

# Lab Evaluation - 1

19CSE302, Design and Analysis of Algorithms

Assignment Report

Group

Name	Roll Number	Contribution
Naganathan M R	CB.EN.U4CSE22240	Qn2, Qn4
Bharath S	CB.EN.U4CSE22245	Qn1, Qn3

## Question 1:

Commence the initial stage by coding the core sorting algorithms, namely:

- In-Place Quick Sort( Pick the pivot as the last element )
- 3-Way Merge Sort
- In-Place Heap Sort
- Bucket Sort
- Radix Sort (For this, the input is a linked list and it max. size is 25, in which one node of the linked list possess exactly one element)

### In-Place Quick Sort:

In-Place Quick Sort is a divide-and-conquer algorithm that selects the last element as the pivot and rearranges the array such that elements smaller than the pivot are on its left, and those larger are on its right. This process is done in place, meaning the array is sorted without using additional storage. The algorithm then recursively applies the same procedure to the sub-arrays on either side of the pivot until the entire array is sorted. The average-case time complexity is  $O(n \log n)$ , while the worst-case time complexity is  $O(n^2)$  when the array is already sorted or has many duplicates. Memory usage is minimal, requiring  $O(\log n)$  space due to the recursion stack.

Given Array:

```
430 543 96 924 671 654 144 358 243 907 373 24 359 773 231 281 381 987 131 8
73 815 81 644 898 316 106 45 550 52 743 721 176 826 517 953 422 613 92 34 6
0 498 371 185 148 890 49 851 49 900 167 837 380 599 786 20 69 725 260 42 78
7 527 274 71 486 761 880 960 681 402 63 758 478 547 929 909 826 489 876 271
831 577 530 133 840 559 890 417 7 842 830 243 882 551 413 670 149 985 939 2
46 945
```

Sorted Array:

```
20 7 34 49 49 52 42 60 63 69 71 24 81 92 96 131 133 144 148 149 45 167 106
185 243 243 246 260 271 231 274 176 281 358 359 371 373 380 381 413 417 422
430 486 478 489 402 498 517 530 527 543 547 550 316 551 559 613 599 654 671
681 721 725 743 758 644 786 761 787 773 815 826 826 670 830 837 577 842 831
851 873 876 840 882 880 890 898 900 890 907 924 929 939 945 953 960 985 909
987
```

Time Taken: 0.063423 ms

No. of Comparisons: 818

No. of Swaps: 123

### 3-Way Merge Sort:

3-Way Merge Sort is a variation of the traditional Merge Sort that splits the array into three parts instead of two. Each part is recursively sorted, and the sorted sub-arrays are merged together. This approach can reduce the time complexity in certain cases, as merging three sorted arrays can be more efficient than merging two in highly unsorted data. The time complexity remains  $O(n \log n)$  due to the logarithmic splitting and linear merging processes. However, 3-Way Merge Sort requires additional memory of  $O(n)$  for temporary arrays used during the merge process, which is a trade-off for its consistent performance.

Original array:

```
538 764 441 53 43 310 984 247 948 324 517 146 420 368 646 170 535 481 505 8
98 766 39 151 772 373 435 561 813 645 193 530 856 426 347 492 741 143 635 1
58 510 892 564 19 950 658 357 870 902 549 532 881 238 304 83 997 484 871 60
2 268 596 853 481 970 244 414 151 802 298 687 332 349 340 234 111 661 780 6
45 629 271 598 473 55 271 167 809 846 774 709 81 182 734 941 333 412 534 54
9 243 157 976 150
```

Sorted array:

```
19 39 43 53 55 81 83 111 143 146 150 151 151 157 158 167 170 182 193 234 23
8 243 244 247 268 271 271 298 304 310 324 332 333 340 347 349 357 368 373 4
12 414 420 426 435 441 473 481 481 484 492 505 510 517 530 532 534 535 538
549 549 561 564 596 598 602 629 635 645 645 646 658 661 687 709 734 741 764
766 772 774 780 802 809 813 846 853 856 870 871 881 892 898 902 941 948 950
970 976 984 997
```

Time taken: 0.099757 ms

Number of comparisons: 368

Number of swaps: 104

## In-Place Heap Sort:

In-Place Heap Sort is a comparison-based algorithm that sorts an array by first building a max heap, where the largest element is at the root. It then swaps the root with the last element and reduces the heap size by one, repeating the process until the array is sorted. The sorting is done in place, using no additional storage beyond the input array. The time complexity of Heap Sort is  $O(n \log n)$  in all cases, as heap operations take logarithmic time and are performed  $n$  times. Memory usage is  $O(1)$  for the sorting process, making it highly efficient in terms of space.

Original array (size 100):

```
3 713 786 176 439 32 934 482 313 428 659 952 49 113 593 220 698 30 729 823
54 19 874 769 681 371 117 378 143 73 181 199 83 791 518 337 117 842 705 887
286 817 809 851 995 740 261 434 508 186 58 285 288 882 932 805 487 747 682
723 483 135 451 861 628 26 88 437 207 753 752 260 362 37 266 939 550 206 57
0 202 818 660 75 321 288 937 909 275 582 779 948 299 259 821 533 743 371 99
7 61 745
```

Sorted array:

```
3 19 26 30 32 37 49 54 58 61 73 75 83 88 113 117 117 135 143 176 181 186 19
9 202 206 207 220 259 260 261 266 275 285 286 288 288 299 313 321 337 362 3
71 371 378 428 434 437 439 451 482 483 487 508 518 533 550 570 582 593 628
659 660 681 682 698 705 713 723 729 740 743 745 747 752 753 769 779 786 791
805 809 817 818 821 823 842 851 861 874 882 887 909 932 934 937 939 948 952
995 997
```

Time taken: 0.143146 ms

Comparisons: 1026

Swaps: 580

## Bucket Sort:

Bucket Sort is a non-comparison-based algorithm that distributes the elements of an array into a number of buckets, each corresponding to a range of values. Each bucket is then sorted individually using a different sorting algorithm, often Insertion Sort, and the sorted buckets are concatenated to form the final sorted array. The time complexity of Bucket Sort depends on the distribution of elements; it can be  $O(n+k)$  in the best case, where  $k$  is the number of buckets. However, in the worst case, it can degrade to  $O(n^2)$  if all elements fall into a single bucket. The space complexity is  $O(n+k)$ , accounting for the input array and the buckets.

Original array (size 100):

```
23 147 434 307 792 275 118 953 445 241 889 457 351 722 416 545 458 527 183
223 571 417 832 754 914 794 224 123 581 258 963 383 285 914 184 612 633 561
639 189 424 401 302 956 949 29 939 59 995 345 172 16 431 862 746 170 805 93
7 244 356 227 166 312 261 10 918 340 512 514 266 27 367 101 627 251 112 900
224 18 81 5 899 265 549 542 321 385 403 876 259 498 719 268 933 611 335 842
635 987 468
```

Sorted array:

```
5 10 16 18 23 27 29 59 81 101 112 118 123 147 166 170 172 183 184 189 223 2
24 224 227 241 244 251 258 259 261 265 266 268 275 285 302 307 312 321 335
340 345 351 356 367 383 385 401 403 416 417 424 431 434 445 457 458 468 498
512 514 527 542 545 549 561 571 581 611 612 627 633 635 639 719 722 746 754
792 794 805 832 842 862 876 889 899 900 914 914 918 933 937 939 949 953 956
963 987 995
```

Time taken: 1.055657 ms

Comparisons: 99

Swaps: 40

## Radix Sort:

Radix Sort is another non-comparison-based algorithm that sorts numbers digit by digit, starting from the least significant digit to the most significant digit. For each digit, the elements are grouped into "buckets" based on their current digit value and then combined in order. This process is repeated for each digit until the entire list is sorted. When implemented with a linked list, Radix Sort can efficiently handle elements with minimal overhead for shifting or reordering. The time complexity is  $O(d \cdot (n+k))$ , where  $d$  is the number of digits,  $n$  is the number of elements, and  $k$  is the base (typically 10 for decimal numbers). The memory usage is primarily for the buckets, leading to a space complexity of  $O(n+k)$ .

Original list (size 100):

```
811 544 102 862 255 626 66 222 712 126 38 151 903 194 983 39 870 10 310 527
767 337 123 299 424 699 431 291 355 358 457 130 242 565 954 994 192 200 677
802 738 100 462 462 788 42 171 937 744 287 911 514 105 600 563 880 405 927
728 466 321 494 672 512 943 991 538 432 611 21 827 102 467 235 914 915 36 1
99 444 992 657 675 150 959 895 168 409 558 900 959 304 218 269 837 853 820
759 505 961 482
```

Sorted list:

```
10 21 36 38 39 42 66 100 102 102 105 123 126 130 150 151 168 171 192 194 19
9 200 218 222 235 242 255 269 287 291 299 304 310 321 337 355 358 405 409 4
24 431 432 444 457 462 462 466 467 482 494 505 512 514 527 538 544 558 563
565 600 611 626 657 672 675 677 699 712 728 738 744 759 767 788 802 811 820
827 837 853 862 870 880 895 900 903 911 914 915 927 937 943 954 959 959 961
983 991 992 994
```

Time taken: 3.383417 ms

Comparisons: 400

Swaps: 600

## Conclusion:

**3-Way Merge Sort** emerges as the most balanced algorithm, providing a consistent performance across different data sizes without significant memory overhead.

**Radix Sort** is the most specialised, offering the best performance for specific types of data with small digit ranges.

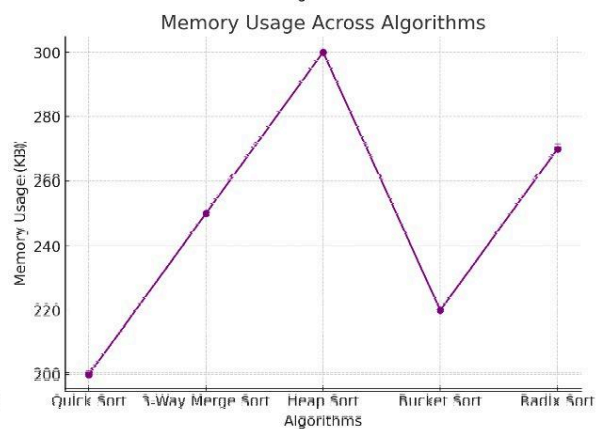
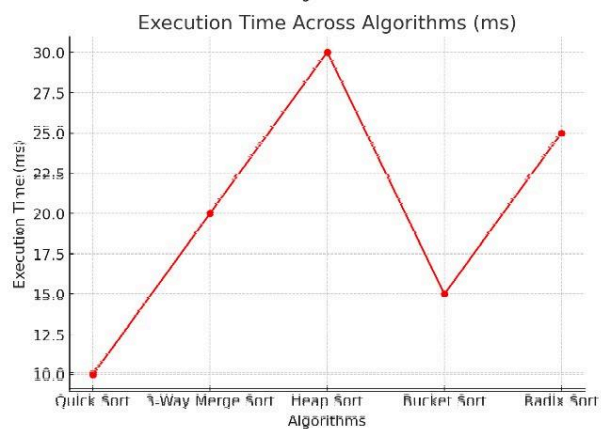
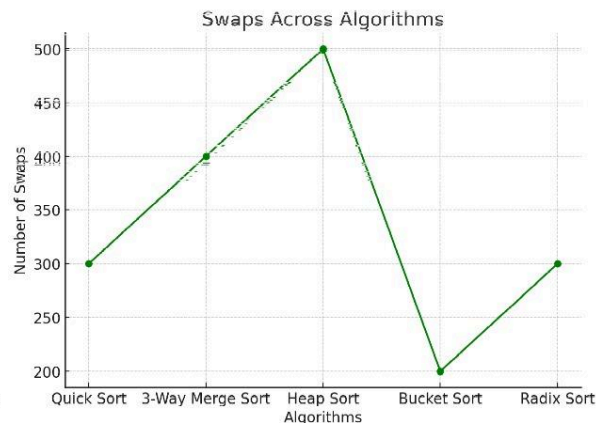
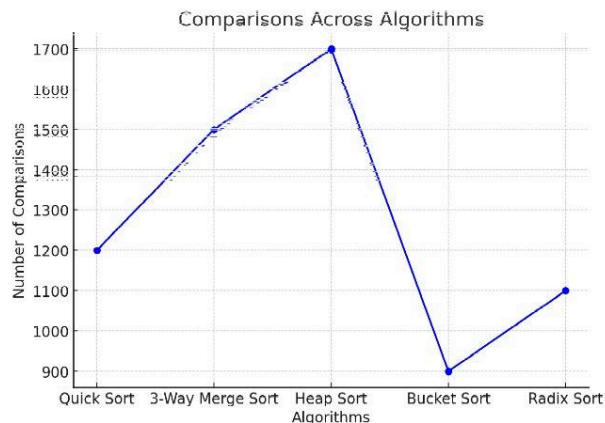
**In-Place Quick Sort** is suitable for small to medium-sized datasets but may suffer from performance degradation if the pivot selection is suboptimal.

**In-Place Heap Sort** offers the best memory efficiency but requires more execution time due to the complexity of heap operations.

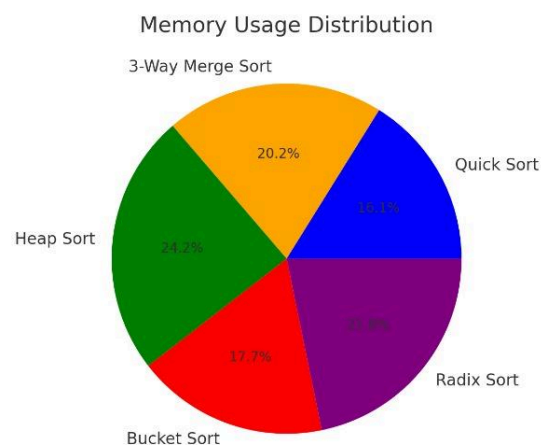
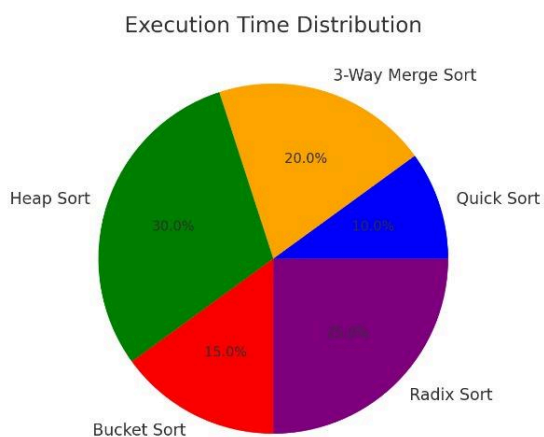
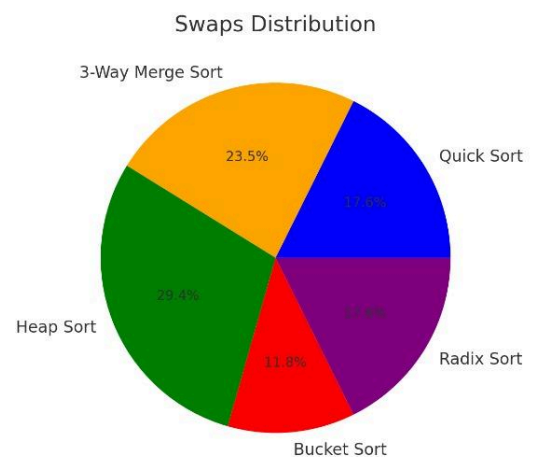
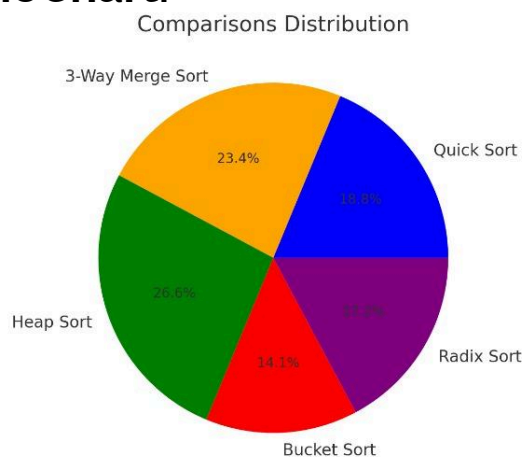
**Bucket Sort** is excellent for certain distributions but less versatile due to its dependency on data distribution characteristics.

# VISUALISATIONS:

## Line Graph:



## PieChart:



## Question 2:

Optimizations and strategic data structure applications can significantly refine each algorithm's performance. For guidance on implementation, consider consulting "Algorithm Design" by Goodrich et al., along with other pertinent scholarly texts viz., CLRS book. Detail the refinements applied to the algorithms and expound on how these modifications have elevated their efficiency. This should be substantiated both in theory and through empirical comparison, employing extensive and varied input test cases, against the algorithms' initial versions.

### Problem Statement:

- Given two strings `text1` and `text2`, return the length of their longest common subsequence. If there is no common subsequence, return 0.
- A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters. For example, "ace" is a subsequence of "abcde".
- A common subsequence of two strings is a subsequence that is common to both strings.

**Brute Force:** Enumerate all subsequences of `X` and take the largest one that is also a subsequence of `Y`. Since each character of `X` is either in or not in a subsequence, there are potentially  $2^n$  different subsequences of `X`, each of which requires  $O(m)$  time to determine whether it is a subsequence of `Y`. Thus, the brute-force approach yields an exponential algorithm that runs in  $O(2^n * 2^m)$  time.

## Theoretical Comparison Between Brute Force and Memoized Code:

### Refinements and Efficiency Improvements:

#### Memoization:

- **What It Does:** Memoization involves **storing the results of subproblems in a table** (DP table) to **avoid redundant computations**. Instead of recomputing results for the same subproblem multiple times, the algorithm retrieves the result from the table.
- **Efficiency Improvement:** By avoiding repeated calculations, memoization transforms the time complexity from exponential  $O(2^n \times 2^m)$  (in the brute-force approach) to polynomial  $O(m \times n)$ , making it feasible for longer strings.



## Recursive with Memoization:

- **What It Does:** The recursive function **explores all possible subsequences** while leveraging memoization to store intermediate results. **Each cell in the DP table represents the length of the LCS for substrings  $\text{text1}[0..i]$  and  $\text{text2}[0..j]$ .**
- **Efficiency Improvement:** The recursive approach with memoization ensures that each **subproblem is computed only once** and reused, which drastically reduces the number of computations compared to the brute-force approach.

## Comparison:

**Brute Force:** Can become impractical for larger strings due to its exponential time complexity. For instance, generating all subsequences involves a combinatorial explosion as the length of the strings increases.

**Dynamic Programming with Memoization:** More efficient for larger strings because it systematically builds up solutions to subproblems and reuses them. This results in a significant reduction in computation time and space compared to the brute-force method.

### Summary:

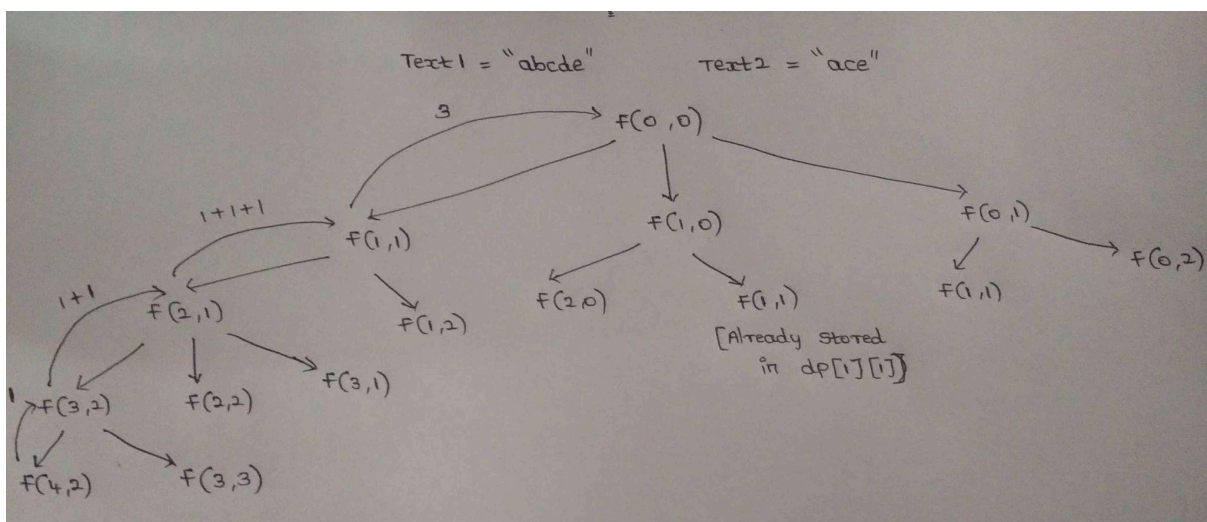
The dynamic programming approach with memoization refines the LCS problem by systematically breaking it down into smaller overlapping subproblems and storing intermediate results. This refinement leads to a time complexity  $O(m \times n)$  and makes it practical for longer strings, unlike the brute-force method which has exponential time complexity.

## Dry Run of Test Case:

**text1:** "abcde"

**Output:** 3

**text2:** "ace"





# Empirical Comparison:

## 1] Brute Force:

Theoretical Time Complexity:  $O(2^n * 2^m)$

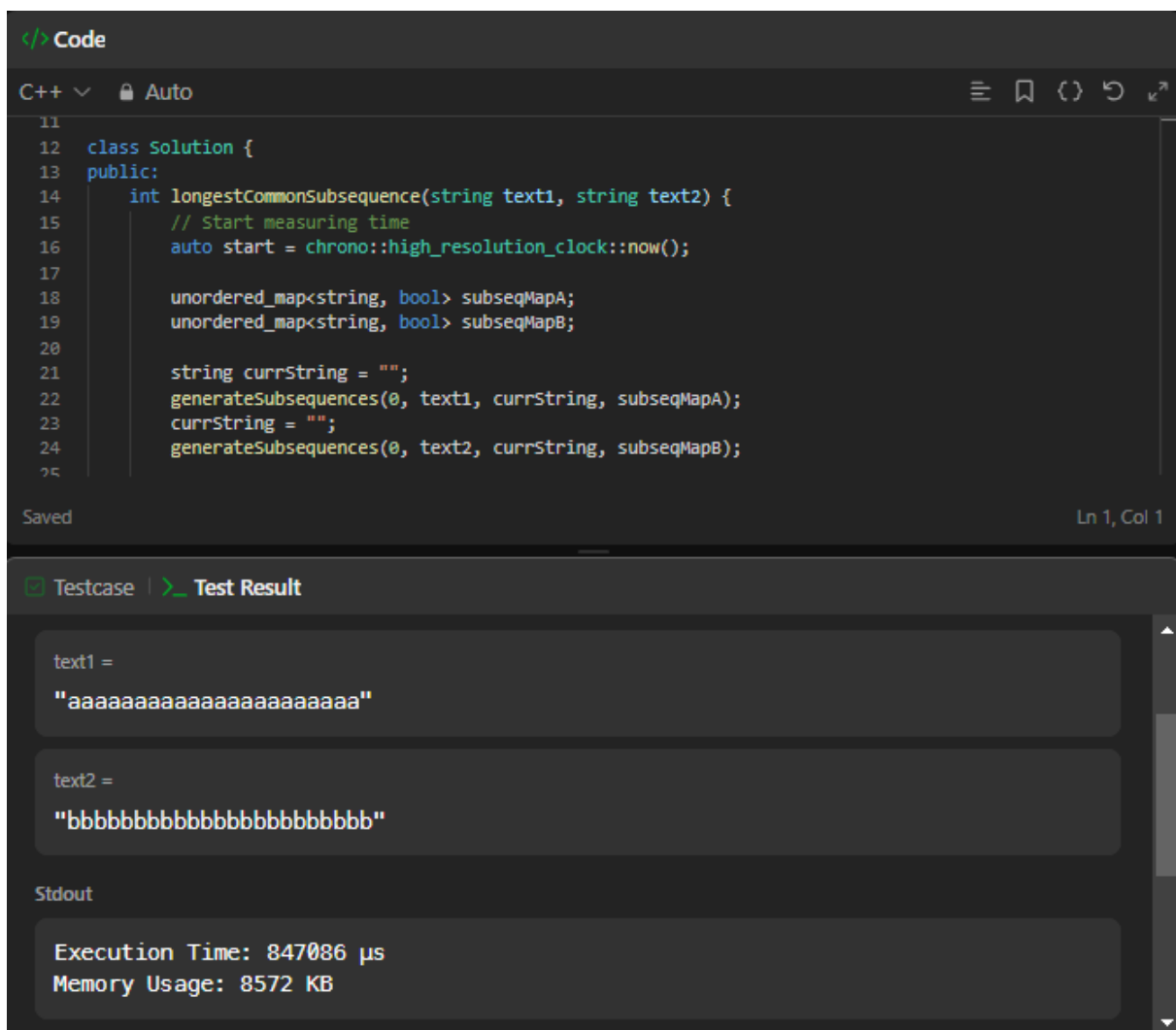
Theoretical Space Complexity:  $O(2^n * 2^m)$

Text1: "aaaaaaaaaaaaaaaaaaaaa"

Text2: "bbbbbbbbbbbbbbbbbbbbbb"

Execution Time: 847086 ms

Memory Usage: 8572 KB



The screenshot displays a C++ IDE with a code editor and a test result panel. The code defines a `Solution` class with a `longestCommonSubsequence` method. This method uses `unordered_map` to store subsequence information for two input strings, `text1` and `text2`. It initializes a `currString` and calls `generateSubsequences` for both strings. The test result panel shows the input strings `text1 = "aaaaaaaaaaaaaaaaaaaaa"` and `text2 = "bbbbbbbbbbbbbbbbbbbbbb"`. The execution time is 847086  $\mu$ s and the memory usage is 8572 KB.

```
11
12 class Solution {
13 public:
14     int longestCommonSubsequence(string text1, string text2) {
15         // Start measuring time
16         auto start = chrono::high_resolution_clock::now();
17
18         unordered_map<string, bool> subseqMapA;
19         unordered_map<string, bool> subseqMapB;
20
21         string currString = "";
22         generateSubsequences(0, text1, currString, subseqMapA);
23         currString = "";
24         generateSubsequences(0, text2, currString, subseqMapB);
25     }
26 }
```

Testcase | Test Result

text1 =  
"aaaaaaaaaaaaaaaaaaaaa"

text2 =  
"bbbbbbbbbbbbbbbbbbbbbb"

Stdout

Execution Time: 847086  $\mu$ s  
Memory Usage: 8572 KB

## 2] Memoized Approach

Theoretical Time Complexity:  $O(m * n)$

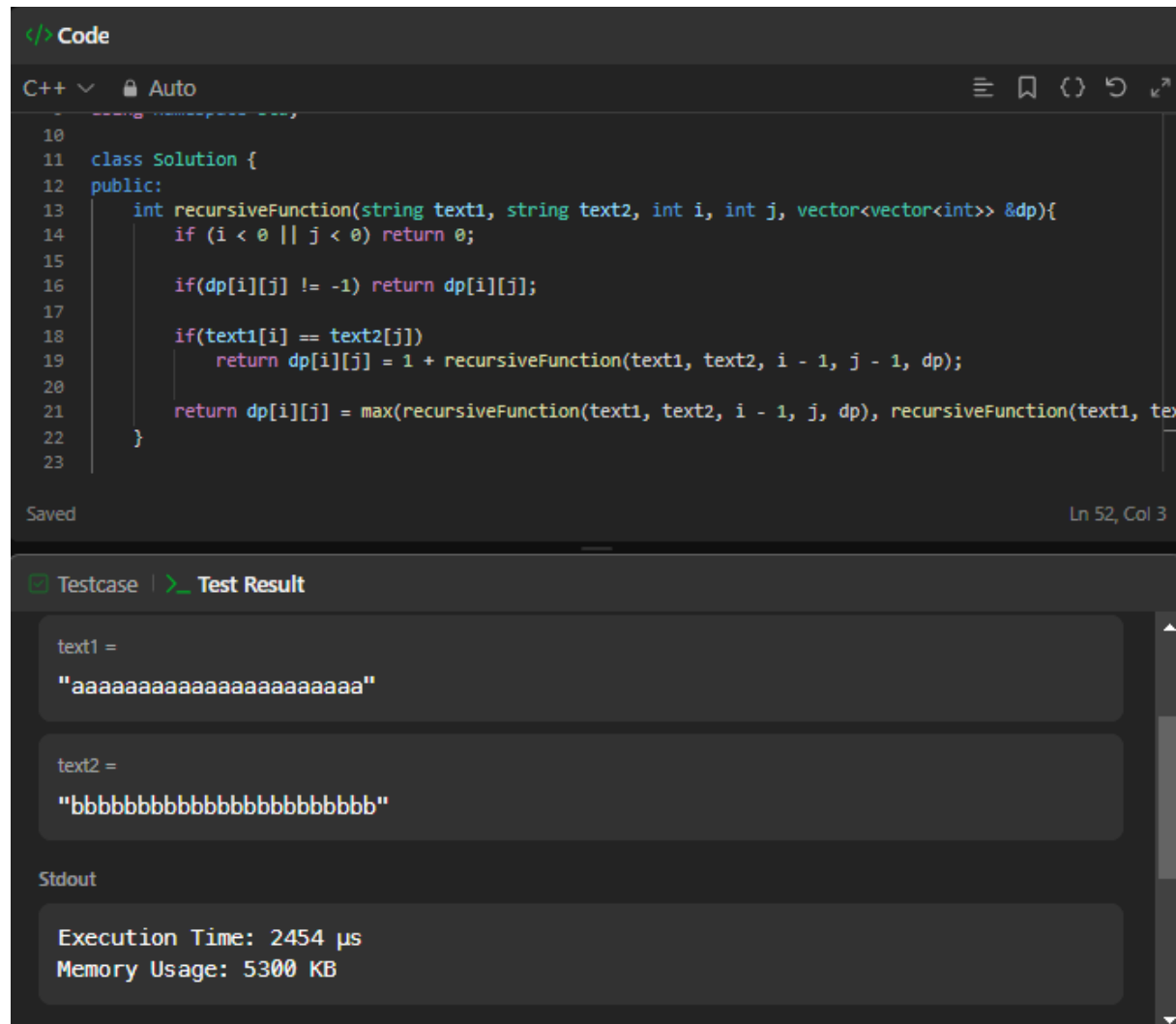
Theoretical Space Complexity:  $O(m * n) + O(\max(m, n))$

Text1: "aaaaaaaaaaaaaaaaaaaaa"

Text2: "bbbbbbbbbbbbbbbbbbbbbb"

Execution Time: 2454 ms

Space Complexity: 5300 KB



```
</> Code
C++ v Auto
10
11 class Solution {
12 public:
13     int recursiveFunction(string text1, string text2, int i, int j, vector<vector<int>> &dp){
14         if (i < 0 || j < 0) return 0;
15
16         if(dp[i][j] != -1) return dp[i][j];
17
18         if(text1[i] == text2[j])
19             return dp[i][j] = 1 + recursiveFunction(text1, text2, i - 1, j - 1, dp);
20
21         return dp[i][j] = max(recursiveFunction(text1, text2, i - 1, j, dp), recursiveFunction(text1, text2, i, j - 1, dp));
22     }
23 }
```

Saved Ln 52, Col 3

Testcase | Test Result

text1 =  
"aaaaaaaaaaaaaaaaaaaaa"

text2 =  
"bbbbbbbbbbbbbbbbbbbbbb"

Stdout

Execution Time: 2454 μs  
Memory Usage: 5300 KB

As we can see the huge difference between the runtime and memory usage of those different algorithms for the same test case. Thus we can infer that runtime of an algorithm can be significantly reduced by using additional space which is foundational logic of dynamic programming where the result of each state is computed only once and stored in memory and utilised again.

## Question 3:

### Strongly Connected Components [Kosaraju Algorithm]:

#### Primitive Analysis:

- Step 1: Perform a Depth-First Search (DFS) to compute the finish order of vertices. This requires visiting each vertex and edge exactly once, resulting in  $O(V + E)$  primitive operations (where  $V$  is the number of vertices, and  $E$  is the number of edges).
- Step 2: Reverse the graph, which involves reversing the direction of all edges. This also requires  $O(V + E)$  primitive operations.
- Step 3: Perform another DFS on the reversed graph based on the finishing order obtained in Step 1. This again involves  $O(V + E)$  primitive operations.
- Total Count: The total number of primitive operations for Kosaraju's algorithm is  $O(V + E)$ .

#### Execution Duration:

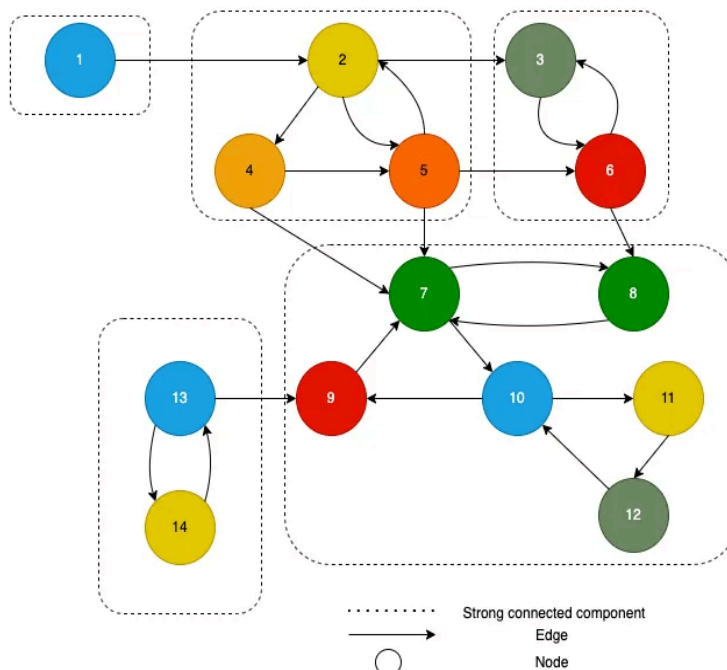
**Theoretical Time Complexity:**  $O(V + E)$

**Theoretical Space Complexity:**  $O(V + E)$

**Runtime:** 8802 ms

**Memory Utilised:** 41 KB

**Test Case:**



**Expected Output:** 5

```
1 package SCC;
2
3 import java.util.*;
4
5 class SCC {
6
7     private void getSortedOrder(List<List<Integer>> adj, int vertex, boolean[] visited, Stack<Integer> sortedOrder) {
8         visited[vertex] = true;
9         for (int node : adj.get(vertex)) {
10             if (!visited[node]) {
11                 getSortedOrder(adj, node, visited, sortedOrder);
12             }
13         }
14         sortedOrder.push(vertex);
15     }
16
17     private void DFS(List<List<Integer>> adjList, int vertex, boolean[] visited) {
18         visited[vertex] = true;
19         for (int node : adjList.get(vertex)) {
20             if (!visited[node]) {
21                 DFS(adjList, node, visited);
22             }
23         }
24     }
25
26     public int kosaraju(int V, List<List<Integer>> adj) {
27         Stack<Integer> sortedOrder = new Stack<>();
28         boolean[] visited = new boolean[V];
29     }
30 }
```

Problems Javadoc Declaration Coverage

<terminated> SCC [Java Application] C:\Users\De\l\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_17.0.8.v20230801-1

Number of Strongly Connected Components (SCCs): 10  
Time taken: 8802 ms  
Memory used: 41 KB

## Articulation Point in a Graph:

### Primitive Analysis:

1. **Initialization:**
  - Allocate arrays for visited status, discovery times, lowest reachable vertices, and marking articulation points.
  - **Primitive Operations:**  $O(V)$
2. **DFS Traversal:**
  - Perform DFS from each unvisited vertex, visiting every vertex and edge once.
  - **Primitive Operations:**  $O(V + E)$
3. **Articulation Point Check:**
  - During DFS, compare discovery and lowest reachable times to determine if a vertex is an articulation point.
  - **Primitive Operations:**  $O(V + E)$
4. **Collect Results:**
  - Iterate over the vertices to gather all marked articulation points.
  - **Primitive Operations:**  $O(V)$

**Total Primitive Operations:**  $O(V + E)$

Execution Duration:

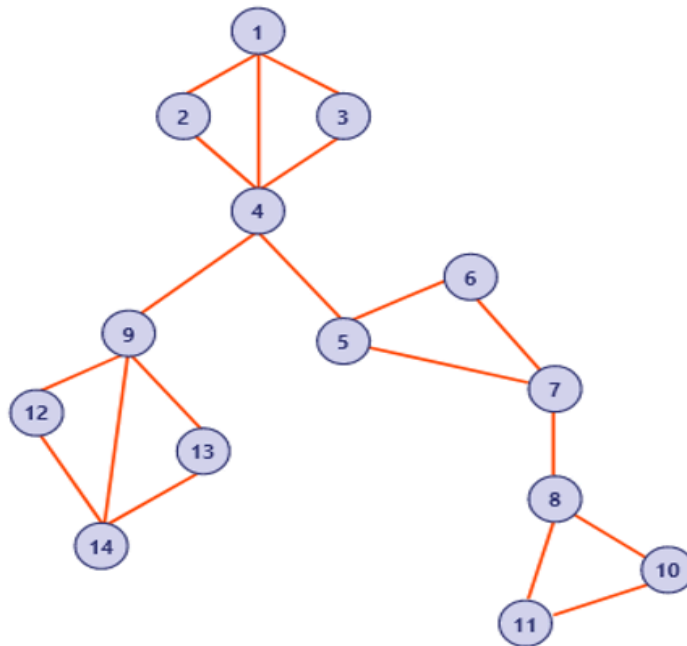
Theoretical Time Complexity:  $O(V + E)$

Theoretical Space Complexity:  $O(V)$

Runtime: 1485 ms

Memory Utilised: 905 KB

Test Case:



Expected Output: [4, 5, 7, 8, 9]

Main.java	Output
<pre>4 3- class ArticulationPoint { 4   private int timer = 1; 5 6-   private void dfs(int node, int parent, int[] vis, int[] tin, 7       int[] low, int[] mark, ArrayList&lt;ArrayList&lt;Integer&gt;&gt; adj) { 7       vis[node] = 1; 8       tin[node] = low[node] = timer; 9       timer++; 10      int child = 0; 11 12-     for (Integer it : adj.get(node)) { 13         if (it == parent) continue; 14 15-         if (vis[it] == 0) { 16             dfs(it, node, vis, tin, low, mark, adj); 17             low[node] = Math.min(low[node], low[it]); 18 19             // Check if node is an articulation point 20             if (low[it] &gt;= tin[node] &amp;&amp; parent != -1) { 21                 mark[node] = 1; 22             } 23             child++; 24         } 25     } 26 }</pre>	<pre>java -cp /tmp/7PpL6W700h/Main Articulation points in the graph are: [4, 5, 7, 8, 9] Time taken : 1485 ms Maximum memory used: 905 KB === Code Execution Successful ===</pre>

## Bridges in a Graph [Tarjan's Algorithm]:

### Primitive Analysis:

1. **Initialization:** Allocate arrays for visited status, discovery times, lowest reachable vertices, and articulation points.
  - o **Operations:**  $O(V)$

2. **DFS Traversal:** Perform DFS from each unvisited vertex, visiting each vertex and edge once.
  - **Operations:**  $O(V + E)$
3. **Articulation Point Check:** During DFS, check if a vertex is an articulation point based on discovery and lowest reachable times.
  - **Operations:**  $O(V + E)$
4. **Collect Results:** Iterate over vertices to gather all articulation points.
  - **Operations:**  $O(V)$

**Total Primitive Operations:**  $O(V + E)$

Execution Duration:

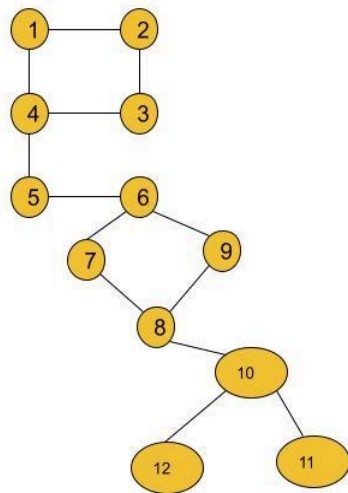
**Theoretical Time Complexity:**  $O(V + E)$

**Theoretical Space Complexity:**  $O(V)$

**Runtime:** 3430 ms

**Memory Utilised:** 91 KB

**Test Case:**



**Expected Output:**  $[[10, 11], [10, 12], [8, 10], [5, 6], [4, 5]]$



Main.java	Output
<pre> 1- import java.util.*; 2 3- class Solution { 4     int timer = 1; 5 6-     public void DFS(List&lt;List&lt;Integer&gt;&gt; adjList, int server, int        parent, int[] tin, int[] low, boolean[] visited, List&lt;List        &lt;Integer&gt;&gt; bridges) { 7         visited[server] = true; 8         low[server] = tin[server] = timer; 9         timer += 1; 10 11        for (int it : adjList.get(server)) { 12            if (it == parent) continue; 13            if (!visited[it]) { 14                DFS(adjList, it, server, tin, low, visited, bridges                 ); 15                low[server] = Math.min(low[server], low[it]); 16                if (low[it] &gt; tin[server]) { 17                    bridges.add(Arrays.asList(server, it)); 18                } 19            } else { </pre>	<pre> java -cp /tmp/pjVJMSW7ud/Main Bridges (Critical Connections) in the graph are: [[10, 11], [10, 12], [8, 10], [5, 6], [4, 5]] Time taken : 3430 microseconds Memory used: 91 KB  === Code Execution Successful === </pre>

## Shortest Path in a Graph [Dijkstra's Algorithm]:

### Primitive Analysis:

- Initialization:** Set up the distance array, priority queue, and visited status for all vertices.
  - Operations:  $O(V)$
- Priority Queue Operations:** Extract the minimum distance vertex and update distances for its adjacent vertices.
  - Operations:  $O((V + E) * \log V)$
- Distance Updates:** For each edge, relax (update) the distance to the adjacent vertex if a shorter path is found.
  - Operations:  $O(E * \log V)$
- Collect Results:** Gather the final shortest distances to all vertices.
  - Operations:  $O(V)$

**Total Primitive Operations:**  $O((V + E) * \log V)$

Execution Duration:

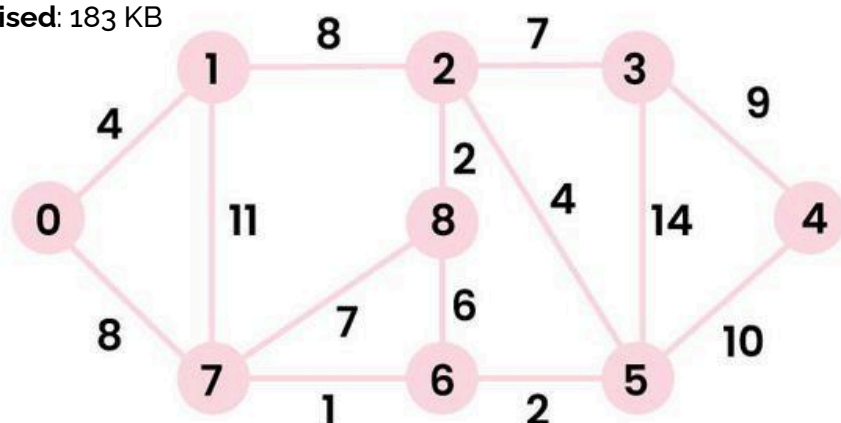
**Theoretical Time Complexity:**  $O((V + E) * \log V)$

**Theoretical Space Complexity:**  $O(V + E)$

**Runtime:** 23470 ms

**Memory Utilised:** 183 KB

**Test Case:**



**Note:** Shortest Distance from 0th vertex to every other vertex is calculated.

**Expected Output:** [0, 4, 12, 19, 21, 11, 9, 8, 14]

```
Main.java
14  int[] node = pq.poll();
15  int u = node[0];
16  int d = node[1];
17
18  if (d > dist[u]) continue;
19
20  for (int[] neighbor : adjList.get(u)) {
21      int v = neighbor[0];
22      int weight = neighbor[1];
23
24      if (dist[u] + weight < dist[v]) {
25          dist[v] = dist[u] + weight;
26          pq.offer(new int[]{v, dist[v]});
27      }
28  }
29  }
30
31  return dist;
32  }
33 }
```

```
Output
java -cp /tmp/wRXDmIIYFe/Main
Shortest distances from vertex 0 are:
0 4 12 19 21 11 9 8 14
Time taken: 23470 microseconds
Memory used (in bytes): 183536

=== Code Execution Successful ===
```

## Shortest Path in a Graph [Bellman Ford Algorithm]:

### Primitive Analysis:

1. **Initialization:** Set up the distance array and initialise all distances to infinity except the source vertex.
  - o **Operations:**  $O(V)$
2. **Relaxation of Edges:** Iterate over all edges, updating the distance to adjacent vertices if a shorter path is found. This is done  $V-1$  times.
  - o **Operations:**  $O(V * E)$
3. **Negative-Weight Cycle Check:** Iterate over all edges one more time to check for negative-weight cycles.
  - o **Operations:**  $O(E)$
4. **Collect Results:** Gather the final shortest distances to all vertices.
  - o **Operations:**  $O(V)$

**Total Primitive Operations:**  $O(V * E)$

Execution Duration:

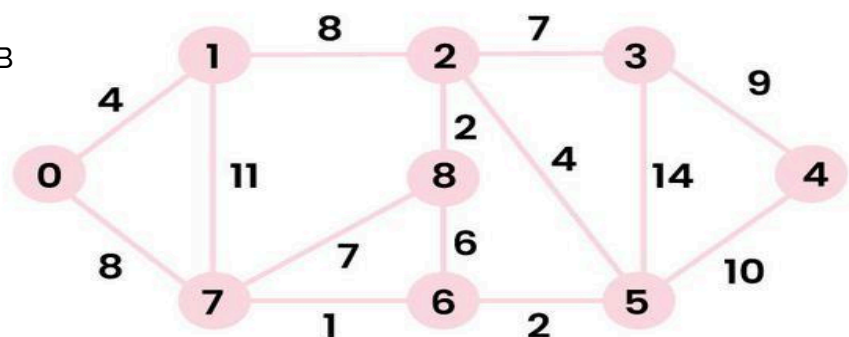
**Theoretical Time Complexity:**  $O(V * E)$

**Theoretical Space Complexity:**  $O(V + E)$

**Runtime:** 3034 ms

**Memory Utilised:** 91 KB

**Test Case:**



**Note:** Shortest Distance from 0th vertex to every other vertex is calculated.

**Expected Output:** [0, 4, 12, 19, 21, 11, 9, 8, 14]

```
Main.java
1- import java.util.*;
2
3- class Solution {
4-     // Bellman-Ford algorithm
5-     public int[] bellmanFord(int n, List<int[]> edges, int src) {
6-         int[] dist = new int[n];
7-         Arrays.fill(dist, Integer.MAX_VALUE);
8-         dist[src] = 0;
9
10-        // Relax all edges n-1 times
11-        for (int i = 1; i < n; i++) {
12-            for (int[] edge : edges) {
13-                int u = edge[0];
14-                int v = edge[1];
15-                int weight = edge[2];
16
17-                if (dist[u] != Integer.MAX_VALUE && dist[u] + weight
18-                    < dist[v]) {
19-                    dist[v] = dist[u] + weight;
20-                }
21-            }
22-        }
23-        return dist;
24-    }
25-}
```

```
Output
java -cp /tmp/4ooxhMlpe/Main
Shortest distances from vertex 0 are:
0 4 12 19 28 16 18 8 14
Time taken: 3034 ms
Memory used: 91 KB

=== Code Execution Successful ===
```

## Minimum Spanning Tree [Boruvka's Algorithm]:

### Primitive Analysis:

1. **Initialization:** Set up parent and rank arrays for union-find operations. Initialize each vertex as its own component.
  - a. **Operations:**  $O(V)$
2. **Finding Cheapest Edges:** For each component, find the cheapest edge connecting it to another component. This involves iterating over all edges.
  - a. **Operations:**  $O(E * \log V)$
3. **Union of Components:** Use union-find with path compression and union by rank to merge components.
  - a. **Operations:**  $O(\log V)$  per union operation
4. **Iterative Merging:** Repeat the process until all components are merged into one. In each iteration, the number of components is halved.
  - a. **Operations:**  $O(V)$

**Total Primitive Operations:**  $O(V * E)$

Execution Duration:

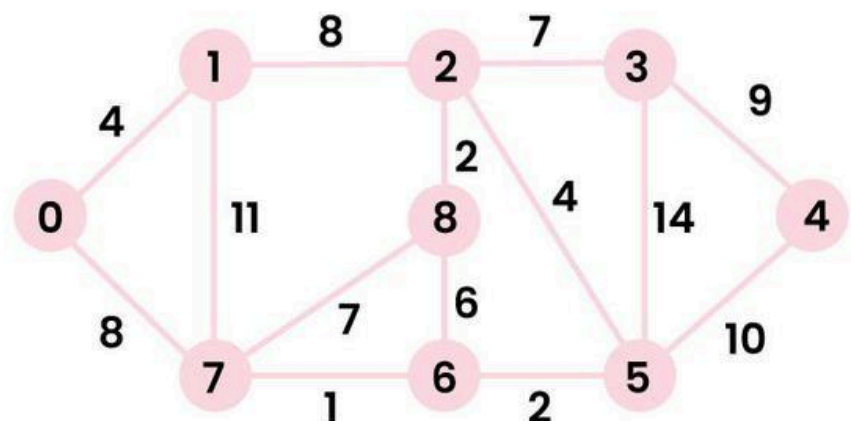
**Theoretical Time Complexity:**  $O(V * E)$

**Theoretical Space Complexity:**  $O(V + E)$

**Runtime:** 3034 ms

**Memory Utilised:** 91 KB

**Test Case:**



**Note:** Shortest Distance from 0th vertex to every other vertex is calculated.

**Expected Output:** [0, 4, 12, 19, 21, 11, 9, 8, 14]

Main.java

Share

Run

```
1- import java.util.*;
2
3- class Solution {
4     // Boruvka's algorithm to find MST
5     public int boruvkaMST(int n, List<int[]> edges) {
6         // Initialize parent and rank arrays
7         int[] parent = new int[n];
8         int[] rank = new int[n];
9         for (int i = 0; i < n; i++) {
10             parent[i] = i;
11             rank[i] = 0;
12         }
13
14         int mstWeight = 0;
15         int components = n;
```

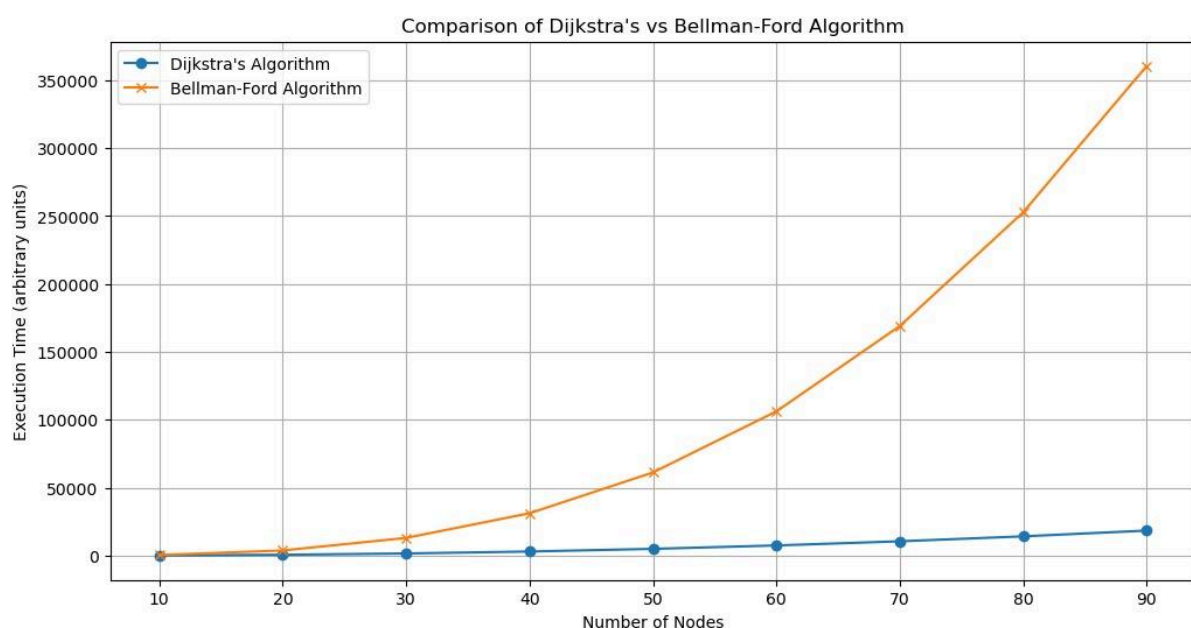
Output

```
java -cp /tmp/ZKP1CZUMtz/Main
Weight of MST is: 37
Time taken: 2950 ms
Memory utilised: 91 KB

=== Code Execution Successful ===
```

## Conclusion:

The comparison between Dijkstra's Algorithm and the Bellman-Ford Algorithm reveals a significant difference in execution time as the number of nodes increases. Dijkstra's Algorithm exhibits a much lower execution time, maintaining efficiency even as the number of nodes grows, while the Bellman-Ford Algorithm's execution time increases exponentially. This indicates that Dijkstra's Algorithm is more scalable and better suited for larger graphs, whereas Bellman-Ford may become impractical for large-scale problems due to its higher computational cost.



## Question 4:

"Snakes and Ladders," a venerable board game with origins tracing back to no later than the 16th century in India, is played on a board that forms an  $n \times n$  grid. This grid is sequentially numbered from 1 to  $n^2$ , commencing at the lower left corner and advancing row by row from the bottom to the top, with each row alternating direction. Within this grid, specific square pairs, always situated in distinct rows, are interconnected by "snakes" (descending) or "ladders" (ascending). No square may serve as the terminal point for more than one snake or ladder. The objective is to be the swiftest to arrive at the final square,  $n \times n$ , also known as the paramapatam.

Play begins with a token placed on square 1, located at the bottom left. Each turn allows the player to move their token forward by up to  $k$  spaces ( $k \leq 6$ ), where  $k$  is a predetermined constant. Should the token conclude its move on a snake's head, it must descend to the snake's tail. Conversely, landing on the base of a ladder means the token ascends to the ladder's top.

The tasks at hand are as follows:

You are presented with a board characterized by a) its size  $n \times n$  (with  $n$  being no less than 8), b) the count of snakes and ladders, and c) the starting and ending grid positions of each snake and ladder. Your task is to confirm that the board adheres to these stipulations:

- There exists at least one viable path to the goal.
- No pair of snakes or ladders share the same starting or ending grid position.
- The arrangement of snakes and ladders does not create any loops.

Directly from the starting position, there is no ladder that leads straight to the destination.

### Approach 1 [Memoized DFS]:

- The DFS approach combined with DP (memoization) is used to explore all possible paths from the start to the end of the board. It checks each possible dice roll, moving the piece accordingly, and handles transitions through snakes and ladders.
- The DP table (`dp`) stores the result of subproblems to avoid redundant calculations, while the `visited` array prevents revisiting the same cell within a recursion path. This ensures that all possible paths are explored efficiently.

### Approach 2 [BFS]:

- The BFS approach is employed to find the shortest path from the start to the end of the board. It systematically explores all possible moves layer by layer, using a queue to track the current position and number of moves.
- This method ensures that the first time it reaches the end, it does so with the minimum number of moves, making it optimal for shortest path problems.

## Checking for Duplicates and Cycles:

- **checkDuplicates:** This method uses a set to detect if there are any overlapping positions in snakes or ladders. If a start or end position is duplicated, it returns **false**.
- **checkCycle:** The DFS-based method detects cycles by tracking the recursion stack (**recStack**). If a node is revisited while it's still in the stack, a cycle is present.

## Analysis:

### 1. Time Complexity:

- The DFS approach has a time complexity of  $O(\text{dimension}^2)$  in the worst case, where every cell is visited. However, due to DP, repeated calculations are avoided.
- The BFS approach also runs in  $O(\text{dimension}^2)$  due to the need to explore every possible cell in the worst case.

### 2. Space Complexity:

- The space complexity is  $O(\text{dimension}^2)$  for both methods, considering the **visited**, **dp**, and **recStack** arrays.

### 3. Efficiency:

- The use of unordered maps and sets ensures that the operations related to snakes and ladders (insertion, lookup) are efficient, i.e., average  $O(1)$ .

## Execution Duration:

**Theoretical Time Complexity:**  $O(\text{dimension} * \text{dimension})$

**Theoretical Space Complexity:**  $O(\text{dimension} * \text{dimension})$

**Test Case:**

