C++ Data Structures & Algorithms Cheatsheet

Problem Pattern Recognition

When you see...	Consider using...
"Sorted array/list"	Binary Search, Two Pointers
"Find minimum/maximum"	Heap, Binary Search (if sorted)
"Frequency counting"	Hash Map/Unordered Map
"Valid parentheses/brackets"	Stack
"Shortest/longest path in graph"	BFS/Dijkstra's (shortest), DFS (longest)
"Top K elements"	Heap (priority_queue)
"Find all combinations/permutations"	Backtracking
"Dynamic programming keywords: optimal, maximum profit, minimum cost"	DP (bottom-up or top-down)
"Intervals/overlaps"	Sorting + Greedy
"Substring problems"	Sliding Window, Two Pointers
"Tree traversal"	DFS, BFS

Data Structures Guide

Arrays & Vectors

cpp

```cpp
vector<int> v = {1, 2, 3};
v.push_back(4);          // Add element
v.pop_back();            // Remove last element
v.size();                // Get size
v[i];                    // Access element (no bounds checking)
v.at(i);                 // Access with bounds checking
sort(v.begin(), v.end()); // Sort
```

Best for:

- Sequential access
- Constant-time random access
- When size is known or changes infrequently

Linked List

cpp

```cpp
// Using STL list
list<int> ll = {1, 2, 3};
ll.push_back(4);    // Add to end
ll.push_front(0);   // Add to front
ll.pop_back();      // Remove from end
ll.pop_front();     // Remove from front

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

Best for:

- Frequent insertions/deletions
- When memory needs to be allocated dynamically
- Implementation of other data structures (stacks, queues)

Stack

cpp

```cpp
stack<int> s;
s.push(1);          // Add element
s.pop();            // Remove top element
s.top();            // Access top element
s.empty();          // Check if empty
s.size();           // Get size
```

Best for:

- LIFO (Last In First Out) operations
- Parentheses/bracket matching
- Expression evaluation/parsing
- Function call tracking/execution
- Backtracking algorithms

Queue

cpp

```cpp
queue<int> q;
q.push(1);          // 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
```

Best for:

- FIFO (First In First Out) operations
- BFS (Breadth-First Search)
- Task scheduling
- Print queue implementation

Priority Queue (Heap)

cpp

```cpp
// Max heap (default)
priority_queue<int> maxHeap;
// Min heap
priority_queue<int, vector<int>, greater<int>> minHeap;

maxHeap.push(1);    // Add element
maxHeap.top();      // Get max element
maxHeap.pop();      // Remove max element
maxHeap.empty();    // Check if empty
maxHeap.size();     // Get size
```

Best for:

- Finding k-th largest/smallest elements
- Heap sort
- Dijkstra's algorithm
- Task scheduling with priorities
- Median maintenance

Hash Map/Set

cpp

```cpp
// Hash Map
unordered_map<string, int> map;
map["key"] = 1;       // Insert/update
map.count("key");     // Check if key exists
map.erase("key");     // Remove element

// Hash Set
unordered_set<int> set;
set.insert(1);        // Insert element
set.count(1);         // Check if element exists
set.erase(1);         // Remove element
```

Best for:

- Fast lookups (average O(1))
- Frequency counting
- De-duplication
- Caching
- Two-sum type problems

Map/Set (Binary Search Tree)

cpp

```cpp
// Ordered Map
map<string, int> orderedMap;
orderedMap["key"] = 1;  // Insert/update
orderedMap.lower_bound("key"); // Find key >= given key
orderedMap.upper_bound("key"); // Find key > given key

// Ordered Set
set<int> orderedSet;
orderedSet.insert(1);   // Insert element
auto it = orderedSet.lower_bound(1); // Find element >= given element
```

Best for:

- When ordering is important
- Range queries
- Finding ceiling/floor elements
- Maintaining sorted data

Graph Representations

```cpp
// Adjacency List
vector<vector<int>> adjList(n);
adjList[0].push_back(1);  // Edge from 0 to 1

// Adjacency Matrix
vector<vector<int>> adjMatrix(n, vector<int>(n, 0));
adjMatrix[0][1] = 1;      // Edge from 0 to 1

// Edge List
vector<pair<int, int>> edges;
edges.push_back({0, 1});  // Edge from 0 to 1
```

Best for:

- Network modeling
- Path finding
- Connectivity analysis
- Social networks
- Web crawling

Trie (Prefix Tree)

```cpp
struct TrieNode {
    TrieNode* children[26] = {};
    bool isEnd = false;
};

// Insert word
void insert(TrieNode* root, string word) {
    TrieNode* node = root;
    for (char c : word) {
        int idx = c - 'a';
        if (!node->children[idx])
            node->children[idx] = new TrieNode();
        node = node->children[idx];
    }
    node->isEnd = true;
}

// Search word
bool search(TrieNode* root, string word) {
    TrieNode* node = root;
    for (char c : word) {
        int idx = c - 'a';
        if (!node->children[idx]) return false;
        node = node->children[idx];
    }
    return node->isEnd;
}
```

Best for:

- Prefix matching
- Auto-complete
- Spell checking
- IP routing
- Word games

Union-Find (Disjoint Set)

cpp

```cpp
class UnionFind {
    vector<int> parent, rank;
public:
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++) parent[i] = i;
    }

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

    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX == rootY) return;

        if (rank[rootX] < rank[rootY])
            parent[rootX] = rootY;
        else {
            parent[rootY] = rootX;
            if (rank[rootX] == rank[rootY])
                rank[rootX]++;
        }
    }
};
```

Best for:

- Detecting cycles in undirected graphs
- Finding connected components
- Minimum spanning tree algorithms (Kruskal's)
- Network connectivity problems

Common Algorithms

Sorting

cpp

```cpp
// Quicksort (using STL)
sort(arr.begin(), arr.end());

// Custom comparator
sort(arr.begin(), arr.end(), [](int a, int b) {
    return a > b;  // Sort in descending order
});

// Partial sort (for top K elements)
partial_sort(arr.begin(), arr.begin() + k, arr.end());

// Stable sort
stable_sort(arr.begin(), arr.end());
```

When to use:

- Need to arrange elements in specific order
- Preparing data for binary search
- Greedy algorithms
- Merge overlapping intervals

Binary Search

cpp

```cpp
// Using STL
auto it = lower_bound(arr.begin(), arr.end(), target);  // First element >= target
auto it = upper_bound(arr.begin(), arr.end(), target);  // First element > target
bool found = binary_search(arr.begin(), arr.end(), target);

// Manual implementation
int binarySearch(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        if (nums[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

When to use:

- Searching in sorted arrays
- Finding insertion points
- Finding first/last occurrence
- Optimizing by reducing search space
- Search space is monotonic

Two Pointers

cpp

```cpp
// Example: Find pair that sums to target in sorted array
bool findPair(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) return true;
        if (sum < target) left++;
        else right--;
    }
    return false;
}
```

When to use:

- Working with sorted arrays
- Finding pairs with certain constraints
- Palindrome problems
- Container with most water
- Remove duplicates

Sliding Window

cpp

```cpp
// Fixed window example: Maximum sum subarray of size k
int maxSumSubarray(vector<int>& nums, int k) {
    int maxSum = 0, windowSum = 0;
    for (int i = 0; i < nums.size(); i++) {
        windowSum += nums[i];
        if (i >= k - 1) {
            maxSum = max(maxSum, windowSum);
            windowSum -= nums[i - (k - 1)];
        }
    }
    return maxSum;
}

// Variable window example: Smallest subarray with sum >= target
int smallestSubarray(vector<int>& nums, int target) {
    int minLen = INT_MAX, windowSum = 0;
    int left = 0;
    for (int right = 0; right < nums.size(); right++) {
        windowSum += nums[right];
        while (windowSum >= target) {
            minLen = min(minLen, right - left + 1);
            windowSum -= nums[left++];
        }
    }
    return minLen == INT_MAX ? 0 : minLen;
}
```

When to use:

- Substring/subarray problems
- Maximum/minimum substring with constraints
- Finding permutations in a string
- String matching problems
- Stream processing

Depth-First Search (DFS)

```cpp
// Recursive DFS
void dfs(vector<vector<int>>& graph, int node, vector<bool>& visited) {
    if (visited[node]) return;
    visited[node] = true;
    // Process node
    for (int neighbor : graph[node]) {
        dfs(graph, neighbor, visited);
    }
}

// Iterative DFS using stack
void dfsIterative(vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false);
    stack<int> s;
    s.push(start);

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

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

        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                s.push(neighbor);
            }
        }
    }
}
```

When to use:

- Tree/graph traversal
- Finding paths
- Detecting cycles
- Topological sorting
- Connected components
- Maze problems

Breadth-First Search (BFS)

cpp

```cpp
void bfs(vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false);
    queue<int> q;
    q.push(start);
    visited[start] = true;

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

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

When to use:

- Shortest path in unweighted graphs
- Level-order traversal
- Finding connected components
- Word ladder problems
- Minimum steps to reach target

Backtracking

cpp

```cpp
// Example: Generate all subsets
void generateSubsets(vector<int>& nums, vector<vector<int>>& result, vector<int>& current, int
    result.push_back(current);

    for (int i = index; i < nums.size(); i++) {
        current.push_back(nums[i]);
        generateSubsets(nums, result, current, i + 1);
        current.pop_back(); // Backtrack
    }
}
```

When to use:

- Generating all possible combinations/permutations
- Puzzle solving (Sudoku, N-Queens)
- Path finding
- Constraint satisfaction problems
- When you need to explore all possibilities

Dynamic Programming

```cpp
// Top-down (Memoization)
int fibMemo(int n, vector<int>& memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo);
    return memo[n];
}

// Bottom-up (Tabulation)
int fibTab(int n) {
    if (n <= 1) return n;
    vector<int> dp(n+1);
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

When to use:

- Optimization problems
- Counting problems
- When you can express solution in terms of subproblems
- When subproblems overlap
- Fibonacci, knapsack, LCS, edit distance problems

Greedy Algorithms

```cpp
// Example: Activity selection
vector<pair<int, int>> selectActivities(vector<pair<int, int>>& activities) {
    // Sort by end time
    sort(activities.begin(), activities.end(), [](auto& a, auto& b) {
        return a.second < b.second;
    });

    vector<pair<int, int>> selected;
    selected.push_back(activities[0]);
    int lastEnd = activities[0].second;

    for (int i = 1; i < activities.size(); i++) {
        if (activities[i].first >= lastEnd) {
            selected.push_back(activities[i]);
            lastEnd = activities[i].second;
        }
    }
    return selected;
}
```

When to use:

- When local optimal choice leads to global optimal solution
- Interval scheduling
- Huffman coding
- Fractional knapsack
- Dijkstra's algorithm

Divide & Conquer

cpp

```cpp
// Example: Merge Sort
void mergeSort(vector<int>& nums, int left, int right) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;
    mergeSort(nums, left, mid);
    mergeSort(nums, mid + 1, right);
    merge(nums, left, mid, right);
}

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

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

    while (i <= mid) temp[k++] = nums[i++];
    while (j <= right) temp[k++] = nums[j++];

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

When to use:

- Problems that can be broken into similar subproblems
- Merge sort
- Quick sort
- Binary search
- Strassen's matrix multiplication

Common Problem Patterns and Solutions

1. Two Sum

- **Problem**: Find two numbers that add up to a target
- **Solution**:
  - Hash map to store value-index pairs
  - Time: O(n), Space: O(n)

cpp

```cpp
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> map;
    for (int i = 0; i < nums.size(); i++) {
        int complement = target - nums[i];
        if (map.count(complement))
            return {map[complement], i};
        map[nums[i]] = i;
    }
    return {};
}
```

2. Binary Search Variations

- **Finding first/last occurrence**:

cpp

```cpp
int findFirst(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1, result = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            result = mid;
            right = mid - 1; // Continue searching left
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return result;
}
```

3. Sliding Window for Substring

- **Problem**: Find smallest substring containing all characters
- **Solution**:
    - Track character frequencies with hash map
    - Use sliding window to minimize substring length
    - Time: O(n), Space: O(k) where k is character set size

4. Island (Connected Components) Problem

- **Problem**: Count number of islands in a grid

- **Solution**:
  - DFS or BFS from each unvisited land cell
  - Mark visited cells to avoid double counting
  - Time: O(m*n), Space: O(m*n)

cpp

```cpp
int numIslands(vector<vector<char>>& grid) {
    if (grid.empty()) return 0;
    int m = grid.size(), n = grid[0].size(), islands = 0;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == '1') {
                islands++;
                dfs(grid, i, j);
            }
        }
    }
    return islands;
}

void dfs(vector<vector<char>>& grid, int i, int j) {
    int m = grid.size(), n = grid[0].size();
    if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == '0')
        return;

    grid[i][j] = '0'; // Mark as visited
    dfs(grid, i+1, j);
    dfs(grid, i-1, j);
    dfs(grid, i, j+1);
    dfs(grid, i, j-1);
}
```

5. LeetCode Top 75 Problem Patterns

1. **Arrays & Hashing**
   - Two Sum, Group Anagrams, Top K Frequent Elements
2. **Two Pointers**
   - Valid Palindrome, 3Sum, Container With Most Water
3. **Sliding Window**
   - Best Time to Buy/Sell Stock, Longest Substring Without Repeating Characters
4. **Stack**

- Valid Parentheses, Min Stack, Daily Temperatures

5. **Binary Search**
   - Search in Rotated Sorted Array, Find Minimum in Rotated Sorted Array
6. **Linked List**
   - Reverse Linked List, Merge Two Sorted Lists, LRU Cache
7. **Trees**
   - Same Tree, Invert Binary Tree, Binary Tree Level Order Traversal
8. **Tries**
   - Implement Trie, Word Search II
9. **Heap / Priority Queue**
   - Find Median from Data Stream, Merge K Sorted Lists
10. **Backtracking**
    - Combination Sum, Word Search, N-Queens
11. **Graphs**
    - Number of Islands, Pacific Atlantic Water Flow, Course Schedule
12. **Dynamic Programming**
    - Climbing Stairs, House Robber, Longest Increasing Subsequence

Quick Reference: Problem Types to Algorithms

Problem Type	Algorithm/Data Structure
Search in sorted array	Binary Search
Track frequencies	Hash Map
Shortest path in graph	BFS (unweighted), Dijkstra's (weighted)
All paths in graph	DFS
Generate all combinations	Backtracking
Optimal substructure	Dynamic Programming
Find min/max k elements	Heap
Detect cycles in graph	DFS with visited tracking, Union-Find
Connected components	DFS, BFS, Union-Find
Substring problems	Sliding Window
Parentheses matching	Stack
Interval problems	Sorting + Greedy
Topological sorting	DFS, Kahn's algorithm