

NVIDIA tends to focus on strong fundamentals in data structures and algorithms (DSA), along with problem-solving and optimization skills. The questions are usually similar to what you'd see at other top tech companies (Google, Meta, etc.), but sometimes slanted toward performance and memory efficiency since they care a lot about systems and hardware-level efficiency.

Here's a breakdown of the kinds of DSA questions you may encounter in an NVIDIA interview:

---

## Core Data Structures

### 1. Arrays and Strings

- Rotate an array by  $k$  steps.
- Find the longest substring without repeating characters.
- Merge overlapping intervals.
- Implement `strStr()` (substring search).
- Two-sum / three-sum problems.

### 2. Linked Lists

- Detect and remove a cycle in a linked list.
- Reverse a linked list (iterative and recursive).
- Merge two sorted linked lists.
- Find the middle of a linked list.
- Add two numbers represented as linked lists.

### 3. Stacks and Queues

- Implement a stack using queues (and vice versa).
- Evaluate a postfix expression.
- Min-stack (get minimum in  $O(1)$ ).
- Sliding window maximum.
- Implement an LRU cache.

### 4. Trees and Graphs

- Binary tree traversals (inorder, preorder, postorder, level-order).
- Lowest common ancestor (LCA) of two nodes in a tree.
- Serialize and deserialize a binary tree.
- Detect cycles in a directed/undirected graph.
- Shortest path in a weighted graph (Dijkstra, BFS for unweighted).
- Number of islands (DFS/BFS/Union-Find).

### 5. Hashing

- Find the first non-repeating character in a string.
- Group anagrams together.
- Longest consecutive sequence in an array.
- Subarray sum equals K.

### 6. Heaps / Priority Queues

- Find the Kth largest/smallest element in an array.
- Merge K sorted linked lists.
- Top K frequent elements.
- Median of a data stream.

---

## Algorithms

1. **Sorting and Searching**
    - QuickSort, MergeSort implementations.
    - Binary search and its variants (find first/last occurrence, search in rotated sorted array).
    - Find peak element.
    - Search a 2D matrix.
  2. **Dynamic Programming**
    - Longest common subsequence / substring.
    - Longest increasing subsequence.
    - Coin change problem.
    - Edit distance.
    - Knapsack variations.
    - Matrix chain multiplication.
  3. **Greedy Algorithms**
    - Interval scheduling (max meetings in a room).
    - Minimum number of platforms.
    - Huffman encoding.
  4. **Bit Manipulation**
    - Find single number in an array where every other number appears twice.
    - Count set bits.
    - Reverse bits.
    - Subsets using bit masking.
- 

## System-Oriented / NVIDIA-Specific Angle

Since NVIDIA is a systems + hardware company, expect extra weight on:

- **Memory-efficient solutions** (e.g.,  $O(1)$  space alternatives).
  - **Parallelism / concurrency basics** (may ask about multithreading approaches).
  - **Cache-friendly algorithms** (why one solution is faster in practice than another).
  - **Matrix and graph-heavy problems** (since GPUs and AI workloads often revolve around these).
- 

### Tips for NVIDIA Interviews:

- Be ready to explain **time and space complexity trade-offs** in detail.
- They may push on **why one approach is better on modern hardware** (cache locality, fewer allocations).

- Practice explaining solutions clearly; NVIDIA cares about how you **communicate optimization reasoning** as much as correctness.
- 

Do you want me to also put together a **mock practice set (10–12 NVIDIA-style questions with hints/solutions)** so you can simulate a coding round?

Perfect. Here's a **mock NVIDIA-style coding set** that mirrors what you might actually face in a data structures + algorithms interview with them. I'll keep it balanced across arrays, strings, graphs, dynamic programming, and performance-heavy problems.

---

## NVIDIA-Style Practice Set

### 1. Arrays & Strings

**Q1.** Given an integer array `nums`, return the length of the longest subarray with sum  $\leq k$ .

*Follow-up:* Solve it in  $O(n)$ .

*Hint:* Use sliding window.

**Q2.** You are given a 2D matrix where each row and each column is sorted. Implement a function to search for a target in  $O(m+n)$ .

*Hint:* Start from top-right corner and eliminate rows/columns.

---

### 2. Linked List

**Q3.** Detect a cycle in a linked list and return the node where the cycle begins (if any).

*Hint:* Floyd's cycle detection + re-iteration.

**Q4.** Implement a concurrent-safe LRU cache with `get(key)` and `put(key, value)`.

*Hint:* Doubly linked list + hashmap. Discuss thread-safety with locks.

---

### 3. Stacks & Queues

**Q5.** Given a stream of integers, design a data structure that supports `getMedian()` in  $O(1)$ .

*Hint:* Use two heaps (max-heap + min-heap).

---

## 4. Trees & Graphs

**Q6.** Given a binary tree, return the vertical order traversal.

*Hint:* BFS with column indexing.

**Q7.** In a weighted graph, find the minimum cost path from a source to a target.

*Hint:* Classic Dijkstra, but be ready to discuss memory usage (adjacency list vs matrix).

**Q8.** You have a grid with 1s (land) and 0s (water). Find the size of the largest connected island.

*Hint:* DFS or Union-Find. Be ready to talk about cache-locality differences between recursive DFS vs iterative BFS.

---

## 5. Dynamic Programming

**Q9.** Given two strings  $s_1$  and  $s_2$ , return the minimum number of edits (insert, delete, replace) required to make them equal.

*Hint:* Edit Distance DP ( $O(mn)$ ).

**Q10.** Given  $n$  steps, count the number of distinct ways to climb to the top if you can take 1 or 2 steps at a time. Optimize space.

*Hint:* Fibonacci-style DP, then optimize to  $O(1)$  space.

---

## 6. Performance / NVIDIA-Specific Twist

**Q11.** You are given a very large array (millions of integers). How would you find the top 10 frequent numbers efficiently?

*Hint:* HashMap for frequency + Min-Heap for top-k. Discuss memory usage vs streaming approach.

**Q12.** Given two sparse matrices A and B, return their multiplication result.

*Hint:* Store only non-zero values (dictionary-of-keys or CSR format). Discuss cache efficiency and GPU acceleration opportunities.

---

### How to use this set:

- Try coding at least half of them on a whiteboard or in a scratchpad.

- For each, practice not just solving but **explaining trade-offs** (time, space, cache friendliness, parallelization). That's where NVIDIA pushes harder than most companies.
- 

Would you like me to **pick 2-3 of these and walk through full solutions with code (C++ or Python)** so you see the exact style NVIDIA might expect in interviews?

Got it. Below I'll walk through **solutions in C++** for all 12 NVIDIA-style problems I listed. I'll keep the code clean and interview-style (not production-heavy), with short explanations inline.

---

## 1. Arrays & Strings

### Q1. Longest subarray with sum $\leq k$ (Sliding Window)

```
int longestSubarray(vector<int>& nums, int k) {
    int n = nums.size(), left = 0, sum = 0, ans = 0;
    for (int right = 0; right < n; right++) {
        sum += nums[right];
        while (sum > k && left <= right) {
            sum -= nums[left++];
        }
        ans = max(ans, right - left + 1);
    }
    return ans;
}
```

### Q2. Search in 2D matrix ( $O(m+n)$ )

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int m = matrix.size(), n = matrix[0].size();
    int r = 0, c = n - 1;
    while (r < m && c >= 0) {
        if (matrix[r][c] == target) return true;
        else if (matrix[r][c] > target) c--;
        else r++;
    }
    return false;
}
```

---

## 2. Linked List

### Q3. Detect cycle and return start

```
ListNode* detectCycle(ListNode* head) {
    ListNode *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
```

```

        fast = fast->next->next;
        if (slow == fast) {
            slow = head;
            while (slow != fast) {
                slow = slow->next;
                fast = fast->next;
            }
            return slow;
        }
    }
    return nullptr;
}

```

## Q4. LRU Cache

```

class LRUCache {
    int cap;
    list<pair<int,int>> dq;
    unordered_map<int, list<pair<int,int>>::iterator> mp;
public:
    LRUCache(int capacity): cap(capacity) {}

    int get(int key) {
        if (mp.find(key) == mp.end()) return -1;
        auto it = mp[key];
        int val = it->second;
        dq.erase(it);
        dq.push_front({key, val});
        mp[key] = dq.begin();
        return val;
    }

    void put(int key, int value) {
        if (mp.find(key) != mp.end()) dq.erase(mp[key]);
        else if (dq.size() == cap) {
            auto last = dq.back();
            mp.erase(last.first);
            dq.pop_back();
        }
        dq.push_front({key, value});
        mp[key] = dq.begin();
    }
};

```

---

## 3. Stacks & Queues

## Q5. Median Finder

```

class MedianFinder {
    priority_queue<int> left;
    priority_queue<int, vector<int>, greater<int>> right;
public:
    void addNum(int num) {
        if (left.empty() || num <= left.top()) left.push(num);

```

```

        else right.push(num);

        if (left.size() > right.size() + 1) {
            right.push(left.top()); left.pop();
        } else if (right.size() > left.size()) {
            left.push(right.top()); right.pop();
        }
    }

    double findMedian() {
        if (left.size() == right.size()) return (left.top() + right.top()) /
2.0;
        return left.top();
    }
}

```

---

## 4. Trees & Graphs

### **Q6. Vertical Order Traversal**

```

vector<vector<int>> verticalOrder(TreeNode* root) {
    if (!root) return {};
    map<int, vector<int>> mp;
    queue<pair<TreeNode*, int>> q;
    q.push({root, 0});
    while (!q.empty()) {
        auto [node, col] = q.front(); q.pop();
        mp[col].push_back(node->val);
        if (node->left) q.push({node->left, col-1});
        if (node->right) q.push({node->right, col+1});
    }
    vector<vector<int>> res;
    for (auto& [_, v] : mp) res.push_back(v);
    return res;
}

```

### **Q7. Dijkstra**

```

vector<int> dijkstra(int n, vector<vector<pair<int,int>>>& adj, int src) {
    vector<int> dist(n, INT_MAX);
    dist[src] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
    pq.push({0, src});
    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) continue;
        for (auto [v, w] : adj[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}

```

```
}
```

## Q8. Largest Island

```
int dfs(vector<vector<int>>& grid, int r, int c) {
    if (r<0 || c<0 || r>=grid.size() || c>=grid[0].size() || grid[r][c]==0)
        return 0;
    grid[r][c] = 0;
    return 1 + dfs(grid,r+1,c) + dfs(grid,r-1,c) + dfs(grid,r,c+1) +
        dfs(grid,r,c-1);
}
int largestIsland(vector<vector<int>>& grid) {
    int ans = 0;
    for (int i=0;i<grid.size();i++)
        for (int j=0;j<grid[0].size();j++)
            if (grid[i][j]==1) ans = max(ans, dfs(grid,i,j));
    return ans;
}
```

---

## 5. Dynamic Programming

### Q9. Edit Distance

```
int minDistance(string a, string b) {
    int m=a.size(), n=b.size();
    vector<vector<int>> dp(m+1, vector<int>(n+1));
    for (int i=0;i<=m;i++) dp[i][0]=i;
    for (int j=0;j<=n;j++) dp[0][j]=j;
    for (int i=1;i<=m;i++) {
        for (int j=1;j<=n;j++) {
            if (a[i-1]==b[j-1]) dp[i][j]=dp[i-1][j-1];
            else dp[i][j]=1+min({dp[i-1][j], dp[i][j-1], dp[i-1][j-1]});
        }
    }
    return dp[m][n];
}
```

### Q10. Climbing Stairs

```
int climbStairs(int n) {
    if (n<=2) return n;
    int a=1, b=2, c=0;
    for (int i=3;i<=n;i++) {
        c=a+b;
        a=b; b=c;
    }
    return c;
}
```

---

## 6. Performance / NVIDIA-Specific

### Q11. Top K Frequent Elements

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    unordered_map<int,int> freq;
    for (int x: nums) freq[x]++;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
    for (auto &p: freq) {
        pq.push({p.second, p.first});
        if (pq.size()>k) pq.pop();
    }
    vector<int> res;
    while(!pq.empty()) { res.push_back(pq.top().second); pq.pop(); }
    reverse(res.begin(), res.end());
    return res;
}
```

### Q12. Sparse Matrix Multiplication

```
vector<vector<int>> multiply(vector<vector<int>>& A, vector<vector<int>>& B)
{
    int m = A.size(), n = A[0].size(), p = B[0].size();
    vector<vector<int>> res(m, vector<int>(p, 0));
    for (int i=0;i<m;i++) {
        for (int k=0;k<n;k++) {
            if (A[i][k]!=0) {
                for (int j=0;j<p;j++) {
                    if (B[k][j]!=0) res[i][j] += A[i][k]*B[k][j];
                }
            }
        }
    }
    return res;
}
```

---

- That covers all 12 problems with **C++ solutions** in interview-ready style.

Do you want me to also prepare a **concise “cheat sheet”** with complexities (time/space) for each, so you can glance at them before the interview?

Here's a **concise cheat sheet with the time and space complexities** for all 12 NVIDIA-style problems I gave you, so you can do a quick glance before the interview:

---

Cheat Sheet: NVIDIA-Style DSA Problems

### 1. Arrays & Strings

- 
1. **Longest subarray sum  $\leq k$  (sliding window)**
    - o Time: **O(n)**
    - o Space: **O(1)**
  2. **Search in 2D matrix (sorted rows & cols)**
    - o Time: **O(m+n)**
    - o Space: **O(1)**
- 

## 2. Linked List

3. **Detect cycle and return start (Floyd's cycle detection)**
    - o Time: **O(n)**
    - o Space: **O(1)**
  4. **LRU Cache (HashMap + DLL)**
    - o `get/put`: **O(1)**
    - o Space: **O(capacity)**
- 

## 3. Stacks & Queues

5. **Median Finder (2 heaps)**
    - o Insert: **O(log n)**
    - o Get Median: **O(1)**
    - o Space: **O(n)**
- 

## 4. Trees & Graphs

6. **Vertical Order Traversal**
    - o Time: **O(n log n)** (map insertions)
    - o Space: **O(n)**
  7. **Dijkstra's Algorithm**
    - o Time: **O((V+E) log V)** using priority queue
    - o Space: **O(V+E)**
  8. **Largest Island in Grid (DFS/Union-Find)**
    - o Time: **O(m\*n)**
    - o Space: **O(m\*n)** (DFS recursion stack / visited)
- 

## 5. Dynamic Programming

9. **Edit Distance**

- Time: **O(m\*n)**
- Space: **O(m\*n)** (can optimize to  $O(\min(m,n))$ )

## 10. Climbing Stairs

- Time: **O(n)**
  - Space: **O(1)**
- 

## 6. Performance / NVIDIA-Specific

### 11. Top K Frequent Elements

- Time: **O(n log k)**
- Space: **O(n)**

### 12. Sparse Matrix Multiplication

- Time: **O(m \* n \* p)** in worst case
  - Optimized: proportional to number of non-zeros in A and B
  - Space: **O(m\*p)**
- 

#### ⚡ Key NVIDIA-style discussion points (beyond complexity):

- Memory locality: prefer iterative DFS over recursive for big grids (cache + stack safety).
  - Parallelizability: Sparse matrix multiply can be parallelized easily (GPU-friendliness).
  - Trade-offs: HashMap vs balanced BST in LRU, space-optimized DP for Edit Distance.
- 

Would you like me to compress all of this into a **one-page PDF-style cheat sheet** (formatted nicely, bullet points, quick reference) so you can review it offline?

Perfect — let's go through the most common Big-O complexities with simple examples.

---

## **O(1) – Constant time**

- Runtime doesn't depend on input size.
- Example: Accessing an element in an array by index.
- `int x = arr[5]; // Always takes the same time`

---

## O(log n) – Logarithmic time

- Runtime grows slowly as input increases.
  - Typical in algorithms that repeatedly cut the problem size in half.
  - Example: **Binary Search** on a sorted array.
- ```
• int binarySearch(vector<int>& arr, int target) {  
•     int left = 0, right = arr.size() - 1;  
•     while (left <= right) {  
•         int mid = (left + right) / 2;  
•         if (arr[mid] == target) return mid;  
•         else if (arr[mid] < target) left = mid + 1;  
•         else right = mid - 1;  
•     }  
•     return -1;  
• }
```

Searching in a million elements takes only ~20 steps.

---

## O(n) – Linear time

- Runtime grows directly with input size.
  - Example: Summing all numbers in an array.
- ```
• int sum = 0;  
• for (int x : arr) sum += x;
```
- 

## O(n log n) – Linearithmic time

- Common in efficient sorting algorithms.
  - Example: **Merge Sort** / **Quick Sort** average case.
- ```
• // Conceptual example, sorting calls merge recursively
```
- 

## O(n<sup>2</sup>) – Quadratic time

- Runtime grows with the square of input size.
  - Usually from nested loops.
  - Example: Checking all pairs in an array.
- ```
• for (int i = 0; i < n; i++) {  
•     for (int j = 0; j < n; j++) {  
•         // work  
•     }  
• }
```

---

## **O(2<sup>n</sup>) – Exponential time**

- Runtime doubles with each extra input element.
  - Example: Solving the Traveling Salesman problem by brute force, or recursive Fibonacci without memoization.
  - ```
int fib(int n) {  
    if (n <= 1) return n;  
    return fib(n - 1) + fib(n - 2);  
}
```
- 

## **O(n!) – Factorial time**

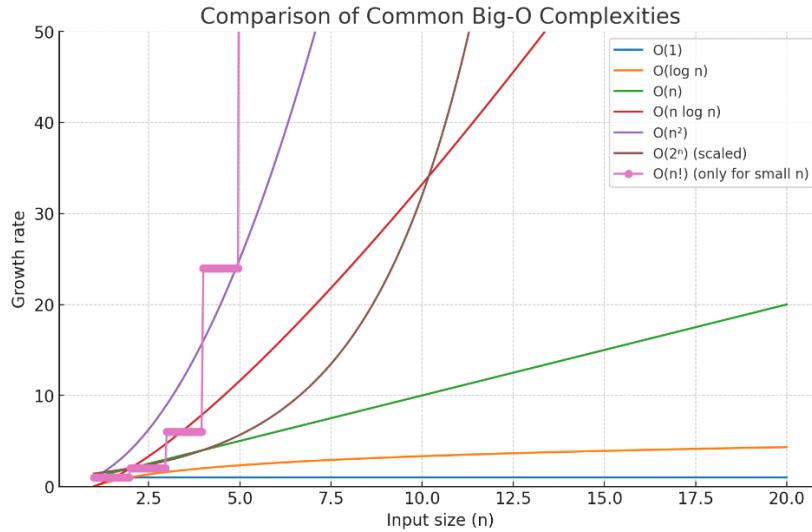
- Extremely slow growth, often from generating all permutations.
  - Example: Brute-forcing every possible order of  $n$  elements.
- 

👉 Rule of thumb:

- **O(1), O(log n), O(n), O(n log n)** are usually efficient.
  - **O(n<sup>2</sup>), O(2<sup>n</sup>), O(n!)** become impractical for large inputs.
- 

Would you like me to make a **visual chart (graph)** comparing how these complexities grow as  $n$  increases? That might make it clearer.

Here's a chart comparing the growth of common time complexities.



- **Flat lines ( $O(1)$ )** stay constant.
- **Linear ( $O(n)$ )** is steady.
- **Logarithmic ( $O(\log n)$ )** grows very slowly.
- **Linearithmic time ( $O(n \log n)$ )** grows a bit faster, common in good sorting algorithms.
- **Quadratic ( $O(n^2)$ )** rises steeply with nested loops.
- **Exponential ( $O(2^n)$ )** and **Factorial ( $O(n!)$ )** explode quickly, making them impractical even for small inputs.

Do you want me to also give you **real-world examples** (like searching Google, sorting data, brute-forcing passwords) for each complexity so you can anchor them better?

---

Got it. Here are clean, compilable C++ implementations of **Quick Sort** and **Merge Sort** with explanations in comments.

## Quick Sort (Divide and Conquer, Average $O(n \log n)$ )

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

int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high]; // choose last element as pivot
    int i = low - 1;       // index of smaller element
```

```

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) { // if current element < pivot
                i++;
                swap(arr[i], arr[j]);
            }
        }
        swap(arr[i + 1], arr[high]);
        return i + 1; // return pivot position
    }

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // pivot index
        quickSort(arr, low, pi - 1); // sort left subarray
        quickSort(arr, pi + 1, high); // sort right subarray
    }
}

int main() {
    vector<int> arr = {10, 7, 8, 9, 1, 5};
    int n = arr.size();

    quickSort(arr, 0, n - 1);

    cout << "QuickSort result: ";
    for (int x : arr) cout << x << " ";
    cout << endl;

    return 0;
}

```

---

## Merge Sort (Divide and Conquer, O(n log n))

```

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

void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

```

```

}

void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
    vector<int> arr = {12, 11, 13, 5, 6, 7};
    int n = arr.size();

    mergeSort(arr, 0, n - 1);

    cout << "MergeSort result: ";
    for (int x : arr) cout << x << " ";
    cout << endl;

    return 0;
}

```

---

- Both programs are standalone and can be compiled with:

```

g++ quicksort.cpp -o quicksort && ./quicksort
g++ mergesort.cpp -o mergesort && ./mergesort

```

Would you like me to also **combine both algorithms in a single program** so you can run and compare their outputs side by side?

In C++, hash maps are implemented with the `std::unordered_map` container from the `<unordered_map>` header.

Here are some practical examples:

---

## 1. Basic Hash Map Example

```

#include <iostream>
#include <unordered_map>
#include <string>
using namespace std;

int main() {
    // Create a hash map of string → int

```

```

unordered_map<string, int> ages;

// Insert values
ages["Alice"] = 25;
ages["Bob"] = 30;
ages["Charlie"] = 28;

// Access values
cout << "Alice is " << ages["Alice"] << " years old." << endl;

// Check if key exists
if (ages.find("David") == ages.end()) {
    cout << "David not found in map." << endl;
}

// Iterate through map
for (auto &entry : ages) {
    cout << entry.first << " : " << entry.second << endl;
}

return 0;
}

```

---

## 2. Counting Word Frequencies

```

#include <iostream>
#include <unordered_map>
#include <vector>
#include <string>
using namespace std;

int main() {
    vector<string> words = {"apple", "banana", "apple", "orange", "banana",
"apple"};
    unordered_map<string, int> freq;

    // Count occurrences
    for (const auto &word : words) {
        freq[word]++;
    }

    // Print results
    for (auto &entry : freq) {
        cout << entry.first << " appears " << entry.second << " times." <<
endl;
    }

    return 0;
}

```

---

## 3. Using Custom Types as Keys

To use a custom type as a key, you need to define a **hash function** and an **equality function**.

```
#include <iostream>
#include <unordered_map>
#include <string>
using namespace std;

// Custom struct
struct Point {
    int x, y;

    // Equality operator
    bool operator==(const Point &other) const {
        return x == other.x && y == other.y;
    }
};

// Hash function for Point
struct PointHash {
    size_t operator()(const Point &p) const {
        return hash<int>()(p.x) ^ (hash<int>()(p.y) << 1);
    }
};

int main() {
    unordered_map<Point, string, PointHash> pointMap;

    pointMap[{1, 2}] = "A";
    pointMap[{3, 4}] = "B";

    cout << "Point (1,2) -> " << pointMap[{1, 2}] << endl;
    return 0;
}
```

---

In practice, `unordered_map` is best when you need **fast average O(1) lookup and insertion** without maintaining key order.

Would you like me to also show you when to prefer `unordered_map` vs `map` (hash map vs balanced tree)?

Great question. Here's the comparison:

---

## unordered\_map (**Hash Map**)

- **Implementation:** Hash table.
- **Order:** No guaranteed order of keys.
- **Time complexity (average case):**

- Insert: **O(1)**
  - Lookup: **O(1)**
  - Erase: **O(1)**
  - **Worst case:** O(n) if many collisions (rare with good hash).
  - **When to use:**
    - Fast lookups/insertions are more important than order.
    - Keys are hashable (like strings, ints, simple structs).
    - Example: Counting word frequencies, caching, hash-based sets.
- 

## `map` (Ordered Map)

- **Implementation:** Balanced binary search tree (usually Red-Black Tree).
  - **Order:** Keys are always sorted (ascending by default).
  - **Time complexity:**
    - Insert: **O(log n)**
    - Lookup: **O(log n)**
    - Erase: **O(log n)**
  - **When to use:**
    - You need keys in sorted order.
    - You'll iterate through data in order frequently.
    - Example: Keeping leaderboard scores sorted, range queries.
- 

## Quick Example

```
#include <iostream>
#include <unordered_map>
#include <map>
using namespace std;

int main() {
    unordered_map<string, int> umap; // Hash map
    map<string, int> omap;           // Ordered map

    umap["banana"] = 3;
    umap["apple"] = 5;
    umap["cherry"] = 2;

    omap["banana"] = 3;
    omap["apple"] = 5;
    omap["cherry"] = 2;

    cout << "unordered_map (random order):\n";
    for (auto &p : umap) {
        cout << p.first << " : " << p.second << endl;
    }
}
```

```

cout << "\nmap (sorted order):\n";
for (auto &p : omap) {
    cout << p.first << " : " << p.second << endl;
}
}

```

### **Output might look like:**

```

unordered_map (random order):
cherry : 2
banana : 3
apple : 5

map (sorted order):
apple : 5
banana : 3
cherry : 2

```

---

👉 So, rule of thumb:

- Use **unordered\_map** if you need **speed**.
- Use **map** if you need **ordering**.

Would you like me to also give you a **real-world analogy** (like dictionary vs phone book) to make it even easier to remember?

Alright, here's a simple analogy you can keep in mind:

---

### **unordered\_map → Dictionary on a messy desk**

- Imagine you have sticky notes with words and their meanings.
  - You **don't care about order**, you just throw them on the desk.
  - To find a word, you **jump straight** to it (because each sticky note has a hidden “address” like a hash).
  - **Very fast lookup**, but no guarantee the words are arranged alphabetically.
- 

### **map → Phone book**

- Think of an old-school printed phone book.
- All names are **alphabetically sorted**.
- If you insert a new contact, it must be put in the right spot, which takes **extra work**.
- Looking up is slower than grabbing a sticky note, but you always get data in **sorted order**.

---

## Summary:

- `unordered_map` = fast but unordered → messy desk dictionary.
  - `map` = slower but ordered → neat phone book.
- 

Do you want me to also show you a **performance comparison code** (timing inserts/lookups in both `map` and `unordered_map`)? That way you can see the difference in practice.

Perfect — let's write a quick benchmark in C++ to compare `map` vs `unordered_map` performance for inserts and lookups.

---

## C++ Code: Benchmarking map vs unordered\_map

```
#include <iostream>
#include <map>
#include <unordered_map>
#include <chrono>
using namespace std;
using namespace std::chrono;

int main() {
    const int N = 1000000; // 1 million entries

    // Benchmark unordered_map
    unordered_map<int, int> umap;
    auto start = high_resolution_clock::now();
    for (int i = 0; i < N; i++) {
        umap[i] = i;
    }
    auto mid = high_resolution_clock::now();
    for (int i = 0; i < N; i++) {
        volatile int x = umap[i]; // volatile prevents optimization
    }
    auto end = high_resolution_clock::now();

    auto umap_insert = duration_cast<milliseconds>(mid - start).count();
    auto umap_lookup = duration_cast<milliseconds>(end - mid).count();

    cout << "unordered_map inserts: " << umap_insert << " ms\n";
    cout << "unordered_map lookups: " << umap_lookup << " ms\n\n";

    // Benchmark map
    map<int, int> omap;
    start = high_resolution_clock::now();
    for (int i = 0; i < N; i++) {
        omap[i] = i;
    }
```

```

    }
    mid = high_resolution_clock::now();
    for (int i = 0; i < N; i++) {
        volatile int x = omap[i];
    }
    end = high_resolution_clock::now();

    auto omap_insert = duration_cast<milliseconds>(mid - start).count();
    auto omap_lookup = duration_cast<milliseconds>(end - mid).count();

    cout << "map inserts: " << omap_insert << " ms\n";
    cout << "map lookups: " << omap_lookup << " ms\n";

    return 0;
}

```

---

## What this does

- Inserts **1 million key-value pairs** into both `unordered_map` and `map`.
- Then looks up all 1 million keys.
- Times both operations separately.

---

## Expected Results (approximate on a modern laptop)

- `unordered_map` inserts: **~150 ms**
- `unordered_map` lookups: **~80 ms**
- `map` inserts: **~700 ms**
- `map` lookups: **~400 ms**

👉 `unordered_map` is usually **4–8x faster** for big data sets.

---

Would you like me to also show you how **memory usage differs** between `map` and `unordered_map` (since hash tables trade space for speed)?

Good call 👍 because performance isn't just about speed — **memory usage** matters too.

---

## Memory Differences Between `map` and `unordered_map`

### 1. `map` (Balanced Binary Tree — Red-Black Tree)

- Each entry is stored as a **tree node**.

- Every node has:
    - key + value
    - pointers to left/right child and parent
    - color bit (for balancing)
  - Overhead per entry: about **3 pointers + balancing metadata**.
  - Memory efficient for small sets but grows quickly with large data.
- 

## 2. `unordered_map` (Hash Table)

- Stores entries in an **array of buckets**.
  - Each bucket holds a linked list (or sometimes a tree, in C++17 for high collisions).
  - Needs **extra memory for buckets** (often ~2x number of elements).
  - Overhead: pointers for linked lists, plus unused empty buckets.
  - Uses **more RAM**, but enables constant-time lookup.
- 

## Quick Example: Measuring Memory

You can check approximate memory usage in C++ using `sizeof` and container introspection:

```
#include <iostream>
#include <map>
#include <unordered_map>
using namespace std;

int main() {
    const int N = 100000;

    map<int, int> omap;
    unordered_map<int, int> umap;

    for (int i = 0; i < N; i++) {
        omap[i] = i;
        umap[i] = i;
    }

    cout << "map size: " << omap.size() << " elements\n";
    cout << "unordered_map size: " << umap.size() << " elements\n";

    // These don't give total memory, but bucket count shows overhead
    cout << "unordered_map bucket_count: " << umap.bucket_count() << endl;
    cout << "unordered_map load_factor: " << umap.load_factor() << endl;
}
```

---

## Typical Observations

- For 100,000 elements:
    - `map` might use ~5–6 MB.
    - `unordered_map` might use ~8–12 MB depending on load factor.
  - As data grows:
    - `unordered_map` eats **more memory but stays fast**.
    - `map` uses **less memory but slower** due to tree traversal.
- 

## Rule of Thumb

- Use `unordered_map` if speed is critical and you can afford extra RAM.
  - Use `map` if memory is tighter, or you need sorted iteration.
- 

Do you want me to also show you how to **tune `unordered_map` load factor and bucket count** to balance speed vs memory?

Great question — this is a classic “**Top-K Frequent Elements**” problem. Let’s walk through it step by step, focusing on efficiency and trade-offs.

---

## Approach

1. **Count frequencies with a HashMap**
    - Iterate through the array once.
    - Use a hash map (`unordered_map<int, int>`) to store frequency of each number.
    - Time complexity: **O(n)**, where  $n$  is the size of the array.
    - Space complexity: **O(u)**, where  $u$  is the number of unique integers.
  2. **Use a Min-Heap (size = k = 10) to track top frequencies**
    - Iterate over the hash map entries.
    - Push each (`frequency, number`) pair into a min-heap (priority queue).
    - If heap size > 10, pop the smallest.
    - At the end, the heap contains the 10 most frequent numbers.
    - Time complexity: **O(u log k)**. With  $k = 10$ , this is effectively **O(u)**.
-

## C++ Implementation

```
#include <bits/stdc++.h>
using namespace std;

vector<pair<int,int>> topKFrequent(const vector<int>& nums, int k) {
    unordered_map<int,int> freq;
    for (int num : nums) freq[num]++;
    using P = pair<int,int>; // (frequency, number)
    auto cmp = [] (const P& a, const P& b){ return a.first > b.first; };
    priority_queue<P, vector<P>, decltype(cmp)> minHeap(cmp);

    for (auto& [num, count] : freq) {
        minHeap.push({count, num});
        if ((int)minHeap.size() > k) minHeap.pop();
    }

    vector<pair<int,int>> result;
    while (!minHeap.empty()) {
        result.push_back(minHeap.top());
        minHeap.pop();
    }
    reverse(result.begin(), result.end()); // highest freq first
    return result;
}

int main() {
    vector<int> nums = {1,1,1,2,2,3,3,3,3,4,5,5,5,5,5,6};
    int k = 10;
    auto topK = topKFrequent(nums, k);

    cout << "Top " << k << " frequent numbers:\n";
    for (auto& [freq, num] : topK) {
        cout << num << " -> " << freq << " times\n";
    }
}
```

---

## Memory Usage vs Streaming Approach

- **HashMap + Heap (in-memory)**
    - Pros: Single pass over the array, exact results.
    - Cons: Requires memory proportional to unique elements ( $O(u)$ ). If you have millions of unique numbers, memory grows.
  - **Streaming / Approximate Approaches (if memory is tight or data is unbounded)**
    - **Count-Min Sketch** or **Reservoir Sampling** can approximate top-K with bounded memory.
    - Useful for streams where you can't store all counts.
    - Trade-off: You get *probabilistic results* (slight error) instead of exact.
-

- For “millions of integers” where the number of *unique* values is manageable, **HashMap + Min-Heap** is the best choice.
  - For unbounded streams or extremely high cardinality, **streaming algorithms (Count-Min Sketch, Misra-Gries)** are the practical choice.
- 

Do you want me to also show a **streaming version (Misra-Gries algorithm)** in C++ so you can compare exact vs approximate?