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

```
// 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

```
// 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

```
// 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

```
// 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

```cpp
// 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

```
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

```
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

```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] = 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

```cpp
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

```
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

```
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

```
vector<vector<int>> dist;  // Distance matrix
int n;

void floydWarshall() {
    // Initialize distance matrix
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) dist[i][j] = 0;
            else dist[i][j] = INT_MAX;
        }
    }

    // Add edges
    // dist[u][v] = weight;  // Add your edges here

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

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

```cpp
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

3.7 Important Notes

3.7.1 Graph Representation

- **Adjacency List**: Space O(V + E), good for sparse graphs
- **Adjacency Matrix**: Space $O(V^2)$, good for dense graphs
- **Edge List**: Space O(E), useful for some algorithms

3.7.2 Algorithm Complexities

- **DFS/BFS**: Time O(V + E), Space O(V)
- **Dijkstra**: Time O((V + E) log V), Space O(V)
- **Floyd-Warshall**: Time $O(V^3)$, Space $O(V^2)$
- **Topological Sort**: Time O(V + E), Space O(V)
- **Cycle Detection**: Time O(V + E), Space O(V)

3.7.3 Usage Tips

- Use DFS for exploring all possibilities, backtracking problems
- Use BFS for shortest path in unweighted graphs, level-order traversal
- Use Dijkstra for shortest path in weighted graphs with positive weights
- Use Floyd-Warshall for all-pairs shortest path or detecting negative cycles
- Use Topological Sort for dependency resolution, build order problems
- Use Cycle Detection for validating DAGs, detecting deadlocks