**Task3**

What is big o notation and memory notation ?

-simplified analysis of an algorithm’s effieciency

-types of measurement :

* Worst –case
* Best –case
* Average -case

### **1. Linear Time Complexity: Big O(n) Complexity**

Linear time complexity means that the running time of an algorithm grows linearly with the size of the input.

Example: an algorithm that [traverses through an array to find a specific element](https://www.geeksforgeeks.org/linear-search/):

### bool findElement(int arr[], int n, int key) { **for** (int i = 0; i < n; i++) { **if** (arr[i] == key) { **return** true; } } **return** false; }

### **2. Logarithmic Time Complexity: Big O(log n) Complexity**

Logarithmic time complexity means that the running time of an algorithm is proportional to the logarithm of the input size.

example, a [binary search algorithm](https://www.geeksforgeeks.org/binary-search/)  :

int binarySearch(int arr[], int l, int r, int x)  
{  
 **if** (r >= l) {  
 int mid = l + (r - l) / 2;  
 **if** (arr[mid] == x)  
 **return** mid;  
 **if** (arr[mid] > x)  
 **return** binarySearch(arr, l, mid - 1, x);  
 **return** binarySearch(arr, mid + 1, r, x);  
 }  
 **return** -1;  
}

### **3. Quadratic Time Complexity: Big O(n2) Complexity**

Quadratic time complexity means that the running time of an algorithm is proportional to the square of the input size.

example, a simple [bubble sort algorithm:](https://www.geeksforgeeks.org/bubble-sort/)

void bubbleSort(int arr[], int n)  
{  
 **for** (int i = 0; i < n - 1; i++) {  
 **for** (int j = 0; j < n - i - 1; j++) {  
 **if** (arr[j] > arr[j + 1]) {  
 swap(&arr[j], &arr[j + 1]);  
 }  
 }  
 }  
}

### **4. Cubic Time Complexity: Big O(n3) Complexity**

Cubic time complexity means that the running time of an algorithm is proportional to the cube of the input size.

example, a naive [matrix multiplication algorithm:](https://www.geeksforgeeks.org/matrix-multiplication/)

### void multiply(int mat1[][N], int mat2[][N], int res[][N]) { **for** (int i = 0; i < N; i++) { **for** (int j = 0; j < N; j++) { res[i][j] = 0; **for** (int k = 0; k < N; k++) res[i][j] += mat1[i][k] \* mat2[k][j]; } } }

### **5. Polynomial Time Complexity: Big O(nk) Complexity**

Polynomial time complexity refers to the time complexity of an algorithm that can be expressed as a polynomial function of the input size **n**. In Big **O** notation, an algorithm is said to have polynomial time complexity if its time complexity is **O(nk)**, where **k** is a constant and represents the degree of the polynomial.

### **6. Exponential Time Complexity: Big O(2n) Complexity**

Exponential time complexity means that the running time of an algorithm doubles with each addition to the input data set.

example, the problem of [generating all subsets of a set](https://www.geeksforgeeks.org/backtracking-to-find-all-subsets/)

### void generateSubsets(int arr[], int n) { **for** (int i = 0; i < (1 << n); i++) { **for** (int j = 0; j < n; j++) { **if** (i & (1 << j)) { cout << arr[j] << " "; } } cout << endl; } }

### **7.Factorial Time Complexity: Big O(n!) Complexity**

Factorial time complexity means that the running time of an algorithm grows factorially with the size of the input. This is often seen in algorithms that generate all permutations of a set of data.

example of a factorial time complexity algorithm, which generates all permutations of an array:

void permute(int\* a, int l, int r)  
{  
 **if** (l == r) {  
 **for** (int i = 0; i <= r; i++) {  
 cout << a[i] << " ";  
 }  
 cout << endl;  
 }  
 **else** {  
 **for** (int i = l; i <= r; i++) {  
 swap(a[l], a[i]);  
 permute(a, l + 1, r);  
 swap(a[l], a[i]); *// backtrack*  
 }  
 }  
}

Q2 Why map is fast in DS?

Fast Access: Maps provide efficient lookup and retrieval operations based on the key. By using an underlying data structure like a balanced binary search tree or a hash table, maps can achieve constant or logarithmic time complexity for common operations.

Q3 What is the shortest path algorithm in weighted graph?[[1]](#endnote-26012)

Dijkstra's Algorithm finds the shortest path between a given node (which is called the "source node") and all other nodes in a graph. This algorithm uses the weights of the edges to find the path that minimizes the total distance (weight) between the source node and all other nodes.

Q5 Difference between b- tree & binary tree & RB tree?

|  |  |  |
| --- | --- | --- |
|  | **B-tree** | **Binary tree** |
| 1. | In a B-tree, a node can have maximum ‘M'(‘M’ is the order of the tree) number of child nodes. | While in binary tree, a node can have maximum two child nodes or sub-trees. |
| 2. | B-tree is called a sorted tree as its nodes are sorted in inorder traversal. | While binary tree is not a sorted tree. It can be sorted in inorder, preorder, or postorder traversal. |
| 3. | B-tree has a height of log(M\*N) (Where ‘M’ is the order of tree and N is the number of nodes). | While binary tree has a height of log2(N) (Where N is the number of nodes). |
| 4. | B-Tree is performed when the data is loaded into the disk. | Unlike B-tree, binary tree is performed when the data is loaded in the RAM(faster memory). |
| 5. | B-tree is used in DBMS(code indexing, etc). | While binary tree is used in Huffman coding and Code optimization and many others. |
| 6. | To insert the data or key in B-tree is more complicated than a binary tree. | While in binary tree, data insertion is not more complicated than B-tree. |
| 7. | B-tree is a self-balancing tree. The height of the tree is automatically adjusted on each update. | A binary tree is not a self-balancing tree. |

#### RED-BLACK TREES

Red-black trees are an evolution of binary search trees that aim to keep the tree balanced without affecting the complexity of the primitive operations. This is done by coloring each node in the tree with either red or black and preserving a set of properties that guarantee that the deepest path in the tree is not longer than twice the shortest one.

A red-black tree is a binary search tree with the following properties:

1. Every node is colored with either red or black.
2. All leaf (nil) nodes are colored with black; if a node’s child is missing then we will assume that it has a nil child in that place and this nil child is always colored black.
3. Both children of a red node must be black nodes.
4. Every path from a node n to a descendent leaf has the same number of black nodes (not counting node n). We call this number the black height of n, which is denoted by bh(n).

Q6 Hash Map types, pros and cons?

Hash maps are indexed data structures. A hash map makes use of a [hash function](https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/) to compute an index with a key into an array of buckets or slots. Its value is mapped to the bucket with the corresponding index. The key is unique and immutable.

The hash map design will include the following functions:

* **set\_val(key, value):** Inserts a key-value pair into the hash map. If the value already exists in the hash map, update the value.
* **get\_val(key):** Returns the value to which the specified key is mapped, or “No record found” if this map contains no mapping for the key.
* **delete\_val(key):** Removes the mapping for the specific key if the hash map contains the mapping for the key.

**Time Complexity:**

Memory index access takes constant time and hashing takes constant time. Hence, the search complexity of a hash map is also constant time, that is, O(1).

**Advantages of HashMaps**

● Fast random memory access through hash functions

● Can use negative and non-integral values to access the values.

● Keys can be stored in sorted order hence can iterate over the maps easily.

**Disadvantages of HashMaps**

● Collisions can cause large penalties and can blow up the time complexity to linear.

● When the number of keys is large, a single hash function often causes collisions.

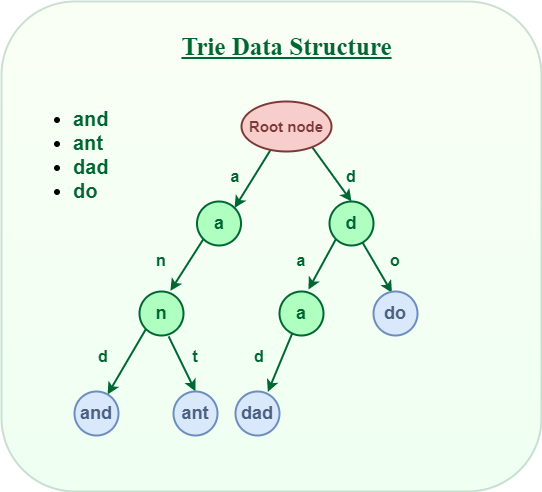
**Applications of HashMaps**

● These have applications in implementations of Cache where memory locations are mapped to small sets.

● They are used to index tuples in Database management systems.

● They are also used in algorithms like the Rabin Karp pattern matching

Q7 What is tries?

**Trie data structure** is a tree-like data structure used for storing a dynamic set of strings. It is commonly used for efficient **retrieval** and **storage** of keys in a large dataset . 

A **Trie data structure** consists of nodes connected by edges. Each node represents a character or a part of a string. The root node, the starting point of the Trie, represents an empty string. Each edge emanating from a node signifies a specific character. The path from the root to a node represents the prefix of a string stored in the Trie.

**struct** **TrieNode** {  
  
 *// pointer array for child nodes of each node*  
 TrieNode\* childNode[26];  
  
 *// Used for indicating ending of string*  
 bool wordEnd;  
  
 TrieNode()  
 {  
 *// constructor*  
 *// initialize the wordEnd variable with false*  
 *// initialize every index of childNode array with*  
 *// NULL*  
 wordEnd = false;  
 **for** (int i = 0; i < 26; i++) {  
 childNode[i] = NULL;  
 }  
 }  
}

1. [↑](#endnote-ref-26012)