currentChar] + 1;
        }

        charIndex[currentChar] = end;
        maxLength = max(maxLength, end - start + 1);
    }

    return maxLength;
}
```

6.2.1 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.

39: Trie Node Implementation

```
struct TrieNode {
    vector<TrieNode*> children;
    bool isEndOfWord;

    TrieNode() {
        children.resize(26, nullptr);
        isEndOfWord = false;
    }
};
```

6.3.1 Trie Node Notes

- **Time Complexity**: O(1) for insertion and search
- **Space Complexity**: O(ALPHABET_SIZE $\times$ N $\times$ M)
- **Applications**: Prefix matching, autocomplete

40: Trie Implementation

```

class Trie {
private:
    TrieNode* root;

public:
    Trie() { root = new TrieNode(); }
    void insert(string word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index])  node->children[index] = new
                TrieNode();
            node = node->children[index];
        }
        node->isEndOfWord = true;
    }
    bool search(string word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index])  return false;
            node = node->children[index];
        }
        return node->isEndOfWord;
    }
    bool startsWith(string prefix) {
        TrieNode* node = root;
        for (char c : prefix) {
            int index = c - 'a';
            if (!node->children[index])  return false;
            node = node->children[index];
        }
        return true;
    }
};
```

6.3.2 Trie Notes

- **Time Complexity**: O(m) where m is string length
- **Space Complexity**: O(ALPHABET_SIZE $\times$ N $\times$ M)
- **Applications**: Prefix matching, autocomplete
- **Memory Usage**: Can be memory intensive for large datasets

7 Mathematics

7.1 Fast Power (Binary Exponentiation)

Efficiently compute large powers using binary exponentiation.

41: Binary Exponentiation - Iterative

```
int64_t power(int64_t base, int64_t exp) {
    int64_t result = 1;
    while (exp > 0) {
        if (exp & 1) result *= base;
        base *= base;
        exp >>= 1;
    }
    return result;
}
```

42: Modular Exponentiation

```
int64_t modPower(int64_t base, int64_t exp, int64_t mod) {
    int64_t result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp & 1) result = (result * base) % mod;
        base = (base * base) % mod;
        exp >>= 1;
    }
    return result;
}
```

7.1.1 Modular Exponentiation Notes

- **Time Complexity**: O(log 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.

43: GCD and LCM Functions

```
int gcd(int a, int b) {
    while (b != 0) {
        a %= b;
        swap(a, b);
    }
    return a;
}

int lcm(int a, int b) {
    return (a / gcd(a, b)) * b;
}
```

7.2.1 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**: LCM(a,b) = (a $\times$ b) / GCD(a,b)
- **Applications**: Number theory, fraction simplification

7.3 Combinatorics

Basic combinatorial functions with modular arithmetic support.

44: Standard nCr and nPr

```
// Don't use for n > 67 (int64_t overflow)
int64_t nCr(int n, int r) {
    if (r < 0 || r > n) return 0;
    if (r > n - r) r = n - r;
    int64_t res = 1;
    for (int i = 0; i < r; ++i) {
        res *= (n - i);
        res /= (i + 1);
    }
    return res;
}

// Don't use for n > 20 or large r (int64_t overflow)
int64_t nPr(int n, int r) {
    if (r < 0 || r > n) return 0;
    int64_t res = 1;
    for (int i = 0; i < r; ++i)
        res *= (n - i);
    return res;
}
```

7.3.1 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

45: Combinatorics with Modular Arithmetic

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

class Combinatorics {
private:
    static const int MOD = 1000000007;
    vector<int64_t> f, inv;

    int64_t pow(int64_t b, int64_t e) const {
        int64_t r = 1;
        while (e) {
            if (e & 1) r = r * b % MOD;
            b = b * b % MOD;
            e >>= 1;
        }
        return r;
    }

public:
    Combinatorics(int n) : f(n + 1), inv(n + 1) {
        f[0] = 1;
        for (int i = 1; i <= n; ++i)
            f[i] = f[i - 1] * i % MOD;
        inv[n] = pow(f[n], MOD - 2);
        for (int i = n - 1; i >= 0; --i)
            inv[i] = inv[i + 1] * (i + 1) % MOD;
    }
    int64_t nCr(int n, int r) const {
        if (r < 0 || r > n) return 0;
        return f[n] * inv[r] % MOD * inv[n - r] % MOD;
    }
    int64_t nPr(int n, int r) const {
        if (r < 0 || r > n) return 0;
        return f[n] * inv[n - r] % MOD;
    }
};
```

7.3.2 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.

46: Sieve of Eratosthenes

```cpp
#include <bits/stdc++.h>
using namespace std;
// Time: O(n log log n), Space: O(n)
// Range: n up to 10^7 (typical CP limit)
// Memory: ~40MB for n=10^7
class Sieve {
public:
    vector<int> prime_factor, primes;

    Sieve(int n) {
        prime_factor.resize(n + 1);
        for (int i = 0; i <= n; i++) prime_factor[i] = i;
        for (int i = 2; i <= n; i++) {
            if (prime_factor[i] == i) {
                primes.push_back(i);
                for (int j = i * i; j <= n; j += i)
                    if (prime_factor[j] == j) prime_factor[j] = i;
            }
        }
    }
};

int main() {
    Sieve sieve(100);
    for (int p : sieve.primes) cout << p << " ";
    cout << "\n";
    for (int i = 12; i <= 15; i++) {
        cout << i << ": prime_factor=" << sieve.prime_factor[i] << "\n";
    }
    return 0;
}
```

7.4.1 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.

47: Binary to Decimal Conversion

```
// Convert binary string to decimal integer
string binaryStr = "1010";
int decimal = stoll(binaryStr, nullptr, 2);
// Result: 10

// Using bitset for larger binary strings
#include <bitset>
const int N = 32; // Enough for standard integers
int decimal = bitset<N>("1010").to_ulong();
// Result: 10

// For longer binary strings
const int LARGE_N = 10000; // For very large binary strings
unsigned long long largeDecimal = bitset<LARGE_N>(longBinaryStr).to_ullong()
    ;
```

8.1.1 Binary to Decimal Notes

- **stoll Method**: Limited to 64-bit integers
- **bitset Method**: Can handle larger binary strings
- **Time Complexity**: O(n) where n is binary string length
- **Applications**: Binary arithmetic, bit manipulation

48: Decimal to Binary Conversion

```
#include <bitset>

// Convert decimal to binary string
int decimal = 10;
const int N = 8; // Number of bits to represent
string binaryStr = bitset<N>(decimal).to_string();
// Result: "00001010"

// Remove leading zeros if needed
binaryStr = binaryStr.substr(binaryStr.find('1') != string::npos ? binaryStr
    .find('1') : N-1);
// Result: "1010"

// Using std::format (C++20)
#include <format>
string binaryStr = format("{:b}", decimal);
// Result: "1010"
```

8.1.2 Decimal to Binary Notes

- **bitset Method**: Most reliable for standard integers
- **format Method**: Clean C++20 approach
- **Leading Zeros**: Need manual handling for clean output
- **Applications**: Binary representation, bit manipulation

49: Coordinate Compression Template

```cpp
template <typename T>
class Compress {
    vector<T> vals;
    unordered_map<T, int> idx;

public:
    Compress(const vector<T>& input) {
        vals = input;
        sort(vals.begin(), vals.end());
        vals.erase(unique(vals.begin(), vals.end()), vals.end());
        for (int i = 0; i < vals.size(); i++)
            idx[vals[i]] = i;
    }

    int operator[](const T& x) const { return idx.at(x); }
    T orig(int i) const { return vals.at(i); }
    int size() const { return vals.size(); }
};
```

50: Coordinate Compression Example

```cpp
// Basic usage
vector<int> data = {1000000, 5, 10000, 6, 7, 1000};
Compress<int> comp(data);

// Convert original value to compressed index
for (int x : data) {
    cout << x << " -> " << comp[x] << endl;
}
// Output: 1000000->5, 5->0, 10000->3, 6->1, 7->2, 1000->4

// Get original value from compressed index
for (int i = 0; i < comp.size(); i++) {
    cout << i << " -> " << comp.orig(i) << endl;
}
// Output: 0->5, 1->6, 2->7, 3->1000, 4->10000, 5->1000000
```

8.1.3 Coordinate Compression Notes

- **Time Complexity**: O(N log N) for construction, O(1) for lookup
- **Space Complexity**: O(N) for sorted list and hashmap
- **Applications**: Segment trees, large value ranges, sparse data
- **Features**: Preserves relative ordering, bidirectional mapping

51: Measure Time Utility

```
#include <iostream>
#include <chrono>
#include <cstdint>
#include <iomanip>
using namespace std;

template<typename Func, typename... Args>
double measure(Func&& f, Args&&... args) {
    auto start = chrono::high_resolution_clock::now();
    forward<Func>(f)(forward<Args>(args)...);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> elapsed = end - start;
    return elapsed.count();
}

int main() {
    cout << fixed << setprecision(4);

    double t1 = measure(funcVoid);
    cout << "funcVoid took " << t1 << " ms\n";

    int64_t res = 0;
    auto wrapper = [&](int n) { res = funcInt(n); };
    double t2 = measure(wrapper, 1000000);
    cout << "funcInt took " << t2 << " ms, sum = " << res << "\n";

    return 0;
}
```

8.1.4 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

52: Random Number Generator

```
#include <iostream>
#include <random>
#include <ctime>
using namespace std;

mt19937_64 ran(time(nullptr));

int r(int a, int b) {
    return ran() % (abs(b - a) + 1) + min(a, b);
}
```

53: Random Number Generator Example Usage

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

8.1.5 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

54: String Split Utility

```
template<typename T>
vector<T> split(const string& line, char delimiter = ' ') {
    vector<T> result;
    stringstream ss(line);
    string token;

    while (getline(ss, token, delimiter)) {
        stringstream convert(token);
        T value;
        convert >> value;
        if (!convert.fail()) {
            result.push_back(value);
        }
    }

    return result;
}

// Basic string split to vector<string>
vector<string> split(const string& line, char delimiter = ' ') {
    vector<string> result;
    stringstream ss(line);
    string token;
    while (getline(ss, token, delimiter)) {
        result.push_back(token);
    }
    return result;
}
```

55: String Split Examples

```
// Split string into integers (default delimiter is space)
string line = "10 20 30";
vector<int> ints = split<int>(line);
// Result: [10, 20, 30]

// Split string into doubles (specify delimiter)
string line2 = "3.14,2.71,1.41";
vector<double> doubles = split<double>(line2, ',');
// Result: [3.14, 2.71, 1.41]

// Read a line from input and split it
string input;
getline(cin, input);
vector<int> values = split<int>(input);
// Now you can use the values vector as needed
```