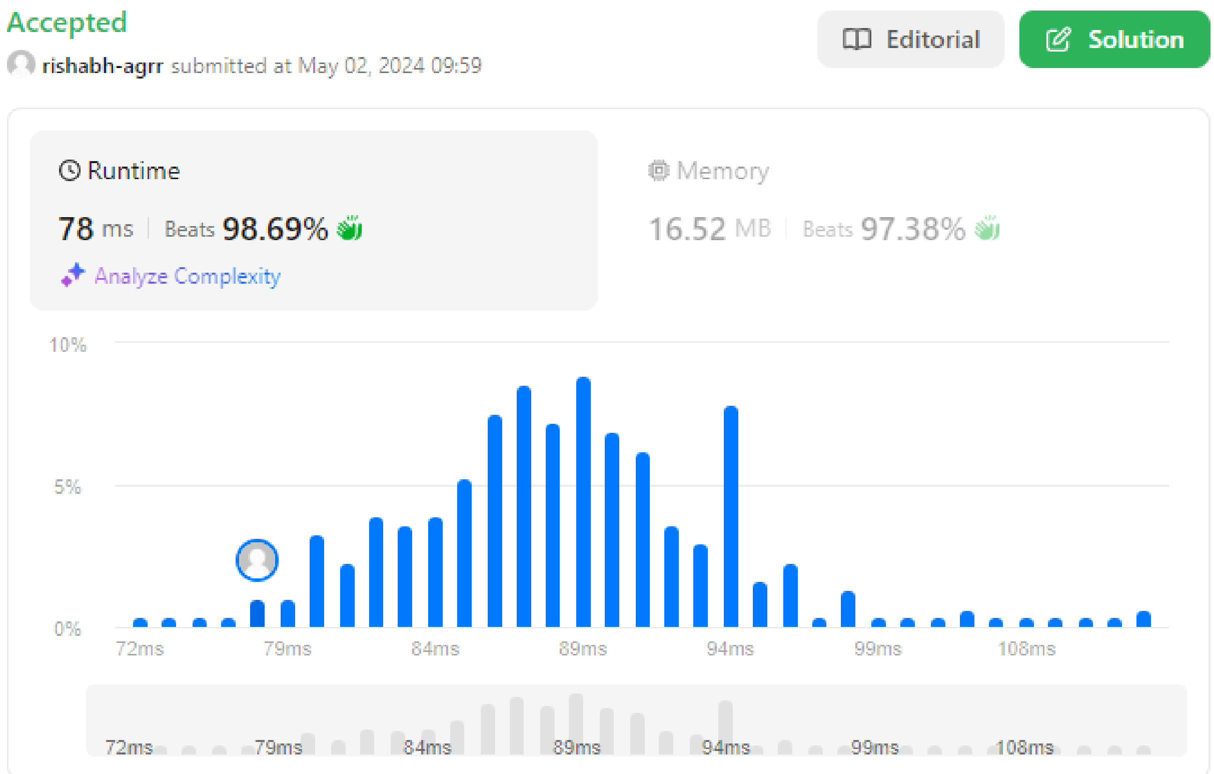
ADA LAB REPORT

Name: Rishabh Kumar
USN: 1BM22CS221
Section: 4D
Faculty In-Charge: Dr. Madhavi P

ADA Week-1

(Q) LeetCode – 448 - Find all disappeared numbers in an array

```
int* findDisappearedNumbers(int* nums, int numsSize, int* returnSize) {
    int temp = 0;
    for (int index = 0; index < numsSize; ++index) {
        temp = abs(nums[index]) - 1;
        nums[temp] = abs(nums[temp]) * -1;
    }
    int insert_index = 0;
    *returnSize = 0;
    for (int index = 0; index < numsSize; ++index) {
        if (nums[index] > 0) {
            ++*returnSize;
            nums[insert_index++] = index + 1;
        }
    }
    return nums;
}
```

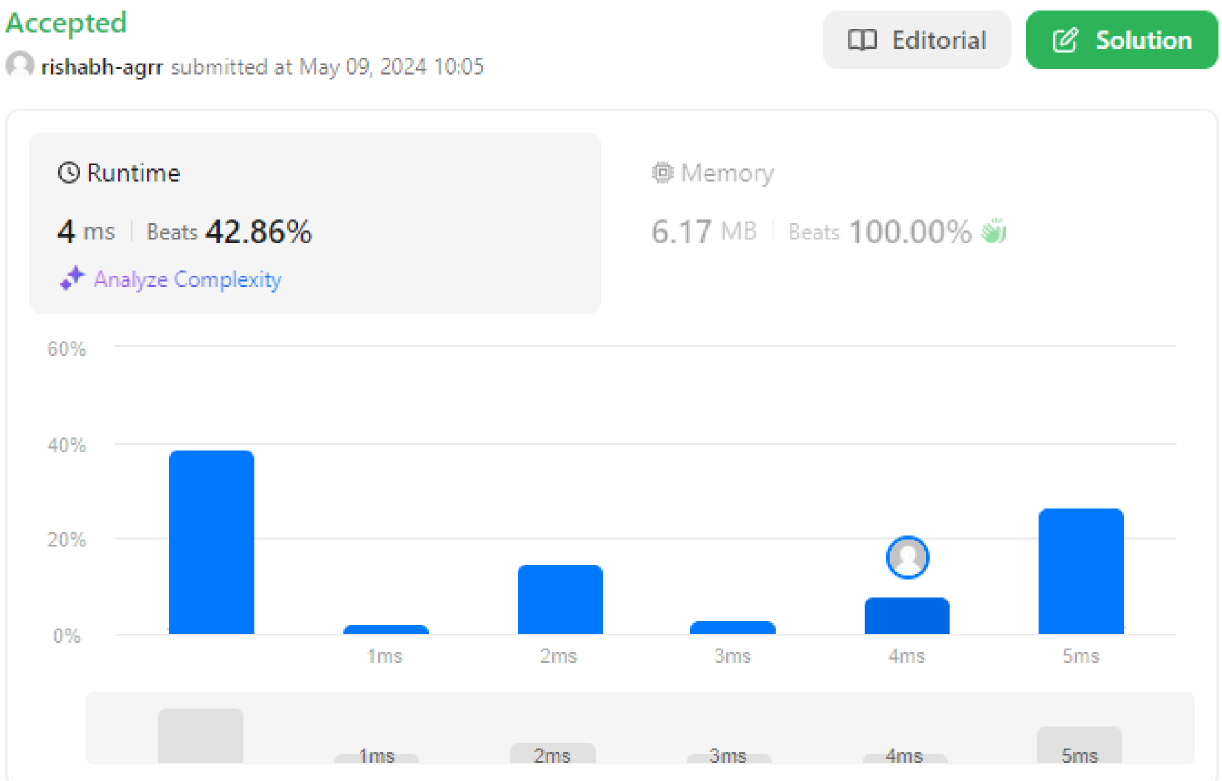


ADA Week-2

(Q) LeetCode – 103 - Binary Tree ZigZag level Order Traversal

```
int** zigzagLevelOrder(struct TreeNode* root, int* returnSize, int** returnColumnSizes) {
    int th = funheight(root);
    int** ret = (int**)calloc(th, sizeof(int*));
    *returnSize = th;
    (*returnColumnSizes) = (int*)calloc(th, sizeof(int));
    for(int i= 0;i<th; i++){
        int cnt = 0;
        ret[i] = (int*)calloc(1<<i, sizeof(int));
        if(i%2){//odd
            funR(root, i, ret[i], &cnt);
        }else{
            funL(root, i, ret[i], &cnt);
        }
        (*returnColumnSizes)[i] = cnt;
```

```
    }
    return ret;
}
```

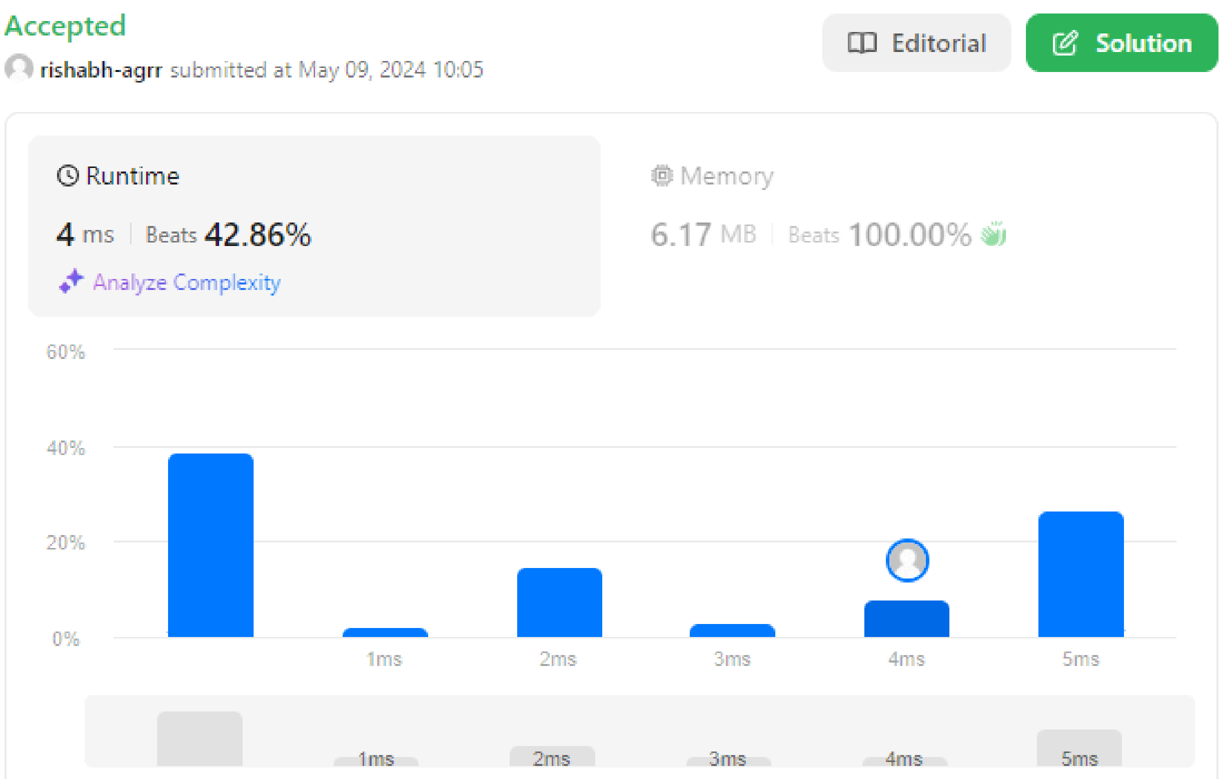


ADA Week-3

(Q) LeetCode – 897 - Increasing Order Search Tree

```
struct TreeNode *createNode(int val)
{
    struct TreeNode *n = malloc(sizeof(struct TreeNode));
    n->val = val;
    n->left = NULL;
    n->right = NULL;

    return n;
}

void fillRightTree(struct TreeNode **tree, struct TreeNode *node)
{
    if (!node)
```

```
    {
        return;
    }

    fillRightTree(tree, node->left);
    (*tree)->right = createNode(node->val);
    *tree = (*tree)->right;
    fillRightTree(tree, node->right);
}

struct TreeNode* increasingBST(struct TreeNode* root) {
    struct TreeNode *dummyRoot = createNode(0);
    struct TreeNode *newTree = dummyRoot;

    fillRightTree(&newTree, root);

    struct TreeNode *rightTreeRoot = dummyRoot->right;
    free(dummyRoot);  // Free the dummy root

    return rightTreeRoot;
}
```



ADA Week - 4

(Q) A program to implement Topological Sort Order

```
// topological sort

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

#define MAX_VERTICES 100

// Adjacency list node
struct Node {
    int vertex;
    struct Node* next;
};

// Graph with adjacency list representation
struct Graph {
    int numVertices;
    struct Node** adjList;
    int* inDegree;
};

// Function to create a new node
struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Function to create a graph with 'V' vertices
struct Graph* createGraph(int V) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = V;
    graph->adjList = (struct Node**)malloc(V * sizeof(struct Node*));
    graph->inDegree = (int*)malloc(V * sizeof(int));

    for (int i = 0; i < V; i++) {
        graph->adjList[i] = NULL;
        graph->inDegree[i] = 0;
    }

    return graph;
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
```

```
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;

    graph->inDegree[dest]++;
}

// Function to perform Kahn's algorithm for topological sorting
void topologicalSortKahn(struct Graph* graph) {
    int V = graph->numVertices;
    int* inDegree = graph->inDegree;

    // Initialize a queue for Kahn's algorithm
    int queue[MAX_VERTICES];
    int front = 0, rear = 0;

    // Enqueue vertices with in-degree 0
    for (int i = 0; i < V; i++) {
        if (inDegree[i] == 0)
            queue[rear++] = i;
    }

    int count = 0; // Count of visited vertices

    // Initialize topological order
    int topologicalOrder[V];

    while (front < rear) {
        int u = queue[front++];
        topologicalOrder[count++] = u;

        // Iterate through all adjacent vertices of u
        struct Node* temp = graph->adjList[u];
        while (temp != NULL) {
            int v = temp->vertex;
            // Decrease in-degree of adjacent vertex
            inDegree[v]--;

            // If in-degree becomes 0, add to queue
            if (inDegree[v] == 0)
                queue[rear++] = v;

            temp = temp->next;
        }
    }

    // Check if there was a cycle
    if (count != V) {
        printf("Graph has a cycle!\n");
        return;
    }

    // Print topological order
    printf("Topological Sort (Kahn's Algorithm): ");
    for (int i = 0; i < V; i++)
```

```
        printf("%d ", topologicalOrder[i]);
    printf("\n");
}

// Function to perform Depth-First Search (DFS)
void DFS(struct Graph* graph, int v, int visited[], int* index, int topologicalOrder[]) {
    visited[v] = 1;

    // Recur for all the vertices adjacent to this vertex
    struct Node* temp = graph->adjList[v];
    while (temp != NULL) {
        if (!visited[temp->vertex])
            DFS(graph, temp->vertex, visited, index, topologicalOrder);
        temp = temp->next;
    }

    // Store the vertex in the topological order
    topologicalOrder[*index] = v;
    (*index)--;
}

// Function to perform topological sorting using DFS
void topologicalSortDFS(struct Graph* graph) {
    int V = graph->numVertices;
    int* visited = (int*)malloc(V * sizeof(int));
    int topologicalOrder[V];
    int index = V - 1;

    // Initialize all vertices as not visited
    for (int i = 0; i < V; i++)
        visited[i] = 0;

    // Perform DFS for each unvisited vertex
    for (int i = 0; i < V; i++) {
        if (!visited[i])
            DFS(graph, i, visited, &index, topologicalOrder);
    }

    // Print topological order
    printf("Topological Sort (DFS Algorithm): ");
    for (int i = 0; i < V; i++)
        printf("%d ", topologicalOrder[i]);
    printf("\n");

    free(visited);
}

// Main function
int main() {
    int V = 6;
    struct Graph* graph = createGraph(V);

    addEdge(graph, 5, 2);
    addEdge(graph, 5, 0);
```

```
    addEdge(graph, 4, 0);
    addEdge(graph, 4, 1);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 1);

    printf("Graph:\n");
    for (int i = 0; i < V; i++) {
        struct Node* temp = graph->adjList[i];
        printf("Vertex %d: ", i);
        while (temp != NULL) {
            printf("%d ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }

    topologicalSortKahn(graph);
    topologicalSortDFS(graph);

    return 0;
}
```

OUTPUT :-

```
Topological Sort (Kahn's Algorithm): 4 5 2 0 3 1
Topological Sort (DFS Algorithm): 4 0 5 2 3 1
```

ADA Week - 5

(Q) A program to implement and compare selection sort and merge sort.

```
// selection sort with timings


#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to swap two elements
```

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, min_idx;

    for (i = 0; i < n - 1; i++) {
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        swap(&arr[min_idx], &arr[i]);
    }
}

// Function to generate an array of random integers
void generateRandomArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        arr[i] = rand();
}

// Function to measure execution time of selectionSort
double measureSelectionSortTime(int arr[], int n) {
    clock_t start = clock();
    selectionSort(arr, n);
    clock_t end = clock();
    return ((double)(end - start)) / CLOCKS_PER_SEC;
}

// Main function to test Selection Sort with varying input sizes
int main() {
    FILE *fp;
    fp = fopen("selection_sort_time.csv", "w");
    if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(fp, "Input Size,Time Taken (seconds)\n");

    // Test selection sort with various input sizes
    for (int size = 1000; size <= 10000; size += 1000) {
        int arr[size];
        generateRandomArray(arr, size);

        double time_taken = measureSelectionSortTime(arr, size);
        printf("Input Size: %d, Time Taken: %f seconds\n", size, time_taken);

        fprintf(fp, "%d,%f\n", size, time_taken);
```

```
    }

    fclose(fp);
    printf("Data written to selection_sort_time.csv\n");

    return 0;
}
```

```
// merge sort with timimngs


#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to merge two subarrays arr[low..mid] and arr[mid+1..high]
void merge(int arr[], int low, int mid, int high) {
    int n1 = mid - low + 1;
    int n2 = high - mid;

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

    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[low + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temporary arrays back into arr[low..high]
    int i = 0, j = 0, k = low;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if any
    while (j < n2) {
```

```
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Function to perform Merge Sort
void mergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2; // Avoids overflow for large low and high
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

// Function to generate an array of random integers
void generateRandomArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        arr[i] = rand();
}

// Function to measure execution time of mergeSort
double measureMergeSortTime(int arr[], int n) {
    clock_t start = clock();
    mergeSort(arr, 0, n - 1);
    clock_t end = clock();
    return ((double)(end - start)) / CLOCKS_PER_SEC;
}

// Main function to test Merge Sort with varying input sizes
int main() {
    FILE *fp;
    fp = fopen("merge_sort_time.csv", "w");
    if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(fp, "Input Size,Time Taken (seconds)\n");

    // Test merge sort with various input sizes
    for (int size = 1000; size <= 10000; size += 1000) {
        int arr[size];
        generateRandomArray(arr, size);

        double time_taken = measureMergeSortTime(arr, size);
        printf("Input Size: %d, Time Taken: %f seconds\n", size, time_taken);

        fprintf(fp, "%d,%f\n", size, time_taken);
    }

    fclose(fp);
    printf("Data written to merge_sort_time.csv\n");
```

```
    return 0;
}
```

OUTPUT :-

```
Selection Sort:
Input Size: 1000, Time Taken: 0.003555 seconds
Input Size: 2000, Time Taken: 0.011786 seconds
...

Merge Sort:
Input Size: 1000, Time Taken: 0.000083 seconds
Input Size: 2000, Time Taken: 0.000203 seconds
```

ADA Week - 6

(Q) A program to implement Quick Sort.

```
// quick sort


#include <stdio.h>

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

// Function to partition the array using the last element as pivot
int partition(int arr[], int low, int high) {
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
```

```c
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot) {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Function to implement QuickSort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi is partitioning index, arr[p] is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Main function to test the QuickSort algorithm
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Unsorted array: \n");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

OUTPUT :-

```
Unsorted array:
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10
```

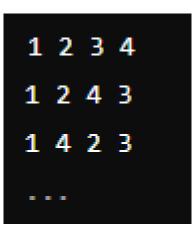
ADA Week - 7

(Q) A program to implement Jhonson-Trotter.

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

#define MAXN 10

int p[MAXN]; // p[i] holds the position of i in the permutation
int dir[MAXN]; // dir[i] = -1 if i is mobile to the left, +1 if mobile to the right

void printPermutation(int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", p[i]);
    }
    printf("\n");
}

int findLargestMobile(int n) {
    int mobile = 0;
    int mobileIndex = -1;

    for (int i = 0; i < n; i++) {
        if ((dir[i] == -1 && i > 0 && p[i] > p[i-1]) || // mobile to the left
            (dir[i] == +1 && i < n-1 && p[i] > p[i+1])) { // mobile to the right
            if (p[i] > mobile) {
                mobile = p[i];
                mobileIndex = i;
            }
        }
    }

    return mobileIndex;
}

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void johnsonTrotter(int n) {
    // Initialize permutation and direction arrays
    for (int i = 0; i < n; i++) {
        p[i] = i + 1; // Initial permutation: 1 2 3 ... n
        dir[i] = -1; // All elements are initially mobile to the left
    }

    // Print the initial permutation
    printPermutation(n);
```

```
    // Find the largest mobile integer and swap it
    int mobileIndex = findLargestMobile(n);
    while (mobileIndex != -1) {
        // Swap p[mobileIndex] with the adjacent element it is pointing to
        int nextIndex = mobileIndex + dir[mobileIndex];
        swap(&p[mobileIndex], &p[nextIndex]);

        // Swap corresponding directions
        swap(&dir[mobileIndex], &dir[nextIndex]);

        // Print the new permutation
        printPermutation(n);

        // Update directions of elements greater than the current largest mobile integer
        for (int i = 0; i < n; i++) {
            if (p[i] > p[nextIndex]) {
                dir[i] = -dir[i];
            }
        }

        // Find the next largest mobile integer
        mobileIndex = findLargestMobile(n);
    }
}

int main() {
    int n;
    printf("Enter the value of n (<= %d): ", MAXN);
    scanf("%d", &n);

    johnsonTrotter(n);

    return 0;
}
```

<u>OUTPUT :-</u>



```
1 2 3 4
1 2 4 3
1 4 2 3
...
```

(Q) A program to implement String Matching using Brute-Force Technique .

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

```
void bruteForceSubstringSearch(char *text, char *pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int i, j;

    for (i = 0; i <= n - m; i++) {
        for (j = 0; j < m; j++) {
            if (text[i + j] != pattern[j])
                break;
        }
        if (j == m) {
            printf("Pattern found at index %d\n", i);
        }
    }
}

int main() {
    char text[100], pattern[100];

    printf("Enter the main text: ");
    scanf("%s", text);

    printf("Enter the pattern to search: ");
    scanf("%s", pattern);

    bruteForceSubstringSearch(text, pattern);

    return 0;
}
```

OUTPUT :-

```
Pattern found at index 0
Pattern found at index 4
```

ADA Week - 8

(Q) A program to implement Heap Sort.

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

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

    // 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 != root) {
        // Swap root with largest
        int temp = arr[root];
        arr[root] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Function to perform Heap Sort
void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

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

int main() {
    int N;
    printf("Enter number of elements: ");
    scanf("%d", &N);

    int arr[N];
```

```
    printf("Enter %d integers: ", N);
    for (int i = 0; i < N; i++)
        scanf("%d", &arr[i]);

    clock_t start_time = clock();

    heapSort(arr, N);

    clock_t end_time = clock();
    double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

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

    printf("Time taken: %f seconds\n", time_taken);

    return 0;
}
```

<u>OUTPUT :-</u>

```
Sorted array using Heap Sort: 1 3 5 7 8 9 10
Time taken: 0.000028 seconds
```

(Q) A program to implement Floyd's Algorithm.

```
#include <stdio.h>

#define INF 99999
#define V 4  // Number of vertices in the graph

void printSolution(int dist[][V]) {
    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("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

void floydWarshall(int graph[][V]) {
    int dist[V][V];
```

```
    // Initialize distances to the input graph's distances
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    // Update distances by considering all vertices as intermediate vertex one by one
    for (int k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (int i = 0; i < V; i++) {
            // Pick all vertices as destination for the above picked source
            for (int j = 0; j < V; j++) {
                // If vertex k is on the shortest path from i to j, then update the value of dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    printSolution(dist);
}

int main() {
    int graph[V][V] = {
        {0, INF, 3, INF},
        {2, 0, INF, INF},
        {INF, 7, 0, 1},
        {6, INF, INF, 0}
    };

    floydWarshall(graph);

    return 0;
}
```

OUTPUT :-

```
  0      INF      3      INF
  2       0      INF     INF
 INF      7       0       1
  6      INF     INF      0
```

ADA Week - 9

(Q) A program to implement KnapSack using Dynamic Programming.

```
#include <stdio.h>

#define N 4

int max(int a, int b) {
    return (a > b) ? a : b;
}

void knapsack(int weights[], int profits[], int W) {
    int dp[N+1][W+1];

    // Initialize the dp array
    for (int i = 0; i <= N; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (weights[i-1] <= w)
                dp[i][w] = max(profits[i-1] + dp[i-1][w-weights[i-1]], dp[i-1][w]);
            else
                dp[i][w] = dp[i-1][w];
        }
    }

    // Maximum profit will be in dp[N][W]
    printf("Maximum profit: %d\n", dp[N][W]);

    // To find which items were selected
    int selected[N];
    int i = N, j = W;
    while (i > 0 && j > 0) {
        if (dp[i][j] != dp[i-1][j]) {
            selected[i-1] = 1;
            j -= weights[i-1];
        } else {
            selected[i-1] = 0;
        }
        i--;
    }

    // Display selected items
    printf("Objects selected in knapsack:\n");
    for (int i = 0; i < N; i++) {
        if (selected[i])
            printf("Object %d (Weight: %d, Profit: %d)\n", i + 1, weights[i], profits[i]);
    }
}

int main() {
```

```
    int weights[] = {2, 1, 3, 2};
    int profits[] = {12, 10, 20, 15};
    int W = 5; // Knapsack capacity

    knapsack(weights, profits, W);

    return 0;
}
```

<u>OUTPUT :-</u>

```
Maximum profit: 37
Objects selected in knapsack:
Object 1 (Weight: 2, Profit: 12)
Object 3 (Weight: 3, Profit: 20)
Object 4 (Weight: 2, Profit: 15)
```

(Q) A program to implement Prim's Algorithm.

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

#define V 5 // Number of vertices in the graph

// A utility function to find the vertex with minimum key value,
// from the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented using adjacency matrix representation
```

```
void primMST(int graph[V][V])
{
    int parent[V]; // Array to store constructed MST
    int key[V];    // Key values used to pick minimum weight edge in cut
    bool mstSet[V]; // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 0th vertex in MST.
    key[0] = 0;     // Make key 0 so that this vertex is picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // 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);

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

        // Update key value and parent index of the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // Print the constructed MST
    printMST(parent, graph);
}

int main()
{
    /* Let us create the following graph
          2    3
       (0)--(1)--(2)
       |   / \   |
      6| 8/   \5 |7
       | /     \ |
       (3)-------(4)
            9          */
    int graph[V][V] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0},
    };
```

```
    // Print the solution
    primMST(graph);

    return 0;
}
```

OUTPUT :-

```
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   7
```

ADA Week - 10

(Q) A program to implement Kruskal's Algorithm.

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

// Structure to represent an edge in the graph
struct Edge {
    int src, dest, weight;
};

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

// Function prototypes
int find(struct Subset subsets[], int i);
void Union(struct Subset subsets[], int x, int y);
int compareEdges(const void* a, const void* b);
void KruskalMST(int V, int E, struct Edge edges[]);

// Function to find the root of a node
int find(struct Subset subsets[], int i) {
```

```c
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// Function to perform union of two subsets
void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Comparison function for qsort
int compareEdges(const void* a, const void* b) {
    struct Edge* edge1 = (struct Edge*)a;
    struct Edge* edge2 = (struct Edge*)b;
    return edge1->weight - edge2->weight;
}

// Function to construct and print MST using Kruskal's algorithm
void KruskalMST(int V, int E, struct Edge edges[]) {
    struct Edge result[V];  // To store the resultant MST
    int e = 0;              // Index variable for result[]
    int i = 0;              // Index variable for sorted edges

    // Step 1: Sort all the edges in non-decreasing order of their weight
    qsort(edges, E, sizeof(edges[0]), compareEdges);

    // Allocate memory for creating V subsets
    struct Subset* subsets = (struct Subset*)malloc(V * sizeof(struct Subset));

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is V-1
    while (e < V - 1 && i < E) {
        // Step 2: Pick the smallest edge. Increment index for next iteration
        struct Edge next_edge = edges[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does not cause a cycle, include it
```

```
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    // Print the edges of MST
    printf("Edges in MST:\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d : %d\n", result[i].src, result[i].dest, result[i].weight);
}

// Driver program to test above functions
int main() {
    int V, E;
    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &V, &E);

    struct Edge edges[E];

    printf("Enter edges (src, dest, weight):\n");
    for (int i = 0; i < E; ++i)
        scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);

    KruskalMST(V, E, edges);

    return 0;
}
```

OUTPUT :-

```
Edges in MST:
1 -- 2 : 1
2 -- 3 : 1
3 -- 4 : 3
4 -- 5 : 2
0 -- 1 : 2
```

(Q) A program to implement Djikstra Algorithm .

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

```
#include <limits.h>

#define V 6  // Number of vertices in the graph

// Function to find the vertex with the minimum distance value, from the set of vertices
// not yet included in shortest path tree
int minDistance(int dist[], int sptSet[]) {
    int min = INT_MAX, min_index;

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

    return min_index;
}

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

// Function that implements Dijkstra's single source shortest path algorithm for a graph
// represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src) {
    int dist[V];     // The output array. dist[i] will hold the shortest distance from src to i

    int sptSet[V];   // sptSet[i] will be 1 if vertex i is included in shortest path tree or shortest distance from src to i is
finalized

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

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices not yet processed.
        // u is always equal to src in the first iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = 1;

        // Update dist value of the adjacent vertices of the picked vertex.
        for (int v = 0; v < V; v++) {
```

```
            // Update dist[v] only if is not in sptSet, there is an edge from u to v,
            // and total weight of path from src to v through u is smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
        }
    }

    // Print the constructed distance array
    printSolution(dist);
}

// Driver program to test above function
int main() {
    // Example graph represented using adjacency matrix
    int graph[V][V] = {
        {0, 4, 0, 0, 0, 0},
        {4, 0, 8, 0, 0, 0},
        {0, 8, 0, 7, 0, 4},
        {0, 0, 7, 0, 9, 14},
        {0, 0, 0, 9, 0, 10},
        {0, 0, 4, 14, 10, 0}
    };

    dijkstra(graph, 0);  // Find shortest paths from source vertex 0

    return 0;
}
```

<u>OUTPUT :-</u>

```
Vertex   Distance from Source
0        0
1        4
2        12
3        19
4        21
5        20
```