VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

ANALYSIS AND DESIGN OF ALGORITHMS

Submitted by

GOPIKA PUSHPARAJAN (1BM23CS101)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Feb 2025 to Jun 2025

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "ANALYSIS AND DESIGN OF ALGORITHMS – 23CS4PCADA" carried out by **GOPIKA PUSHPARAJAN (1BM23CS101),** who is a bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2025. The Lab report has been approved as it satisfies the academic requirements in respect of a **ANALYSIS AND DESIGN OF ALGORITHMS - (23CS4PCADA)** work prescribed for the said degree.

Srushti C S
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

INDEX

Name GOPIKA PUSHPARAJAN Sub. ADA

Std.: 4B Div. Roll No.

Telephone No. E-mail ID.

Blood Group. Birth Day.

Sr.No.	Title	~~Page No.~~ Date	Sign./Remarks
1.	Stacks - using Arrays & Linked Lists	28/2/25	
2	Queues - using Arrays & Linked Lists	7/3/25	
3	Mergesort	21/3/25	10
4.	Quicksort	4/4/25	10
5	Prims algorithm	4/4/25	6
6.	Topological sort	16/5/25	6
7	0/1 Knapsack	16/5/25	6
8	Floyd's Algorithm	16/5/25	10
9.	Dijkstra's Algorithm	16/5/25	10
10.	N Queen's, using Backtrack	16/5/25	10
11.	Johnson Trotter's Algorithm	23/5/25	10
12.	Heap sort	23/5/25	10

Program -1

Question: Topological Sorting

Code:

```c
#include <stdio.h>
#include <stdlib.h>

// Maximum number of vertices in the graph
#define MAX_VERTICES 100

// Structure for adjacency list
struct Graph {
    int V;            // Number of vertices
    int **adj;        // Adjacency matrix
    int *inDegree;    // Array to store in-degrees of vertices
};

// Function to create a graph
struct Graph* createGraph(int V) {
    struct Graph *graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->adj = (int**)malloc(V * sizeof(int*));
    graph->inDegree = (int*)malloc(V * sizeof(int));

    for (int i = 0; i < V; i++) {
        graph->adj[i] = (int*)malloc(V * sizeof(int));
        graph->inDegree[i] = 0;  // Initialize in-degrees to 0
        for (int j = 0; j < V; j++) {
            graph->adj[i][j] = 0;  // Initialize adjacency matrix to 0
        }
    }

    return graph;
}
```

```c
// Function to add an edge to the graph
void addEdge(struct Graph *graph, int u, int v) {
    graph->adj[u][v] = 1;
    graph->inDegree[v]++;
}

// Function for Topological Sort using Kahn's Algorithm (BFS)
void topologicalSort(struct Graph *graph) {
    int V = graph->V;
    int *queue = (int*)malloc(V * sizeof(int));  // Queue for Kahn's algorithm
    int front = 0, rear = 0;

    // Add all vertices with in-degree 0 to the queue
    for (int i = 0; i < V; i++) {
        if (graph->inDegree[i] == 0) {
            queue[rear++] = i;
        }
    }

    int count = 0;  // Counter for visited vertices

    // Process the graph
    while (front != rear) {
        int u = queue[front++];  // Dequeue a vertex
        printf("%d ", u);         // Print the current vertex
        count++;

        // Reduce the in-degree of neighboring vertices
        for (int v = 0; v < V; v++) {
            if (graph->adj[u][v] == 1) {
                graph->inDegree[v]--;
                if (graph->inDegree[v] == 0) {
                    queue[rear++] = v;  // Enqueue if in-degree becomes 0
                }
            }
        }
    }

    // If the number of visited vertices is less than the number of vertices in
the graph,
    // it indicates a cycle, so topological sorting is not possible.
    if (count != V) {
        printf("\nCycle detected, topological sort not possible.\n");
    }
}

int main() {
    int V, E;
```

```
    // Taking user input for number of vertices and edges
    printf("Enter the number of vertices: ");
    scanf("%d", &V);

    // Create a graph with V vertices
    struct Graph *graph = createGraph(V);

    // Taking user input for number of edges
    printf("Enter the number of edges: ");
    scanf("%d", &E);

    // Taking user input for each edge (u, v)
    for (int i = 0; i < E; i++) {
        int u, v;
        printf("Enter edge (u v): ");
        scanf("%d %d", &u, &v);
        if (u >= V || v >= V || u < 0 || v < 0) {
            printf("Invalid edge. Please enter valid vertices.\n");
            i--;  // Decrement to re-enter the edge
            continue;
        }
        addEdge(graph, u, v);
    }

    // Perform topological sort
    printf("Topological Sort: ");
    topologicalSort(graph);

    // Free memory
    for (int i = 0; i < V; i++) {
        free(graph->adj[i]);
    }
    free(graph->adj);
    free(graph->inDegree);
    free(graph);

    return 0;
}
```

Result:

```
PS C:\Users\student\Desktop\os_207> cd "c:\Users\student\Desktop\os_207\" ; if ($?) { gcc ada.c -o ada } ; if ($?) { .\ada }
Enter the number of vertices: 6
Enter the number of edges: 6
Enter edge (u v): 5 2
Enter edge (u v): 5 0
Enter edge (u v): 4 0
Enter edge (u v): 4 1
Enter edge (u v): 2 3
Enter edge (u v): 3 1
Topological Sort: 4 5 0 2 3 1
PS C:\Users\student\Desktop\os_207>
```

Program -2

Question: Merge Sort

Code:

```c
#include <stdio.h>

// Function to merge two halves
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    // Create temp arrays
    int L[n1], R[n2];

    // Copy data
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge temp arrays back into arr[l..r]
    i = 0; j = 0; k = l;

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

```c
    // Copy remaining elements
    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
}

// Merge sort function
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

// Main function
int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array (Merge Sort): ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

Result:

```
PS C:\Users\student\Desktop\os_207> cd "c:\Users\student\Desktop\os_207\" ; if ($?) { gcc ada.c -o ada } ; if ($?) { .\ada }
Enter number of elements: 5
Enter 5 elements:
2 3 1 5 4
Sorted array (Merge Sort): 1 2 3 4 5
PS C:\Users\student\Desktop\os_207>
```

Program -3

Question: Quick Sort

Code:

```c
#include <stdio.h>

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Last element as pivot
    int i = low - 1;        // Index of smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// Quick sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // Partition index

        // Recursively sort subarrays
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```c
// Main function
int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    quickSort(arr, 0, n - 1);

    printf("Sorted array (Quick Sort): ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

Result:

Program -4

Question: Prims Algorithm

Code:

```c
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

// Function to find the vertex with the minimum key value
```

```
int minKey(int key[], bool mstSet[], int V) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }

    return min_index;
}

// Function to implement Prim's algorithm
void primMST(int graph[][5], int V) {
    int parent[V];  // Array to store the constructed MST
    int key[V];     // Key values used to pick the minimum weight edge
    bool mstSet[V]; // To represent the set of vertices included in MST

    // Initialize all keys to infinity, mstSet to false
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }

    // Always include the first vertex in MST
    key[0] = 0;  // Make key[0] 0 so that it is picked first
    parent[0] = -1; // First node has no parent

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices
        // not yet included in MST
        int u = minKey(key, mstSet, V);

        // Add the picked vertex to the MST set
        mstSet[u] = true;

        // Update key and parent values of the adjacent vertices of the picked
vertex
        for (int v = 0; v < V; v++) {
            // graph[u][v] is non-zero only for adjacent vertices of u
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than the current
key[v]
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                key[v] = graph[u][v];
                parent[v] = u;
            }
```

```
        }
    }

    // Print the constructed MST
    printf("Edge   Weight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d   %d\n", parent[i], i, graph[i][parent[i]]);
    }
}

// Main function
int main() {
    int V, E;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);

    int graph[V][V];
    // Initialize the graph to 0
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            graph[i][j] = 0;
        }
    }

    // Take input for the edges
    printf("Enter the edges (u, v, weight):\n");
    for (int i = 0; i < E; i++) {
        int u, v, weight;
        printf("Enter edge %d (u v weight): ", i + 1);
        scanf("%d %d %d", &u, &v, &weight);
        graph[u][v] = weight;
        graph[v][u] = weight;  // Undirected graph
    }

    primMST(graph, V);

    return 0;
}
```

Result:

```
Enter the number of vertices: 5
Enter the number of edges: 7
Enter the edges (u, v, weight):
Enter edge 1 (u v weight): 0 1 2
Enter edge 2 (u v weight): 0 3 6
Enter edge 3 (u v weight): 1 2 3
Enter edge 4 (u v weight): 1 3 8
Enter edge 5 (u v weight): 1 4 5
Enter edge 6 (u v weight): 2 4 7
Enter edge 7 (u v weight): 3 4 9
Edge   Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
PS C:\Users\student\Desktop\os_207>
```

Program -5

Question: Krushkals Algorithm

Code:

```c
#include <stdio.h>
#include <stdlib.h>

// Structure to represent a weighted edge
typedef struct {
    int u, v, weight;
} Edge;

// Structure to represent a subset for union-find
typedef struct {
    int parent, rank;
} Subset;

// Function to compare two edges based on their weights
int compareEdges(const void *a, const void *b) {
    return ((Edge *)a)->weight - ((Edge *)b)->weight;
}

// Find function for Union-Find with path compression
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent); // Path compression
    }
    return subsets[i].parent;
}
```

```c
// Union function for Union-Find with union by rank
void unionSets(Subset subsets[], int x, int y) {
    int rootX = find(subsets, x);
    int rootY = find(subsets, y);

    if (rootX != rootY) {
        if (subsets[rootX].rank < subsets[rootY].rank) {
            subsets[rootX].parent = rootY;
        } else if (subsets[rootX].rank > subsets[rootY].rank) {
            subsets[rootY].parent = rootX;
        } else {
            subsets[rootY].parent = rootX;
            subsets[rootX].rank++;
        }
    }
}

// Kruskal's algorithm to find the MST
void kruskalMST(Edge edges[], int V, int E) {
    Edge result[V];  // Array to store the final MST
    int e = 0;       // Count of edges in MST
    int i = 0;       // Index for sorted edges
    Subset *subsets = (Subset *)malloc(V * sizeof(Subset));

    // Initialize subsets: each vertex is its own parent and rank is 0
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Sort edges in non-decreasing order of weight
    qsort(edges, E, sizeof(edges[0]), compareEdges);

    // Process each edge
    while (e < V - 1 && i < E) {
        // Get the next edge
        Edge nextEdge = edges[i++];

        // Find roots of the vertices u and v
        int x = find(subsets, nextEdge.u);
        int y = find(subsets, nextEdge.v);

        // If including this edge does not cause a cycle, include it in the
result
        if (x != y) {
            result[e++] = nextEdge;
            unionSets(subsets, x, y);
        }
```

```
    }

    // Print the resulting MST
    printf("Edges in the MST:\n");
    for (int i = 0; i < e; i++) {
        printf("%d -- %d == %d\n", result[i].u, result[i].v, result[i].weight);
    }

    free(subsets);
}

// Driver program to test the above functions
int main() {
    int V, E;

    // Take user input for number of vertices and edges
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);

    // Dynamically allocate space for edges
    Edge *edges = (Edge *)malloc(E * sizeof(Edge));

    // Take user input for each edge (u, v, weight)
    printf("Enter the edges in the format (u v weight):\n");
    for (int i = 0; i < E; i++) {
        printf("Edge %d: ", i + 1);
        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].weight);
    }

    // Call Kruskal's algorithm
    kruskalMST(edges, V, E);

    // Free allocated memory
    free(edges);

    return 0;
}
```

Result:

```
Enter the number of vertices: 4
Enter the number of edges: 5
Enter the edges in the format (u v weight):
Edge 1: 0 1 10
Edge 2: 0 2 6
Edge 3: 0 3 5
Edge 4: 1 3 15
Edge 5: 2 3 4
Edges in the MST:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
PS C:\Users\student\Desktop\os_207>
```

Program -6

Question: Dijkstras Algorithm

Code:

```c
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define MAX 100  // Maximum number of vertices

// Function to find the vertex with the minimum distance
int minDistance(int dist[], bool visited[], int V) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++) {
        if (!visited[v] && dist[v] < min) {
            min = dist[v];
            min_index = v;
        }
    }
```

```
    return min_index;
}

// Function to print the shortest distances
void printSolution(int dist[], int V, int src) {
    printf("Vertex\tDistance from Source %d\n", src);
    for (int i = 0; i < V; i++) {
        printf("%d\t%d\n", i, dist[i]);
    }
}

// Dijkstra's algorithm function
void dijkstra(int graph[MAX][MAX], int V, int src) {
    int dist[V];         // Output array. dist[i] will hold the shortest distance
from src to i
    bool visited[V];     // visited[i] will be true if vertex i is processed

    // Initialize all distances as infinite and visited[] as false
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        visited[i] = false;
    }

    dist[src] = 0;  // Distance to self is always 0

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, visited, V);
        visited[u] = true;

        for (int v = 0; v < V; v++) {
            if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
                dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    printSolution(dist, V, src);
}

// Main function
int main() {
    int V, E;
    int graph[MAX][MAX] = {0};

    printf("Enter the number of vertices: ");
    scanf("%d", &V);
```

```
    printf("Enter the number of edges: ");
    scanf("%d", &E);

    printf("Enter the edges (u v weight):\n");
    for (int i = 0; i < E; i++) {
        int u, v, w;
        printf("Edge %d: ", i + 1);
        scanf("%d %d %d", &u, &v, &w);
        graph[u][v] = w;
        // If the graph is undirected, uncomment the next line
        // graph[v][u] = w;
    }

    int src;
    printf("Enter the source vertex: ");
    scanf("%d", &src);

    dijkstra(graph, V, src);

    return 0;
}
```

Result:

```
Enter the number of vertices: 5
Enter the number of edges: 7
Enter the edges (u v weight):
Edge 1: 0 1 10
Edge 2: 0 4 5
Edge 3: 1 2 1
Edge 4: 2 3 4
Edge 5: 4 1 3
Edge 6: 4 2 9
Edge 7: 4 3 2
Enter the source vertex: 0
Vertex  Distance from Source 0
0       0
1       8
2       9
3       7
4       5
PS C:\Users\student\Desktop\os_207>
```

Program -7

Question: Johnson Trotter Algorithm

Code:

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

#define LEFT -1
#define RIGHT 1

// Function to print a permutation
void printPermutation(const vector<int>& perm) {
    for (int num : perm)
        cout << num << " ";
    cout << endl;
}

// Function to find the largest mobile integer
int getMobile(const vector<int>& perm, const vector<int>& dir) {
    int mobile = 0;
    for (int i = 0; i < perm.size(); i++) {
        int next = i + dir[perm[i] - 1];
        if (next >= 0 && next < perm.size() && perm[i] > perm[next]) {
            if (perm[i] > mobile) {
                mobile = perm[i];
            }
        }
    }
    return mobile;
}

// Function to find the index of a given element in the permutation
int findPosition(const vector<int>& perm, int mobile) {
    for (int i = 0; i < perm.size(); i++) {
        if (perm[i] == mobile)
            return i;
    }
    return -1;
}

// Johnson-Trotter algorithm
void generatePermutations(int n) {
    vector<int> perm(n), dir(n, LEFT);

    // Initialize permutation and directions
    for (int i = 0; i < n; i++) {
        perm[i] = i + 1;
    }

    printPermutation(perm);
```

```
    while (true) {
        int mobile = getMobile(perm, dir);
        if (mobile == 0)
            break;

        int pos = findPosition(perm, mobile);
        int swapPos = pos + dir[mobile - 1];

        // Swap the mobile number with its adjacent element
        swap(perm[pos], perm[swapPos]);
        swap(dir[perm[swapPos] - 1], dir[perm[pos] - 1]);

        // Reverse direction of all elements greater than the mobile number
        for (int i = 0; i < n; i++) {
            if (perm[i] > mobile)
                dir[perm[i] - 1] = -dir[perm[i] - 1];
        }

        printPermutation(perm);
    }
}

// Main function
int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    cout << "Permutations using Johnson-Trotter algorithm:" << endl;
    generatePermutations(n);

    return 0;
}
```

Result:

Input:

```typescript
Enter the number of elements: 3
```

Output:

```cpp
Permutations using Johnson-Trotter algorithm:
1 2 3
1 3 2
3 1 2
3 2 1
2 3 1
2 1 3
```

Program -8

Question: Fractional Knapsack Algorithm

Code:

```c
#include <stdio.h>
#include <stdlib.h>

// Structure to represent an item
typedef struct {
    int weight;
    int value;
    double ratio;
} Item;

// Comparison function to sort items by value-to-weight ratio in descending order
int compare(const void *a, const void *b) {
    Item *item1 = (Item *)a;
    Item *item2 = (Item *)b;

    if (item1->ratio < item2->ratio)
        return 1;
    else if (item1->ratio > item2->ratio)
        return -1;
    return 0;
}
```

```c
// Function to solve the fractional knapsack problem
double fractionalKnapsack(int capacity, Item items[], int n) {
    // Sort items by their value-to-weight ratio
    for (int i = 0; i < n; i++) {
        items[i].ratio = (double)items[i].value / items[i].weight;
    }

    // Sort items by their ratio in decreasing order
    qsort(items, n, sizeof(Item), compare);

    double totalValue = 0.0;  // Variable to store the total value of the
knapsack
    int currentWeight = 0;    // Current weight of the knapsack

    // Loop through the items
    for (int i = 0; i < n; i++) {
        if (currentWeight + items[i].weight <= capacity) {
            // If the whole item can be taken, take it
            currentWeight += items[i].weight;
            totalValue += items[i].value;
        } else {
            // If the item cannot be taken whole, take the fraction that fits
            int remainingCapacity = capacity - currentWeight;
            totalValue += items[i].value * ((double)remainingCapacity /
items[i].weight);
            break;  // The knapsack is full
        }
    }

    return totalValue;
}

// Driver code
int main() {
    int n, capacity;

    // Take the number of items and knapsack capacity as input
    printf("Enter the number of items: ");
    scanf("%d", &n);

    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);

    Item items[n];

    // Take item values and weights as input
    for (int i = 0; i < n; i++) {
        printf("Enter weight and value for item %d: ", i + 1);
        scanf("%d %d", &items[i].weight, &items[i].value);
```

```
    }

    // Get the maximum value that can be obtained in the knapsack
    double maxValue = fractionalKnapsack(capacity, items, n);

    printf("Maximum value in the knapsack = %.2f\n", maxValue);

    return 0;
}
```

Result:

```
Enter the number of items: 3
Enter the capacity of the knapsack: 50
Enter weight and value for item 1: 10 60
Enter weight and value for item 2: 20 100
Enter weight and value for item 3: 30 120
Maximum value in the knapsack = 240.00
PS C:\Users\student\Desktop\os_207>
```

Program -9

Question: Leetcode – 1903

1903. Largest Odd Number in String

Easy | Topics | Companies | Hint

You are given a string `num`, representing a large integer. Return *the **largest-valued odd** integer (as a string) that is a **non-empty substring** of `num`, or an empty string `""` if no odd integer exists.*

A **substring** is a contiguous sequence of characters within a string.

Example 1:

Code:

```cpp
class Solution {
public:
    string largestOddNumber(string num) {
        if (num.back() % 2 == 1) return num;
        int i = num.length() - 1;
        while (i >= 0) {
            int n = num[i];
            if (n % 2 == 1) return num.substr(0, i + 1);
            i--;
        }
        return "";
    }
};
```

Result:

Program -10

Question: Heap Sort

Code:

```c
#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to heapify a subtree rooted at node i
void heapify(int arr[], int n, int i) {
    int largest = i;   // Initialize largest as root
    int left = 2 * i + 1;  // Left child index
    int right = 2 * i + 2; // Right child index

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // If largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);

        // Recursively heapify the affected subtree
        heapify(arr, n, largest);
    }
}

// Function to implement heap sort
void heapSort(int arr[], int n) {
    // Build a max heap
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // One by one extract elements from heap
```

```
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to end
        swap(&arr[0], &arr[i]);

        // Call heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// Main function
int main() {
    int n;

    // Take user input for the array size
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Take user input for the array elements
    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Call heapSort to sort the array
    heapSort(arr, n);

    // Print the sorted array
    printf("Sorted array (Heap Sort): ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Result:

```
Enter the number of elements: 10
Enter the elements:
1 3 2 5 9 8 7 10 6 4
Sorted array (Heap Sort): 1 2 3 4 5 6 7 8 9 10
PS C:\Users\student\Desktop\os_207>
```

Program -11

Question: Floyd Warshall Algorithm

Code:

```
#include <stdio.h>
#include <limits.h>

#define MAX 100
#define INF INT_MAX  // Infinite distance value

// Function to implement Floyd-Warshall algorithm
void floydWarshall(int graph[MAX][MAX], int V) {
    // dist[][] will be the output matrix that contains the shortest distances
    int dist[V][V];

    // Initialize the solution matrix with the input graph
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (graph[i][j] == 0 && i != j) {
                dist[i][j] = INF;  // No path exists
            } else {
                dist[i][j] = graph[i][j];
            }
        }
    }

    // Floyd-Warshall algorithm to compute the shortest path
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] +
dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
```

```
                }
            }
        }
    }

    // Printing the result
    printf("Shortest distances between every pair of vertices:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF) {
                printf("INF ");
            } else {
                printf("%d ", dist[i][j]);
            }
        }
        printf("\n");
    }
}

// Main function
int main() {
    int V, E;

    printf("Enter the number of vertices: ");
    scanf("%d", &V);

    printf("Enter the number of edges: ");
    scanf("%d", &E);

    int graph[MAX][MAX] = {0};  // Initialize graph with 0 (no path)

    // Take input for the edges
    printf("Enter the edges (u v weight):\n");
    for (int i = 0; i < E; i++) {
        int u, v, weight;
        printf("Edge %d: ", i + 1);
        scanf("%d %d %d", &u, &v, &weight);
        graph[u][v] = weight;
        // If the graph is undirected, uncomment the following line:
        // graph[v][u] = weight;
    }

    // Run Floyd-Warshall algorithm
    floydWarshall(graph, V);

    return 0;
}
```

Result:

```
PS C:\Users\student\Desktop\os_207> cd "c:\Users\student\Desktop\os_207\" ; if ($?) { g++ ada.cpp -o ada } ; if ($?) { .\ada }
Enter the number of vertices: 4
Enter the number of edges: 6
Enter the edges (u v weight):
Edge 1: 0 1 5
Edge 2: 0 2 10
Edge 3: 1 2 2
Edge 4: 1 3 3
Edge 5: 2 3 1
Edge 6: 3 0 7
Shortest distances between every pair of vertices:
0 5 7 8
10 0 2 3
8 13 0 1
7 12 14 0
PS C:\Users\student\Desktop\os_207>
```

Program -12

Question: Leetcode 847

847. Shortest Path Visiting All Nodes

Solved

Hard | Topics | Companies

You have an undirected, connected graph of `n` nodes labeled from `0` to `n - 1`. You are given an array `graph` where `graph[i]` is a list of all the nodes connected with node `i` by an edge.

Return *the length of the shortest path that visits every node*. You may start and stop at any node, you may revisit nodes multiple times, and you may reuse edges.

Code:

Code

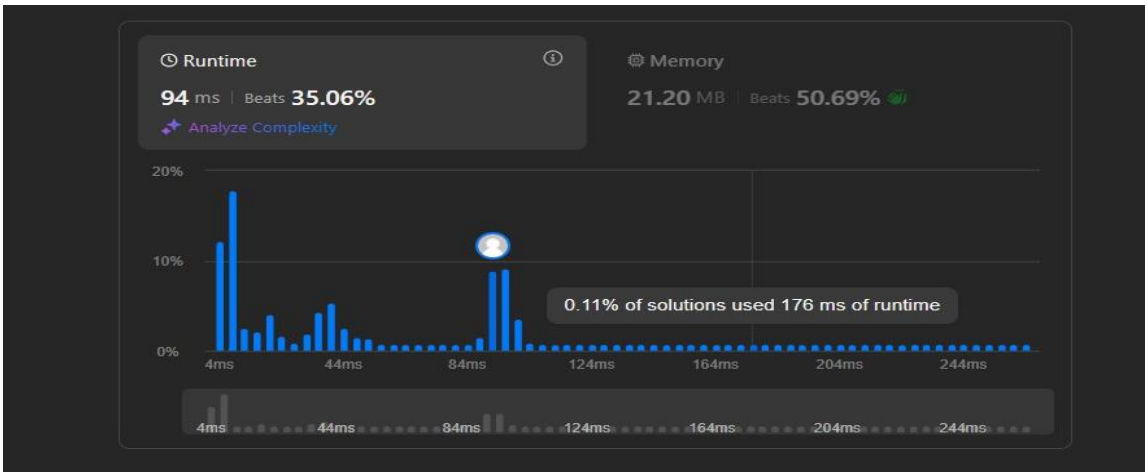
C++ | Auto

```cpp
class Solution {
public:
    // Class to carry three values at a time
    class tuple
    {
        public:
        int node; // on which current node we are standing
        int mask; // mask of that node
        int cost; // cost of path explore by this node
        tuple(int node, int mask, int cost)
        {
            this -> node = node;
            this -> mask = mask;
            this -> cost = cost;
        }
    };

    int shortestPathLength(vector<vector<int>>& graph) {
        // total number of nodes present
        int n = graph.size(); // extracting size of graph


        queue<tuple> q; // queue of class tuple type

         // set to take care which path we have already visited
        set<pair<int, int>> vis;

        int all = (1 << n) - 1; // if all nodes visited then

        // we don't know which node will give us shortest path so intially for all nodes we will store in our queue
        for(int i = 0; i < n; i++)
        {
            int maskValue = (1 << i); // 2 ^ i

            tuple thisNode(i, maskValue, 1); // make a tuple for every nod

            q.push(thisNode); // push tuple into our queue

            vis.insert({i, maskValue}); // also update into our set
        }

        // Implementing BFS
        while(q.empty() == false) // until queue is not empty
        {
```

Saved

```cpp
        }

        // Implementing BFS
        while(q.empty() == false) // until queue is not empty
        {
            tuple curr = q.front(); // extracting front tuple
            q.pop(); // pop from queuu

            // if mask value becomes all, that means we have visited all of our nodes, so from here return cost - 1
            if(curr.mask == all)
            {
                return curr.cost - 1;
            }

            // if not, start exploring in its adjcant
            for(auto &adj: graph[curr.node])
            {
                int bothVisitedMask = curr.mask; // current mask

                // we are moving from one node o anthor node
                bothVisitedMask = bothVisitedMask | (1 << adj);

                // make tuple of this path
                tuple ThisNode(adj, bothVisitedMask, curr.cost + 1);

                // if this path is not explored i.e
                // if it is not present in our set then,
                if(vis.find({adj, bothVisitedMask}) == vis.end())
                {
                    vis.insert({adj, bothVisitedMask}); // insert into set

                    q.push(ThisNode); // also insert into queue
                }
            }

        }

        // return -1, but this condition will never come
        return -1;
    }
};
```

Result:



Program -13

Question: N-Queens Algorithm

Code:

```c
#include <stdio.h>
#include <stdbool.h>

#define MAX 100

// Function to print the chessboard
void printSolution(int board[MAX][MAX], int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", board[i][j]);
        }
        printf("\n");
    }
}

// Function to check if a queen can be placed on board[row][col]
// It checks the row, upper diagonal, and lower diagonal
bool isSafe(int board[MAX][MAX], int row, int col, int N) {
    // Check this column on the upper side
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 1) {
            return false;
        }
    }

    // Check upper diagonal on the left side
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    // Check upper diagonal on the right side
    for (int i = row - 1, j = col + 1; i >= 0 && j < N; i--, j++) {
        if (board[i][j] == 1) {
            return false;
```

```
        }
    }

    return true;
}

// Function to solve the N-Queens problem using backtracking
bool solveNQueens(int board[MAX][MAX], int row, int N) {
    // If all queens are placed, return true
    if (row >= N) {
        return true;
    }

    // Consider this row and try all columns
    for (int col = 0; col < N; col++) {
        // Check if it's safe to place the queen
        if (isSafe(board, row, col, N)) {
            // Place the queen
            board[row][col] = 1;

            // Recur to place the rest of the queens
            if (solveNQueens(board, row + 1, N)) {
                return true;
            }

            // If placing queen in board[row][col] doesn't lead to a solution,
remove it (backtrack)
            board[row][col] = 0;
        }
    }

    // If the queen cannot be placed in any column in this row, return false
    return false;
}

// Main function
int main() {
    int N;

    // Take user input for the number of queens
    printf("Enter the number of queens: ");
    scanf("%d", &N);

    // Initialize the board with 0s
    int board[MAX][MAX] = {0};

    // Solve the N-Queens problem
    if (solveNQueens(board, 0, N)) {
        // Print the solution if a solution is found
```

```
        printf("Solution:\n");
        printSolution(board, N);
    } else {
        printf("No solution exists for %d queens.\n", N);
    }

    return 0;
}
```

Result:

```
PS C:\Users\student\Desktop\os_207> cd "c:\Users\student\Desktop\os_207\" ; if ($?) { g++ ada.cpp -o ada } ; if ($?) { .\ada }
Enter the number of queens: 4
Solution:
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
PS C:\Users\student\Desktop\os_207>
```