Time & Space Complexity Cheat Sheet - C++

BIG O NOTATION HIERARCHY (Fastest to Slowest)

```
O(1) < O(log n) < O(√n) < O(n) < O(n log n) < O(n²) < O(n³) < O(2^n) < O(n!)
Constant < Logarithmic < Root < Linear < Linearithmic < Quadratic < Cubic < Expon
```

ACCEPTABLE TIME COMPLEXITIES FOR HACKERRANK

For n = 10^5 (typical Citadel constraint):

Complexity	Max n	Example
O(1)	Any	Hash lookup, array access
O(log n)	10^18	Binary search
O(√n)	10^14	Prime checking
O(n)	10^8	Single loop
O(n log n)	10^6	Sorting, heap operations
$O(n^2)$	10^4	Nested loops (DANGER for n=10^5)
$O(n^3)$	500	Triple nested loops
O(2^n)	20	Subset generation

For 75-minute test with n=10^5:

- **SAFE:** O(n), O(n log n)
- **RISKY:** $O(n^2)$ will likely timeout
- **NO GO:** $O(n^3)$ or worse

C++ STL DATA STRUCTURES COMPLEXITY

Vector (Dynamic Array)

Operation	Average	Worst	Notes
`v[i]`	O(1)	O(1)	Index access
`v.push_back(x)`	O(1)	O(n)	Amortized O(1)
`v.insert(it, x)`	O(n)	O(n)	Insert at position
`v.pop_back()`	O(1)	O(1)	Remove last
`v.erase(it)`	O(n)	O(n)	Remove at position
`find(v.begin(), v.end(), x)`	O(n)	O(n)	Search
`sort(v.begin(), v.end())`	O(n log n)	O(n log n)	Sort
`reverse(v.begin(), v.end())`	O(n)	O(n)	Reverse
`min_element(v.begin(), v.end())`	O(n)	O(n)	Find min
`max_element(v.begin(), v.end())`	O(n)	O(n)	Find max
`accumulate(v.begin(), v.end(), 0)`	O(n)	O(n)	Sum

unordered_map (Hash Table)

Operation	Average	Worst	Notes
`m[key]`	O(1)	O(n)	Access/Insert
`m.erase(key)`	O(1)	O(n)	Delete
`m.count(key)`	O(1)	O(n)	Check existence
`m.find(key)`	O(1)	O(n)	Find
Iteration	O(n)	O(n)	All keys/values

unordered_set (Hash Set)

Operation	Average	Worst	Notes

`s.insert(x)`	O(1)	O(n)	Add element
`s.erase(x)`	O(1)	O(n)	Remove element
`s.count(x)`	O(1)	O(n)	Check membership
`s.find(x)`	O(1)	O(n)	Find

map (Ordered Map - Red-Black Tree)

Operation	Complexity	Notes
`m[key]`	O(log n)	Access/Insert
`m.erase(key)`	O(log n)	Delete
`m.count(key)`	O(log n)	Check existence
`m.find(key)`	O(log n)	Find
Iteration	O(n)	In sorted order

set (Ordered Set - Red-Black Tree)

Operation	Complexity	Notes
`s.insert(x)`	O(log n)	Add element
`s.erase(x)`	O(log n)	Remove element
`s.count(x)`	O(log n)	Check membership
`s.find(x)`	O(log n)	Find

deque (Double-ended Queue)

Operation	Complexity	Notes
`d.push_back(x)`	O(1)	Add to right
`d.push_front(x)`	O(1)	Add to left
`d.pop_back()`	O(1)	Remove from right

`d.pop_front()`	O(1)	Remove from left
`d[i]`	O(1)	Random access

priority_queue (Heap)

Operation	Complexity	Notes
`pq.push(x)`	O(log n)	Insert
`pq.pop()`	O(log n)	Remove top
`pq.top()`	O(1)	Peek top
`make_heap()`	O(n)	Build heap from range

COMMON ALGORITHM COMPLEXITIES

Sorting Algorithms

```
// Built-in sort - O(n log n) time, O(log n) space
sort(arr.begin(), arr.end()); // Introsort (quicksort + heapsort + insertion)

// Stable sort - O(n log n) time, O(n) space
stable_sort(arr.begin(), arr.end());

// Counting sort - O(n + k) where k is range
// Only for integers in limited range
void countingSort(vector<int>& arr, int max_val) {
    vector<int> count(max_val + 1, 0);
    for (int num : arr) {
        count[num]++;
    }

    int idx = 0;
    for (int num = 0; num <= max_val; num++) {
        for (int i = 0; i < count[num]; i++) {
            arr[idx++] = num;
        }
    }
}
```

Search Algorithms

```cpp
// Linear search - O(n)
int linearSearch(vector<int>& arr, int target) {
    auto it = find(arr.begin(), arr.end(), target);
    if (it != arr.end()) {
        return it - arr.begin();
    }
    return -1;
}

// Binary search - O(log n)
int binarySearch(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

// STL binary search functions
bool found = binary_search(arr.begin(), arr.end(), target); // O(log n)
auto it = lower_bound(arr.begin(), arr.end(), target); // First >= target
auto it = upper_bound(arr.begin(), arr.end(), target); // First > target
```

Graph Algorithms

```cpp
// BFS - O(V + E) time, O(V) space
// DFS - O(V + E) time, O(V) space
// Dijkstra - O((V + E) log V) with heap
// Bellman-Ford - O(VE)
// Floyd-Warshall - O(V³)
// Kruskal's MST - O(E log E)
// Prim's MST - O(E log V) with heap
// Topological Sort - O(V + E)
```

Tree Algorithms

```cpp
// Tree traversal - O(n) time, O(h) space
```

```
// BST search - O(h) average, O(n) worst
// BST insert - O(h) average, O(n) worst
// Balanced tree operations - O(log n)
```

String Algorithms

```
// Pattern matching (naive) - O(nm)
// KMP pattern matching - O(n + m)
// Rabin-Karp - O(n + m) average
// String comparison - O(min(n, m))
// Substring search - O(n)
```

SPACE COMPLEXITY

Common Space Patterns

```
// O(1) - Constant space
int constantSpace(vector<int>& arr) {
    int result = 0;
    for (int num : arr) {
        result += num;
    }
    return result;
}

// O(n) - Linear space
vector<int> linearSpace(vector<int>& arr) {
    return arr; // Copy vector
}

// O(n) - Hash table for frequency
unordered_map<int, int> frequencyCount(vector<int>& arr) {
    unordered_map<int, int> freq;
    for (int num : arr) {
        freq[num]++;
    }
    return freq;
}

// O(h) - Recursion depth for tree
int treeHeight(TreeNode* root) {
    if (!root) return 0;
    return 1 + max(treeHeight(root->left), treeHeight(root->right));
```

```
}

// O(2^n) - All subsets
void allSubsets(vector<int>& arr, int index, vector<int>& current,
                vector<vector<int>>& result) {
    if (index == arr.size()) {
        result.push_back(current);
        return;
    }

    // Include current element
    current.push_back(arr[index]);
    allSubsets(arr, index + 1, current, result);
    current.pop_back();

    // Exclude current element
    allSubsets(arr, index + 1, current, result);
}
```

OPTIMIZATION TECHNIQUES

1. Use Hash Table for O(1) Lookup

```
// SLOW - O(n²)
for (int num : arr1) {
    if (find(arr2.begin(), arr2.end(), num) != arr2.end()) { // O(n) search
        result.push_back(num);
    }
}

// FAST - O(n)
unordered_set<int> set2(arr2.begin(), arr2.end()); // O(n) to build
for (int num : arr1) { // O(n)
    if (set2.count(num)) { // O(1) lookup
        result.push_back(num);
    }
}
```

2. Avoid Repeated Calculations

```
// SLOW - O(n²)
for (int i = 0; i < n; i++) {
    int total = accumulate(arr.begin(), arr.begin() + i, 0); // Recalculates each
```

```
}

// FAST - O(n)
vector<int> prefix(arr.size() + 1, 0);
for (int i = 0; i < arr.size(); i++) {
    prefix[i + 1] = prefix[i] + arr[i];
}
```

3. Two Pointers Instead of Nested Loops

```
// SLOW - O(n²)
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        if (arr[i] + arr[j] == target) {
            return {i, j};
        }
    }
}

// FAST - O(n) with sorted array
int left = 0, right = n - 1;
while (left < right) {
    int total = arr[left] + arr[right];
    if (total == target) {
        return {left, right};
    } else if (total < target) {
        left++;
    } else {
        right--;
    }
}
```

4. Sliding Window Instead of Recalculating

```
// SLOW - O(n²)
for (int i = 0; i <= n - k; i++) {
    int window_sum = accumulate(arr.begin() + i, arr.begin() + i + k, 0); // O(k)
}

// FAST - O(n)
int window_sum = accumulate(arr.begin(), arr.begin() + k, 0);
for (int i = k; i < n; i++) {
    window_sum += arr[i] - arr[i - k]; // O(1) update
}
```

5. Use deque for Queue Operations

```
// SLOW - O(n) for pop_front on vector
vector<int> queue;
queue.push_back(x);
queue.erase(queue.begin()); // O(n)

// FAST - O(1) for all operations
deque<int> queue;
queue.push_back(x);
queue.pop_front(); // O(1)
```

6. Binary Search Instead of Linear Search

```
// SLOW - O(n)
for (int i = 0; i < sorted_arr.size(); i++) {
    if (sorted_arr[i] == target) {
        return i;
    }
}

// FAST - O(log n)
int left = 0, right = sorted_arr.size() - 1;
while (left <= right) {
    int mid = left + (right - left) / 2;
    if (sorted_arr[mid] == target) {
        return mid;
    } else if (sorted_arr[mid] < target) {
        left = mid + 1;
    } else {
        right = mid - 1;
    }
}

// Or use STL
auto it = lower_bound(sorted_arr.begin(), sorted_arr.end(), target);
if (it != sorted_arr.end() && *it == target) {
    return it - sorted_arr.begin();
}
```

COMMON MISTAKES TO AVOID

1. Nested Loops on Large Input

```
// TIMEOUT for n=10^5
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) { // O(n²)
        process(i, j);
    }
}
```

2. Sorting Inside Loop

```
// TIMEOUT - O(n² log n)
for (int i = 0; i < n; i++) {
    vector<int> sorted_sub(arr.begin(), arr.begin() + i);
    sort(sorted_sub.begin(), sorted_sub.end()); // Sort every iteration
}
```

3. String Concatenation in Loop

```
// SLOW - O(n²) because strings create new copies
string result = "";
for (char c : chars) {
    result += c; // Creates new string each time
}

// FAST - O(n)
string result(chars.begin(), chars.end());
// Or
result.reserve(chars.size());
for (char c : chars) {
    result += c;
}
```

4. Using Vector for Frequent Membership Testing

```
// SLOW - O(n) per lookup
vector<int> seen;
for (int num : arr) {
    if (find(seen.begin(), seen.end(), num) != seen.end()) { // O(n)
        continue;
    }
    seen.push_back(num);
}
```

```cpp
// FAST - O(1) per lookup
unordered_set<int> seen;
for (int num : arr) {
    if (seen.count(num)) { // O(1)
        continue;
    }
    seen.insert(num);
}
```

5. Unnecessary Deep Copies

```cpp
// SLOW - Copies entire vector each time
void backtrack(vector<int> arr) { // Pass by value creates copy!
    if (condition) {
        result.push_back(arr);
    }

    for (int i = 0; i < arr.size(); i++) {
        vector<int> new_arr = arr; // BAD - unnecessary copy
        new_arr[i] = x;
        backtrack(new_arr);
    }
}

// FAST - Modify in place
void backtrack(vector<int>& arr) { // Pass by reference
    if (condition) {
        result.push_back(arr);
    }

    for (int i = 0; i < arr.size(); i++) {
        int old_val = arr[i];
        arr[i] = x; // Modify
        backtrack(arr);
        arr[i] = old_val; // Restore
    }
}
```

QUICK COMPLEXITY CHECKS

Before implementing, ask:

1. **What's n?** (array length, string length, etc.)
2. **How many nested loops?** Each adds O(n)

3. **Am I sorting?** That's $O(n \log n)$
4. **Am I using hash table?** Lookups are $O(1)$
5. **Am I searching unsorted array?** That's $O(n)$
6. **Will this timeout?**
   - $n=10^5$ and $O(n^2)$ → YES
   - $n=10^5$ and $O(n \log n)$ → NO
   - $n=10^3$ and $O(n^2)$ → NO

RULE OF THUMB FOR CITADEL

n range	Acceptable complexity
$n \leq 10$	$O(n!)$ is acceptable (brute force permutations)
$n \leq 20$	$O(2^n)$ is acceptable (subset enumeration)
$n \leq 500$	$O(n^3)$ is acceptable
$n \leq 10^4$	$O(n^2)$ is acceptable
$n \leq 10^5$	$O(n \log n)$ or $O(n)$ required
$n \leq 10^6$	$O(n)$ or $O(\log n)$ required

For $n=10^5$ (most Citadel problems):

- ✅ $O(n)$, $O(n \log n)$
- ⚠️ $O(n^2)$ - likely timeout
- ❌ $O(n^3)$ or worse - guaranteed timeout

C++ PERFORMANCE TIPS

Fast I/O

```
// Disable sync with C I/O for faster cin/cout
ios_base::sync_with_stdio(false);
cin.tie(NULL);
```

```
// Or use scanf/printf for even faster I/O
int n;
scanf("%d", &n);
printf("%d\n", result);
```

Reserve Space

```
// Reserve space for vectors when size is known
vector<int> arr;
arr.reserve(n); // Avoids reallocations
```

Use const& for Large Objects

```
// SLOW - Copies entire vector
void process(vector<int> arr) { }

// FAST - No copy
void process(const vector<int>& arr) { }
```

Prefer emplace over push

```
// Constructs in place (slightly faster)
v.emplace_back(args);
// vs
v.push_back(Type(args));
```

Use auto for Iterators

```
// More readable and potentially faster
for (auto& x : arr) { }
for (const auto& [key, value] : map) { }
```