**========================🡺Graph Algorithms===============================**

* **BFS on unweighted graph → gives shortest path using Queue**
* **DFS uses Stack (recursion )**
* **Adjacency Matrix (2D array) for undirected graph is symmetric🡺Adjacency matrix is bad for sparse graphs (wastes space). 🡺Checking edge between nodes is O(1)🡺** **Space = O(n²)**
* **Complete graph with n vertices → n(n–1)/2 edges**
* **Sum of degrees = In undirected graph = 2 × Number of edges(e)🡺2\*e (Degree sum is always 2e (twice the edge count)**
* **graph representation is best for sparse graph🡺** **Adjacency List**
* **data structure is used by Map? C. Balanced Binary Tree**

**✅ Part 2: Types of Graphs**

| **Type** | **Explanation** |
| --- | --- |
| **Undirected Graph** | **Edges have no direction (A—B)** |
| **Directed Graph (Digraph)** | **Edges have direction (A → B)** |
| **Weighted Graph** | **Edges carry a cost/weight** |
| **Unweighted Graph** | **All edges are equal (used in BFS for shortest)** |
| **Cyclic Graph** | **Contains at least one cycle** |
| **Acyclic Graph** | **No cycles (DAG = Directed Acyclic Graph)** |
| **Connected Graph** | **There is a path between every pair of nodes** |
| **Disconnected Graph** | **Not all nodes are reachable** |
| **Complete Graph (Kn)** | **Every node is connected to every other node** |

**Topological Sort Valid for DAG only**

**🔚 Summary Table:**

| **Task** | **DFS** | **BFS** |
| --- | --- | --- |
| **Data Structure Used** | **Stack** | **Queue** |
| **Finds Shortest Path (Unweighted)** | **❌** | **✅** |
| **Recursive** | **✅** | **❌** |
| **Level Order Traversal** | **❌** | **✅** |

**MCQs**

**How many edges in a complete graph with 7 vertices?**

**A. 21  
B. 49  
C. 14  
D. 42**

**Sum of degrees in a graph with 10 vertices and 15 edges is?**

**A. 30  
B. 15  
C. 10  
D. 25**

# Sorting Algorithms

**Types of Sorting Algorithms**

| **Algorithm** | **Time (Avg)** | **Time (Worst)** | **Space** | **Stable** | **In-place** |
| --- | --- | --- | --- | --- | --- |
| **Bubble Sort** | **O(n²)** | **O(n²)** | **O(1)** | **✅** | **✅** |
| **Insertion Sort** | **O(n²)** | **O(n²)** | **O(1)** | **✅** | **✅** |
| **Selection Sort** | **O(n²)** | **O(n²)** | **O(1)** | **❌** | **✅** |
| **Merge Sort** | **O(n log n)** | **O(n log n)** | **O(n)** | **✅** | **❌** |
| **Quick Sort** | **O(n log n)** | **O(n²)** | **O(log n)** | **❌** | **✅** |
| **Heap Sort** | **O(n log n)** | **O(n log n)** | **O(1)** | **❌** | **✅** |

**Insertion Sort 🡺** **Worst-case: O(n²) – when the array is reverse sorted**

**🡺** **Best-case: O(n) – already sorted array**

**🡺** **A list is already sorted except a few elements at the end. Best sorting insertion sort**

**Quick Sort – Divide & Conquer 🡺 Recursive call on subarrays**

**🡺 Worst-case = O(n²)**

**When? → When pivot is always the smallest/largest (e.g., sorted input)**

**🔹 Average-case = O(n log n)**

**Good for large datasets**

**Selection Sort**

* **Repeatedly picks the minimum element and places at beginning**
* **Always O(n²) comparisons**
* **Uses minimum number of swaps: At most (n – 1)**
* **uses the minimum number of swaps**

### Q5. If file is reverse sorted, how many comparisons does Insertion Sort make?

A. O(n)  
B. O(n log n)  
C. O(n²)  
D. O(1)

**Sorting Selection Strategy**

| **Situation** | **Best Algorithm** |
| --- | --- |
| Nearly sorted | Insertion Sort ✅ |
| Small data (≤ 20 elements) | Insertion / Bubble |
| Large unsorted data | Merge / Quick Sort |
| Minimal swaps required | Selection Sort ✅ |
| Memory constraint (O(1) space) | Heap Sort ✅ |
| Stable sort needed with consistent output | Merge / Insertion ✅ |

# 🔷 **Topic 3: Hashing**

**🔑 Goal:**

**To provide constant time access (O(1)) to data using a key.**

**✅ Part 2: Hash Function**

**A hash function h(k) maps a key k to an index in the hash table.**

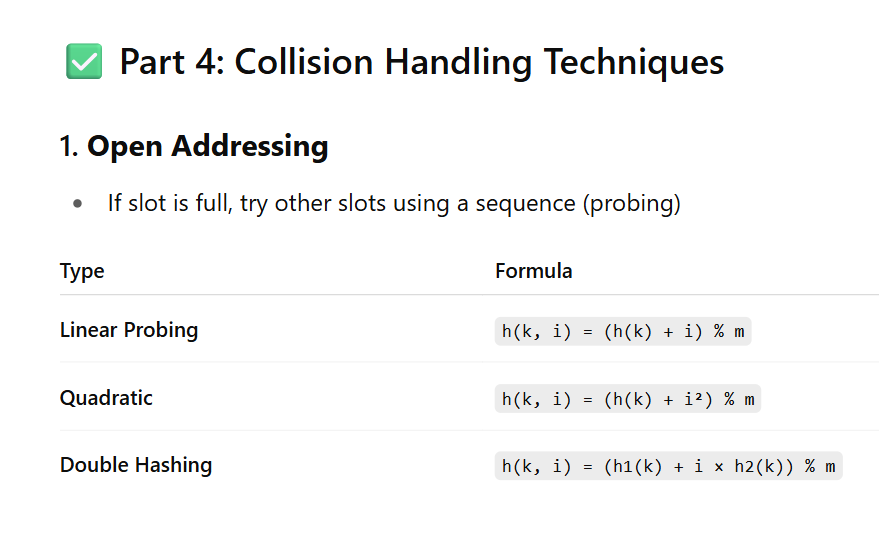
**🡺h(k) = k % table\_size;**

**example**

**If k = 103, and table\_size = 13,  
Then → h(103) = 103 % 13 = 12**

**✅ Part 3: Hash Table Insertion (Basic)**

1. **Compute hash: index = h(k)**
2. **If slot is empty, insert**
3. **If slot is occupied, handle the collision**

****

2. Chaining

* Each slot holds a linked list of entries
*  Use the chaining method: Chaining is a collision resolution technique, not a prevention technique. When a collision occurs, chaining links the keys with the same hash index in a linked list. It handles collisions but doesn't prevent them.

**🧠 Summary Points:**

* **Hash table gives O(1) average lookup**
* **Collisions are resolved via Probing or Chaining**
* **Linear probing causes primary clustering**
* **Double Hashing reduces clustering**
* **Hash functions must distribute keys uniformly**

**Clustering in Hashing — Quick Recap**

**There are two types of clustering that reduce performance in open addressing:**

| **Clustering Type** | **Description** |
| --- | --- |
| **Primary** | **When many keys hash to the same or nearby slots, forming a large cluster.** |
| **Secondary** | **When keys hash to the same initial index and follow same probe sequence.** |

**Note-**

**Chaining doesn't use probing at all, so:**

* **✅ It completely avoids clustering — no primary, no secondary**

### Double Hashing Formula:

To resolve the collision:

ini

Copy code

next\_index = (h1(k) + i \* h2(k)) % table\_size

Where i is the collision attempt number (starting from 1 if first probe failed).

**MCQs**

**Q1. Suppose you use double hashing on a table of size 10 with:**

* **h1(k) = k % 10**
* **h2(k) = 7 – (k % 7)  
  Where will key 77 be placed if index 7 and 4 are filled?**

**A. 1  
B. 0  
C. 2  
D. 3**

**Q2. Which hashing method suffers from secondary clustering?**

**A. Linear Probing  
B. Quadratic Probing  
C. Double Hashing  
D. Chaining**

**Q3. You have a hash table of size 10 and insert the keys: 35, 25, 15 using k % 10. If linear probing is used, what will the table look like?**

**A. [15, 25, 35]  
B. [\_\_, \_\_, \_\_, \_\_, \_\_, 15, 25, 35, \_\_, ]  
C. [, \_\_, \_\_, \_\_, \_\_, 35, 25, 15, \_\_, ]  
D. [, \_\_, \_\_, \_\_, \_\_, 15, 25, \_\_, \_\_, 35]**

**Q4. In the multiplication method, the value of A = (√5 – 1)/2 is preferred because:**

**A. It gives maximum collisions  
B. It minimizes collisions by spreading keys uniformly  
C. It's easier to compute  
D. It's used for chaining only**

**Q5. Which of the following is not a valid advantage of chaining?**

**A. Less affected by load factor  
B. Constant time lookup in worst case  
C. No clustering  
D. Easy deletion**

**Q6. You insert the keys 21, 32, 43 into a hash table of size 5 using k % 5. What will be the final table using quadratic probing?**

**A. [43, \_\_, 21, 32, \_\_]  
B. [21, 32, 43, \_\_, \_\_]  
C. [21, \_\_, 32, \_\_, 43]  
D. [21, 43, 32, \_\_, \_\_]**

**Q7. A hash table of size m = 100 has a load factor α = 0.6. How many keys are inserted?**

**A. 60  
B. 6  
C. 40  
D. 600**

**Q8. Which of the following best reduces both primary and secondary clustering?**

**A. Linear probing  
B. Quadratic probing  
C. Chaining  
D. Double hashing**

**Q9. In a hash table using linear probing, what happens if load factor becomes very high (close to 1)?**

**A. More collisions, performance drops  
B. Better performance  
C. Load factor doesn't affect probing  
D. Table becomes read-only**

**Q10. You’re using double hashing with:**

* **h1(k) = k % 11**
* **h2(k) = 1 + (k % 9)  
  What is the second probe index for key k = 100 if index 1 is taken?**

**A. 3  
B. 10  
C. 9  
D. 5**

**✅ Answers + Explanations**

**Q1 → A. 1**

* **h1(77) = 7,**
* **h2(77) = 7 – 0 = 7**
* **Index 7 filled → try (7 + 1×7) % 10 = 14 % 10 = 4 → filled**
* **Try (7 + 2×7) % 10 = 21 % 10 = \*\*1\*\* ✅**

**Q2 → B. Quadratic Probing**

* **Quadratic avoids primary clustering, but same distance hash causes secondary clustering. ✅**

**Q3 → B. [\_\_, \_\_, \_\_, \_\_, \_\_, 35,25,15 \_\_, \_\_]**

* **35 % 10 = 5 → insert at 5**
* **25 % 10 = 5 → taken → next = 6**
* **15 % 10 = 5 → taken → 6 taken → next = 7 ✅**

**Q4 → B. Minimizes collisions**

**Using irrational A like (√5 – 1)/2 ensures better distribution of keys across table ✅**

**Q5 → B. Constant time lookup in worst case**

**In chaining, worst-case = length of list → O(n), not constant ❌**

**Q6 →**

* **21 → 1st attempt: 21 % 5 = 1 → index 1**
* **32 → 32 % 5 = 2 → index 2**
* **43 → 43 % 5 = 3 → index 3  
  ✅ So final: [21, 32, 43, ]**

**Q7 → A. 60**

**α = n/m → 0.6 = n/100 → n = 60**

**Q8 → D. Double hashing**

**Most effective against both clustering types.**

**Q9 → A. More collisions, performance drops**

**At high load, more keys compete → more linear scans → O(n) performance ❌**

**Ques 10] Step-by-step calculation:**

**✅ Step 1: Calculate h1(k)**

**h1(100) = 100 % 11 = 1**

**So, the first probe is at index 1.**

**But index 1 is already taken.**

**✅ Step 2: Calculate h2(k)**

**h2(100) = 1 + (100 % 9) = 1 + 1 = 2**

**📌 Double Hashing Formula:**

**To resolve the collision**

**next\_index = (h1(k) + i \* h2(k)) % table\_size**

**Where i is the collision attempt number (starting from 1 if first probe failed).**

**🔁 Now let’s compute the second probe (i = 1):**

**next\_index = (1 + 1×2) % 11 = 3 % 11 = 3**

**So, second probe index is 3.**

**✅ If index 3 is free, then key 100 will be inserted at index 3.**

**✅ 📘 Hashing – Summary & Formula Sheet**

**📌 What is Hashing?**

**Hashing is a technique to map keys to indexes in a hash table using a hash function.**

**Used in:**

* **Maps (unordered\_map in C++)**
* **Symbol tables**
* **Sets**
* **Memory lookup**
* **Database indexing**

**🧮 1. Hash Function Basics**

**General form:**

**perl**

**Copy code**

**h(k) = key → index**

**🎯 Good Hash Function should:**

* **Minimize collisions**
* **Distribute keys uniformly**
* **Be fast to compute**

**🔢 2. Common Hash Functions**

**➤ a) Division Method:**

**matlab**

**Copy code**

**h(k) = k % m**

* **m should be prime, not power of 2, to reduce clustering.**

**➤ b) Multiplication Method:**

**matlab**

**Copy code**

**h(k) = floor(m \* ((k × A) % 1))**

* **A good choice: A ≈ (√5 – 1) / 2 ≈ 0.618033**

**🔁 3. Collision Resolution Techniques**

**A. Open Addressing (single array):**

**➤ Linear Probing:**

**matlab**

**Copy code**

**h(k, i) = (h(k) + i) % m**

* **Problem: Primary clustering**

**➤ Quadratic Probing:**

**matlab**

**Copy code**

**h(k, i) = (h(k) + c1·i + c2·i²) % m**

* **Less clustering, but secondary clustering exists**
* **Need to choose c1, c2 carefully**

**➤ Double Hashing:**

**matlab**

**Copy code**

**h(k, i) = (h1(k) + i·h2(k)) % m**

* **Requires:**
  + **h2(k) ≠ 0**
  + **h2(k) should not share factors with m**

**✅ Best for avoiding both primary & secondary clustering.**

**B. Chaining (using linked lists per bucket):**

* **Each index points to a list of elements**
* **Easy insertion, deletion**
* **Handles load factor > 1**
* **Slightly more memory**

**📐 4. Load Factor (α)**

**Copy code**

**α = n / m**

* **n = number of keys inserted**
* **m = size of hash table**

**📌 Lower α = fewer collisions**

**🚦 5. Clustering Summary**

| **Method** | **Primary Clustering** | **Secondary Clustering** |
| --- | --- | --- |
| **Linear Probing** | **✅ Yes** | **✅ Yes** |
| **Quadratic Probing** | **❌ No** | **✅ Yes** |
| **Double Hashing** | **❌ No** | **❌ No ✅** |
| **Chaining** | **❌ No** | **❌ No** |

**🔍 6. Time Complexity (Average Case)**

| **Operation** | **Hash Table (with good hashing)** |
| --- | --- |
| **Insert** | **O(1)** |
| **Delete** | **O(1)** |
| **Search** | **O(1)** |

**Worst case can be O(n) if hash function is poor or table is full.**

**✍️ 7. Example: Double Hashing**

**Given:**

* **m = 10**
* **h1(k) = k % 10**
* **h2(k) = 7 – (k % 7)**
* **Insert key k = 77**

**h1(77) = 7**

**h2(77) = 7 – (77 % 7) = 7 – 0 = 7**

**Probes:**

**1st → (7 + 0×7) % 10 = 7**

**2nd → (7 + 1×7) % 10 = 14 % 10 = 4**

**3rd → (7 + 2×7) % 10 = 21 % 10 = 1**

**...**

**🎓 Bonus: Hashing in STL (C++)**

| **Container** | **Uses Hashing?** | **Backing Structure** |
| --- | --- | --- |
| **unordered\_map** | **✅ Yes** | **Hash Table (Chaining)** |
| **unordered\_set** | **✅ Yes** | **Hash Table** |
| **map** | **❌ No** | **Balanced BST (Red-Black Tree)** |
| **set** | **❌ No** | **Balanced BST** |

** In a hash table, data is stored in buckets/slots determined by a hash function.**

** When two or more keys hash to the same index, it’s called a collision.**

** Collisions are common due to the finite size of the hash table.**

** Various techniques like chaining, linear probing, quadratic probing, or double hashing are used to resolve collisions.**

**Rehashing usually means increasing table size and re-applying the hash function, not just handling collisions during insertion.**

**✅ C++ STL + Core Concepts – Ultimate Summary Sheet**

**🔹 What is STL?**

**STL = Standard Template Library  
It provides:**

| **Component** | **Description** | **Examples** |
| --- | --- | --- |
| **Containers** | **Data structures** | **vector, set, map** |
| **Algorithms** | **Predefined operations** | **sort(), find()** |
| **Iterators** | **Pointers to access containers** | **begin(), end()** |
| **Function Objects** | **Custom functions used in STL** | **greater<>(), lambdas** |

**📦 STL Containers (with Time Complexities)**

**1. Sequence Containers (store in order)**

| **Container** | **Insert End** | **Insert Front** | **Access by Index** | **Use Case** |
| --- | --- | --- | --- | --- |
| **vector** | **O(1)\*** | **❌ O(n)** | **✅ O(1)** | **Dynamic arrays** |
| **deque** | **✅ O(1)** | **✅ O(1)** | **✅ O(1)** | **Fast front & back insertions** |
| **list** | **✅ O(1)** | **✅ O(1)** | **❌ O(n)** | **Frequent middle/front inserts** |

**Note: \* amortized constant time**

**2. Associative Containers (sorted, unique keys)**

| **Container** | **Key-Value** | **Sorted** | **Duplicates** | **Lookup** |
| --- | --- | --- | --- | --- |
| **set** | **❌** | **✅ Yes** | **❌ No** | **O(log n)** |
| **map** | **✅** | **✅ Yes** | **❌ No** | **O(log n)** |
| **multiset** | **❌** | **✅** | **✅ Yes** | **O(log n)** |
| **multimap** | **✅** | **✅** | **✅ Yes** | **O(log n)** |

**Based on Red-Black Tree**

**3. Unordered Containers (fast, no order)**

| **Container** | **Key-Value** | **Sorted** | **Duplicates** | **Lookup** | **Based On** |
| --- | --- | --- | --- | --- | --- |
| **unordered\_set** | **❌** | **❌ No** | **❌** | **O(1) avg** | **Hash Table** |
| **unordered\_map** | **✅** | **❌** | **❌** | **O(1) avg** | **Hash Table** |
| **unordered\_multiset** | **❌** | **❌** | **✅** | **O(1) avg** | **Hash Table** |

**🔥 Faster than set / map but not ordered**

**4. Container Adapters**

| **Adapter** | **Built On** | **Operations Allowed** |
| --- | --- | --- |
| **stack** | **deque/list** | **push, pop (LIFO)** |
| **queue** | **deque/list** | **push, pop (FIFO)** |
| **priority\_queue** | **vector** | **push, top (max by default)** |

**🔁 Iterators**

| **Type** | **Containers** | **Move Backward?** |
| --- | --- | --- |
| **Input Iterator** | **istream, algorithms** | **❌** |
| **Output Iterator** | **ostream, output** | **❌** |
| **Forward Iterator** | **forward\_list** | **❌** |
| **Bidirectional** | **list, set, map** | **✅** |
| **Random Access** | **vector, deque** | **✅✅✅** |

**Use with auto, begin(), end():**

**cpp**

**Copy code**

**for (auto it = v.begin(); it != v.end(); ++it)**

**cout << \*it;**

**🔍 Common STL Algorithms**

| **Function** | **Purpose** |
| --- | --- |
| **sort()** | **Sort container (O(n log n))** |
| **reverse()** | **Reverse elements** |
| **find()** | **Linear search (use in vector)** |
| **binary\_search()** | **Requires sorted container** |
| **count()** | **Count occurrences** |
| **lower\_bound()** | **First element ≥ x** |
| **upper\_bound()** | **First element > x** |

**⚡ Time Complexity Summary**

| **Operation** | **vector** | **list** | **set / map** | **unordered\_map** |
| --- | --- | --- | --- | --- |
| **Insertion (end)** | **O(1)** | **O(1)** | **O(log n)** | **O(1)\*** |
| **Access by index** | **O(1)** | **❌** | **O(log n)** | **O(1)\*** |
| **Search** | **O(n)** | **O(n)** | **O(log n)** | **O(1)\*** |
| **Front Insert** | **O(n)** | **O(1)** | **❌** | **❌** |

**\*Average case; Worst-case O(n) if hash collisions**

**🔥 MCQ Trick Examples from STL**

**Q1. Which STL is best to insert and remove from both ends quickly?  
✅ Answer: deque**

**Q2. Which container stores key-value pairs in sorted order?  
✅ Answer: map**

**Q3. Which container allows duplicates and keeps sorted?  
✅ Answer: multiset**

**Q4. Which of the following is random access: list, deque, set?  
✅ Answer: deque**

**Q5. What happens if we access a missing key in map using []?  
✅ \*\*Answer: It creates a new entry with default value**

**💡 STL Usage Tips**

* **✅ Use unordered\_map for fast access when sorting is not needed**
* **✅ Use set when order matters and duplicates not allowed**
* **✅ Use multiset or multimap for duplicate keys**
* **✅ Use list for frequent middle/front insert/delete**
* **❌ Avoid vector for large front insertions**

**Would you like to follow this with:**

* **🔁 20 mixed STL MCQs,**
* **💻 STL-based code practice,**
* **or go to Sorting/Graph next?**

**Let’s master it step-by-step 🚀**

**You said:**

**🔁 \*\*20 mixed STL MC\*\***

**this**

**ChatGPT said:**

**Perfect! Let's tackle 🔁 20 Mixed STL MCQs — covering tricky and exam-focused questions on vector, set, map, unordered\_map, list, deque, and related STL concepts.**

**Each MCQ includes Answer ✅ + Explanation.**

**✅ 20 Mixed STL MCQs with Explanation**

**1. What is the default sorting order of std::set in C++?**

**A. Insertion Order  
B. Descending  
C. Ascending  
D. No order**

**✅ Correct: C. Ascending  
👉 set automatically stores unique elements in ascending order using std::less.**

**2. Which STL container allows random access using indexing?**

**A. map  
B. list  
C. vector  
D. set**

**✅ Correct: C. vector  
👉 vector[2] is valid. list and map don’t support direct indexing.**

**3. Which container would you use to store key-value pairs with fast lookup and no sorting?**

**A. map  
B. vector  
C. unordered\_map  
D. deque**

**✅ Correct: C. unordered\_map  
👉 unordered\_map uses hashing and is fast (avg O(1)) but unsorted.**

**4. Which one of these allows duplicates?**

**A. set  
B. unordered\_set  
C. multiset  
D. map**

**✅ Correct: C. multiset  
👉 set and unordered\_set only allow unique elements.**

**5. Which of the following is true about list?**

**A. Supports random access  
B. Insertion at front is O(1)  
C. Memory is contiguous  
D. It's implemented as array**

**✅ Correct: B. Insertion at front is O(1)  
👉 list is doubly linked, so front/back insertions are fast. No random access.**

**6. What does v.capacity() return for a vector?**

**A. Number of elements stored  
B. Size in bytes  
C. Maximum size allowed  
D. Number of allocated slots**

**✅ Correct: D. Number of allocated slots  
👉 capacity() returns how many elements it can hold without reallocation.**

**7. Which container gives O(1) average-case search and insert?**

**A. map  
B. unordered\_map  
C. vector  
D. set**

**✅ Correct: B. unordered\_map  
👉 Based on hash tables, so operations are O(1) on average.**

**8. Which algorithm is used by priority\_queue in C++?**

**A. AVL Tree  
B. Heap  
C. Hash Table  
D. BST**

**✅ Correct: B. Heap  
👉 It is implemented as a max-heap by default.**

**9. Which iterator supports both forward and backward movement?**

**A. Forward Iterator  
B. Input Iterator  
C. Bidirectional Iterator  
D. Output Iterator**

**✅ Correct: C. Bidirectional Iterator  
👉 Used in list, set, map**

**10. Which of the following is true for unordered\_map?**

**A. Sorted by key  
B. Allows duplicate keys  
C. Slower than map  
D. Fast lookup (O(1) average)**

**✅ Correct: D. Fast lookup (O(1) average)**

**11. What will be printed?**

**cpp**

**Copy code**

**vector<int> v = {5, 10, 15};**

**cout << v.front();**

**A. 0  
B. 5  
C. 15  
D. Error**

**✅ Correct: B. 5  
👉 front() returns first element.**

**13. How are elements ordered in a map?**

**A. Insertion order  
B. Descending  
C. Sorted ascending by key  
D. Unordered**

**✅ Correct: C. Sorted ascending by key**

**14. What is output of this?**

**cpp**

**Copy code**

**map<int, string> m;**

**m[1] = "A";**

**m[2] = "B";**

**cout << m[3];**

**A. Error  
B. Null  
C. Empty string  
D. "C"**

**✅ Correct: C. Empty string  
👉 m[3] creates a new key with default value "".**

**15. Which container should be used for implementing browser history (Back/Forward)?**

**A. stack  
B. queue  
C. list  
D. deque**

**✅ Correct: D. deque  
👉 Allows moving in both directions efficiently.**

**16. Which header is required for unordered\_map?**

**A. <map>  
B. <unordered\_set>  
C. <unordered\_map>  
D. <hash\_map>**

**✅ Correct: C. <unordered\_map>**

**17. What is the output of this snippet?**

**cpp**

**Copy code**

**set<int> s = {5, 3, 9};**

**for (int x : s) cout << x << " ";**

**A. 5 3 9  
B. 3 5 9  
C. 9 5 3  
D. Compilation error**

**✅ Correct: B. 3 5 9  
👉 set automatically sorts in ascending order.**

**18. Which operation is invalid on list?**

**A. push\_front()  
B. pop\_back()  
C. sort()  
D. list[2]**

**✅ Correct: D. list[2]  
👉 list does not support random access.**

**19. How many elements can be inserted in a set with this code?**

**cpp**

**Copy code**

**set<int> s;**

**s.insert(1);**

**s.insert(1);**

**s.insert(2);**

**A. 2  
B. 3  
C. 1  
D. 0**

**✅ Correct: A. 2  
👉 set stores unique elements.**

**20. Which STL container is best for implementing undo functionality?**

**A. queue  
B. stack  
C. deque  
D. vector**

**✅ Correct: B. stack  
👉 Last-in-first-out = perfect for undo.**

**Summary: BFS vs DFS**

| **Feature** | **BFS** | **DFS** |
| --- | --- | --- |
| **Data Structure** | **Queue** | **Stack / Recursion** |
| **Use Case** | **Shortest path** | **Cycle detection** |
| **Order** | **Level-by-level** | **Deep first** |
| **Time Complexity** | **O(V + E)** | **O(V + E)** |
| **Space Complexity** | **O(V)** | **O(V)** |

**📌 Summary**

| **Dijkstra Feature** | **Description** |
| --- | --- |
| **Best for** | **Shortest path (positive weights)** |
| **Data Structure Used** | **Priority Queue (Min Heap)** |
| **Not suitable for** | **Graphs with negative weights** |
| **Uses** | **Greedy strategy** |

**🎯 Kruskal vs Prim — Summary Table (Updated)**

| **Feature** | **Kruskal** | **Prim** |
| --- | --- | --- |
| **Type** | **Edge-based** | **Vertex-based** |
| **Sorting Needed** | **✅ Yes (on edges)** | **❌ No** |
| **Data Structure** | **Disjoint Set / Union-Find** | **Min Heap (Priority Queue)** |
| **Best For** | **Sparse Graphs** | **Dense Graphs** |
| **Time** | **O(E log E)** | **O(E log V) with Min Heap** |
| **Graph Input** | **✅ Edge List: {u, v, weight}** | **✅ Adjacency List: adj[u] = {v, weight}** |

**📘 Bellman-Ford Algorithm – Summary Table**

| **Feature** | **Description** |
| --- | --- |
| **Purpose** | **Single-source shortest path with negative weight edges** |
| **Graph Type** | **Works on both Directed and Undirected graphs** |
| **Input Representation** | **✅ Edge List (each edge as {u, v, weight})** |
| **Works with Negative Weights?** | **✅ Yes** |
| **Detects Negative Cycle?** | **✅ Yes (only algorithm that does this)** |
| **Fails When?** | **❌ If asked to compute shortest paths in graph with negative weight cycle** |
| **Approach Type** | **Dynamic Programming + Relaxation** |
| **Edge Relaxation Count** | **V - 1 times (V = number of vertices)** |
| **Extra Check** | **One final pass to detect negative cycle** |
| **Time Complexity** | **O(V × E)** |
| **Space Complexity** | **O(V) (for distance array)** |
| **Better Than Dijkstra?** | **✅ If graph has negative weights** |
| **Slower Than Dijkstra?** | **✅ In general (Dijkstra: O(E log V), Bellman-Ford: O(VE))** |
| **Common Use Cases** | **- Detecting arbitrage in finance - Finding routes in weighted networks with penalties - Detecting inconsistencies in time graphs** |

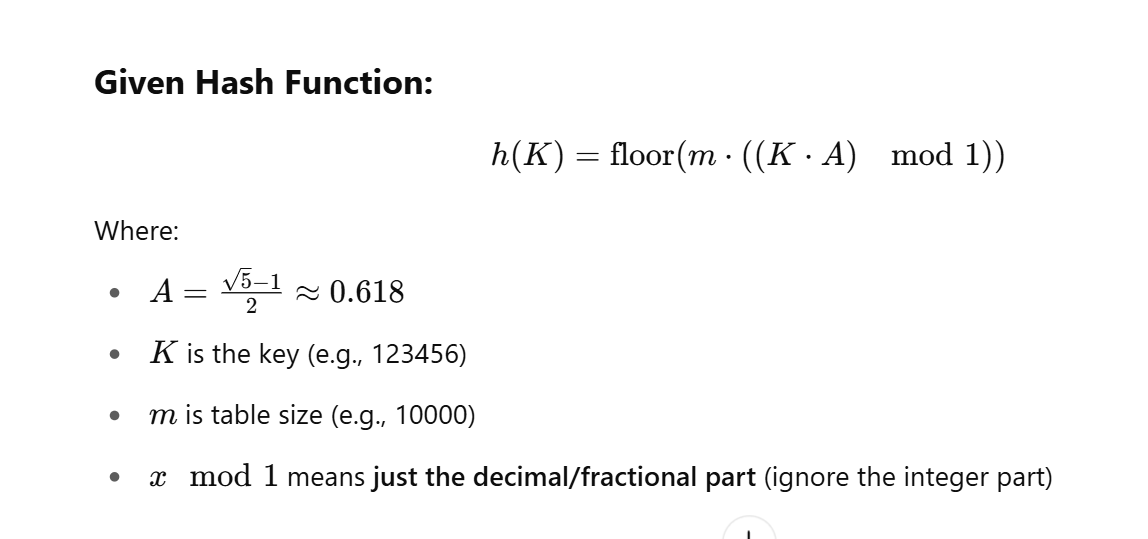
**🧠 Floyd-Warshall Algorithm – Detailed Conceptual Explanation**

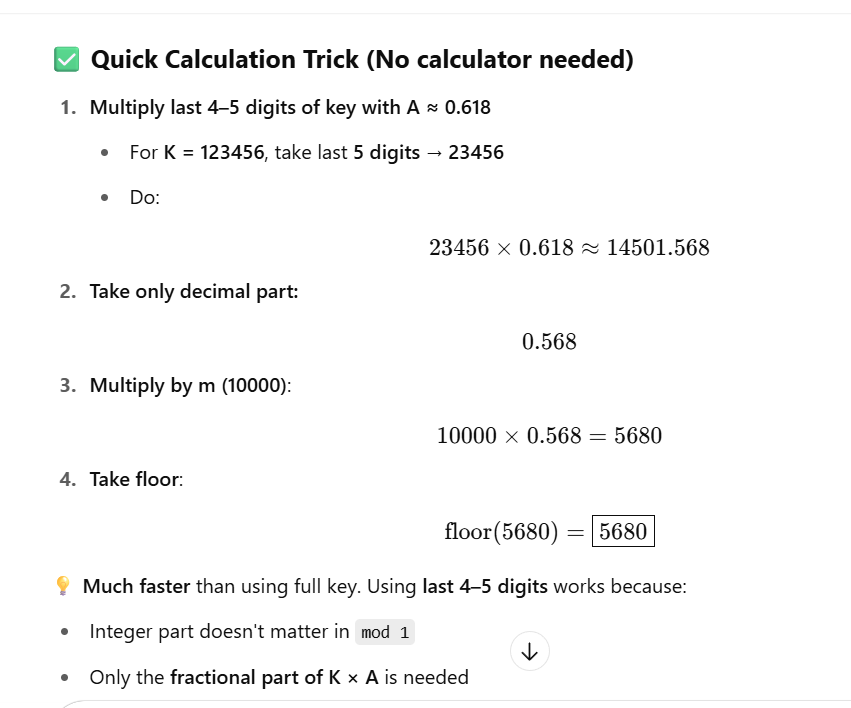
**🔍 What It Solves:**

**✅ All-Pairs Shortest Path (APSP) problem  
Given a directed weighted graph (with positive or negative weights), find the shortest path distance between every pair of vertices.**

**📌 Key Properties:**

| **Property** | **Description** |
| --- | --- |
| **Type of Graph** | **Directed or undirected; can include negative weights but no negative cycles** |
| **Works on** | **Weighted graphs (not unweighted like BFS or Dijkstra’s priority queue form)** |
| **Time Complexity** | **O(V³) (V = number of vertices)** |
| **Space Complexity** | **O(V²) for distance matrix** |
| **Based on** | **Dynamic Programming** |
| **Handles** | **Multiple edges, self-loops, negative weights** |
| **Doesn’t Work If** | **Graph contains negative weight cycles** |

****

****

**If m = 50 (like your original options were in 40s):**

**Try:**

**23456×0.618=14501.568⇒frac=0.568⇒50×0.568=28.4⇒floor=28**

**=============================NOTES================================**

**data structure is used by Map🡺Balanced Binary Tree🡺Red-Black Tree is a balanced BST, so this is accurate.**

**List – ✅ Clist is a doubly linked list.**

**🔷 STL Container Comparison: vector vs list vs deque**

| **Feature** | **vector** | **list** | **deque** |
| --- | --- | --- | --- |
| **Type** | **Dynamic array** | **Doubly linked list** | **Double-ended dynamic array** |
| **Random Access** | **✅ O(1)** | **❌ O(n)** | **✅ O(1)** |
| **Insertion at End** | **✅ O(1)\*** | **✅ O(1)** | **✅ O(1)** |
| **Insertion at Front** | **❌ O(n)** | **✅ O(1)** | **✅ O(1)** |
| **Insertion in Middle** | **❌ O(n)** | **✅ O(1)\*\*** | **❌ O(n)** |
| **Deletion at End** | **✅ O(1)** | **✅ O(1)** | **✅ O(1)** |
| **Deletion at Front** | **❌ O(n)** | **✅ O(1)** | **✅ O(1)** |
| **Deletion in Middle** | **❌ O(n)** | **✅ O(1)\*\*** | **❌ O(n)** |
| **Memory Overhead** | **Low (contiguous)** | **High (node pointers)** | **Medium** |
| **Cache Locality** | **Excellent (contiguous)** | **Poor (non-contiguous)** | **Good** |
| **Use for Stack?** | **✅ Yes** | **✅ Yes** | **✅ Yes** |
| **Use for Queue?** | **❌ No** | **✅ Yes** | **✅ Yes** |
| **Best For** | **Random access, end insert** | **Frequent insert/delete anywhere** | **Front/back operations** |

**🔸 Use-Case Summary:**

* **vector**
  + **Best when you need fast access to elements and insert/delete only at the end.**
  + **Example: Storing items for indexed lookup, dynamic arrays.**
* **list**
  + **Best when you need frequent insertions/deletions anywhere, but don’t care about random access.**
  + **Example: Managing items in a playlist, undo stack with random deletes.**
* **deque**
  + **Best when you need fast insertions/deletions at both ends, with some random access.**
  + **Example: Sliding window problems, double-ended queues.**

**Bellman-Ford algorithm be applied to weighted and directed graphs**

**Bellman-Ford Algorithm:**

* **It is used to find shortest paths from a single source to all other vertices.**
* **Unlike Dijkstra’s algorithm, Bellman-Ford can handle negative weights.**
* **Works for both positive and negative edge weights, as long as there are no negative cycles.**

**Sparse matrix will be commonly represented by: triplet representation, CSR, and CSC are the common methods**

**Branch and Bound, Dynamic Programming, or Approximation algorithms are preferred for Travelling Salesman Problem**

**Insertion Sort works by taking elements one by one and inserting them into their correct position in the already sorted part of the array.**

**🔹 In Best Case (already sorted):**

* **Comparisons ≈ N – 1**
* **Time Complexity: O(N)**

**🔹 In Worst Case (reverse order): N²**

**Quantitative Analysis deals with measurable metrics like:**

* **Scalability**
* **Load handling**
* **Performance benchmarking**
* **Throughput, latency**

**🔹 Qualitative Analysis deals with:**

* **Maintainability**
* **Reliability**
* **Usability**
* **Stability**
* **Code readability, structure**

**So together, they include:**

* **How well a system handles load, concurrency, and scales up (quantitative)**
* **And how stable, maintainable, or reliable it is (qualitative)**

**🔹 Selection Sort:**

* **Selection sort selects the minimum element in each pass and swaps it with the current index.**
* **It performs exactly (n – 1) swaps, regardless of the data.**

### **Q. A programmer wants to implement an Array class that should be able to store integer values and float values depending on how the class is instantiated. Which concept should be used?**

**Options:**

A. Function Overloading  
B. Function Template  
C. Class Template  
D. Virtual Function

### ✅ **Correct Answer: C. Class Template**

### **Explanation:**

#### 🔹 Goal:

You want to **create a class** that works with **multiple data types** (like int, float, etc.).

This is exactly what **C++ templates** are designed for.

### 🔍 **Why Class Template?**

A **Class Template** allows you to define a **generic class** that can be instantiated with **any data type**.

**Q. Which one of the following is an application of Stack Data Structure?**

**Options:**

**A. Managing function calls  
B. The stock span problem  
C. Arithmetic expression evaluation  
D. All of the above**

**✅ Correct Answer: D. All of the above**

**Explanation:**

**Stacks are used in many problems and systems due to their LIFO (Last-In, First-Out) behavior.**

**🔹 Option A: Managing Function Calls ✅**

* **When a function is called, the system pushes the return address and local variables onto the call stack.**
* **On return, it pops the function context.**
* **Fundamental to function recursion and execution.**

**🔹 Option B: The Stock Span Problem ✅**

* **This classic problem uses a stack to track previous higher stock prices to efficiently compute span.**
* **Reduces time complexity from O(n²) to O(n) using stack.**

**🔹 Option C: Arithmetic Expression Evaluation ✅**

* **Used to evaluate:**
  + **Infix → Postfix/Prefix**
  + **Postfix/Prefix evaluation**
* **Stack holds operands and operators as the expression is parsed.**

**The concatenation of two lists is to be performed in O(1) time🡺Circular Singly Linked List**

** A. Singly Linked List  
→ No tail pointer by default. To find last node, you'd traverse ⇒ O(n)**

** B. Doubly Linked List  
→ Same as singly; unless a tail pointer is maintained, it's O(n).**

** D. Array Implementation  
→ Concatenation would involve copying all elements into a new array ⇒ O(n)**

**🔹 Other Related Notations:**

| **Notation** | **Meaning** | **Used For** |
| --- | --- | --- |
| **O(f(n))** | **Upper Bound** | **Worst-case** |
| **Ω(f(n))** | **Lower Bound** | **Best-case** |
| **Θ(f(n))** | **Tight Bound** | **Exact growth rate** |

**edges are there in a complete graph with n vertices A. (n(n – 1)) / 2**

** A. Tree – Not a data structure used to implement the algorithm.**

** C. Stack – Used in DFS, not suitable for shortest paths in unweighted graphs.**

** D. Heap – Used in Dijkstra’s algorithm for weighted graphs, to get the minimum distance efficiently using a priority queue / min-heap.**

**Q. Which of the following statements about an adjacency matrix is/are WRONG?**

**Statements:**

1. **Adjacency matrix takes O(n²) time to determine edges in graph G**
2. **One can easily determine whether there is an edge connecting two vertices**
3. **Adjacency matrix for undirected graph is asymmetric**
4. **None of the above**

**🔹 Statement 1: "Adjacency matrix takes O(n²) time to determine edges in graph G"**

* **Wrong wording: Determining *if there is an edge between two vertices* is O(1) in an adjacency matrix.**
* **However, to list all edges or traverse the entire matrix, yes, it would take O(n²) time.  
  ✅ So this statement is technically true, not wrong.**

**🔹 Statement 2: "One can easily determine whether there is an edge connecting two vertices"**

* **In adjacency matrix, you simply check if matrix[i][j] == 1 or > 0.  
  ✅ This is true and fast (O(1))**

**🔹 Statement 3: "Adjacency matrix for undirected graph is asymmetric"**

**❌ This is FALSE.**

* **For undirected graphs, the adjacency matrix is symmetric:**
  + **If there is an edge between i and j, then matrix[i][j] = matrix[j][i]**

[**Data Compression MCQ Questions & Answers - Letsfindcourse**](https://letsfindcourse.com/technical-questions/dsa-mcq/data-compression-mcq-questions-and-answers)