VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT

on

Analysis and Design of Algorithms

(23CS4PCADA)

Submitted by:

Chethan N (1BM23CS075)

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

April 2025 - July 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**” carried out by **Chethan N (1BM23CS075),** who is 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 2024-25. The Lab report has been approved as it satisfies the academic requirements in respect of **Analysis and Design of Algorithms - (23CS4PCADA)** work prescribed for the said degree.

Sowmya T
Associate Professor
Department of CSE
BMSCE, Bengaluru

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

Table Of Contents

Github Link: https://github.com/chethannhub/ADA_Lab

1. Experiments

1.1 Experiment - 1

1.1.1 Question:

Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.

1.1.2 Code:

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


void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temporary arrays
    int* L = (int*)malloc(n1 * sizeof(int));
    int* R = (int*)malloc(n2 * sizeof(int));

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

    // Merge
    i = 0;
    j = 0;
    k = left;
```

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

    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];

    free(L);
    free(R);
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
```

```
}

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, size);

    mergeSort(arr, 0, size - 1);

    printf("Sorted array: ");
    printArray(arr, size);

    return 0;
}
```

1.1.3 Output:

a.

```
Original array: 38 27 43 3 9 82 10
Sorted array: 3 9 10 27 38 43 82
```

1.2 Experiment - 2

1.2.1 Question:

Sort a given set of N integer elements using Quick Sort technique and compute its time taken.

1.2.2 Code:

```
#include <stdio.h>

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

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

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

        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

```c
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);
    return 0;
}
```

1.2.3 Output:

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

1.3 Experiment - 3

1.3.1 Question:

a. Find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm.
b. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal’s algorithm.

1.3.2 Code:

a.

```
#include <stdio.h>
#include <limits.h>
#define V 5

int minKey(int key[], int mstSet[]) {
    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;
}

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]]);
}

void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    int mstSet[V];


    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = 0;

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
```

```
        int u = minKey(key, mstSet);
        mstSet[u] = 1;


        for (int v = 0; v < V; v++) {
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
        }
    }

    printMST(parent, graph);
}

int main() {
    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}
    };

    primMST(graph);

    return 0;
}
```

b.

```
#include <stdio.h>

#include <stdlib.h>


struct Edge {

    int src, dest, weight;

};


struct Graph {

    int V, E;
```

```
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
    return graph;
}

struct Subset {
    int parent;
    int rank;
};

int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

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++;
    }
}

int compareEdges(const void* a, const void* b) {
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight - b1->weight;
}

void KruskalMST(struct Graph* graph) {
    int V = graph->V;
    struct Edge result[V];
    int e = 0;
    int i = 0;

    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compareEdges);

    struct Subset* subsets = (struct Subset*)malloc(V * sizeof(struct Subset));

    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    while (e < V - 1 && i < graph->E) {
```

```
        struct Edge next = graph->edge[i++];

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

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

    printf("Edge \tWeight\n");
    for (i = 0; i < e; ++i)
        printf("%d - %d \t%d\n", result[i].src, result[i].dest, result[i].weight);

    free(subsets);
}

int main() {
    int V = 4;
    int E = 5;
    struct Graph* graph = createGraph(V, E);

    graph->edge[0] = (struct Edge){0, 1, 10};
    graph->edge[1] = (struct Edge){0, 2, 6};
    graph->edge[2] = (struct Edge){0, 3, 5};
    graph->edge[3] = (struct Edge){1, 3, 15};
    graph->edge[4] = (struct Edge){2, 3, 4};
```

```
    KruskalMST(graph);


    free(graph->edge);
    free(graph);


    return 0;
}
```

1.3.1 Output:

a.

```
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

b.

```
Edge    Weight
2 - 3   4
0 - 3   5
0 - 1   10
```

1.4 Experiment - 4

1.4.1 Question:

Write program to obtain the Topological ordering of vertices in a given digraph.

1.4.2 Code:

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

#define MAX 100

int graph[MAX][MAX];
int visited[MAX];
int stack[MAX];
int top = -1;
int n;

void addEdge(int u, int v) {
    graph[u][v] = 1;
}

void dfs(int v) {
    visited[v] = 1;
    for (int i = 0; i < n; i++) {
        if (graph[v][i] && !visited[i]) {
            dfs(i);
        }
    }
    stack[++top] = v;
}

void topologicalSort() {
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    printf("Topological Order:\n");
    while (top != -1) {
        printf("%d ", stack[top--]);
    }
    printf("\n");
}
```

```
int main() {
    int edges, u, v;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &edges);

    for (int i = 0; i < n; i++) {
        visited[i] = 0;
        for (int j = 0; j < n; j++) {
            graph[i][j] = 0;
        }
    }
    printf("Enter edges (u v) where u -> v:\n");
    for (int i = 0; i < edges; i++) {
        scanf("%d %d", &u, &v);
        addEdge(u, v);
    }
    topologicalSort();
    return 0;
}
```

1.4.3 Output:

```
Enter number of vertices: 6
Enter number of edges: 6
Enter edges (u v) where u -> v:
5 2
5 0
4 0
4 1
2 3
3 1
Topological Order:
5 4 2 3 1 0
```

1.5 Experiment - 5

1.5.1 Question:

Implement 0/1 Knapsack problem using dynamic programming.

1.5.2 Code:

```
#include <stdio.h>


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


int knapsack(int W, int weights[], int values[], int n) {

    int dp[n+1][W+1];

        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(dp[i-1][w], values[i-1] + dp[i-1][w-weights[i-1]]);
            }
            else {
                dp[i][w] = dp[i-1][w];
            }
        }
    }

   return dp[n][W];
}

int main() {
    int n, W;

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

    int values[n], weights[n];
```

```
    printf("Enter the values of the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &values[i]);
    }

    printf("Enter the weights of the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &weights[i]);
    }

    int result = knapsack(W, weights, values, n);
    printf("Maximum value in Knapsack = %d\n", result);

    return 0;
}
```

1.5.3 Output:

```
Enter number of items: 4
Enter the capacity of knapsack: 7
Enter the values of the items:
16 19 23 28
Enter the weights of the items:
2 3 4 5
Maximum value in Knapsack = 44
```

1.6 Experiment - 6

1.6.1 Question:

Implement All Pair Shortest paths problem using Floyd’s algorithm.

1.6.2 Code:

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

#define INF INT_MAX
#define MAX_VERTICES 100

void floydWarshall(int graph[MAX_VERTICES][MAX_VERTICES], int n) {
    int dist[MAX_VERTICES][MAX_VERTICES];

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j)
                dist[i][j] = 0;
            else if (graph[i][j] == 0)
                dist[i][j] = INF;
            else
                dist[i][j] = graph[i][j];
        }
    }

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF) {
                    dist[i][j] = (dist[i][j] < dist[i][k] + dist[k][j]) ? dist[i][j] : dist[i][k] + dist[k][j];
                }
            }
        }
    }

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

```
        printf("\n");
    }
}

int main() {
    int n, graph[MAX_VERTICES][MAX_VERTICES];

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

    printf("Enter the adjacency matrix (use 0 for no direct edge):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    floydWarshall(graph, n);

    return 0;
}
```

1.6.3 Output:

```
Shortest distances between every pair of vertices:
     0     3     7     5
     2     0     6     4
     3     1     0     5
     5     3     2     0
```

1.7 Experiment - 7

1.7.1 Question:

Implement Fractional Knapsack using Greedy technique.

1.7.2 Code:

```
#include <stdio.h>

typedef struct {
    int value;
    int weight;
} Item;

// Function to find the maximum of two floats
float max(float a, float b) {
    return (a > b) ? a : b;
}

// Function to sort items by value-to-weight ratio in descending order
void sortItemsByRatio(Item items[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            float r1 = (float)items[j].value / items[j].weight;
            float r2 = (float)items[j + 1].value / items[j + 1].weight;
            if (r1 < r2) {
                Item temp = items[j];
                items[j] = items[j + 1];
                items[j + 1] = temp;
            }
        }
    }
}

// Fractional Knapsack function
float fractionalKnapsack(int capacity, Item items[], int n) {
    sortItemsByRatio(items, n);

    float totalValue = 0.0;
    int currWeight = 0;

    for (int i = 0; i < n; i++) {
        if (currWeight + items[i].weight <= capacity) {
            // Take the whole item
            currWeight += items[i].weight;
```

```
            totalValue += items[i].value;
        } else {
            // Take the fraction of the remaining capacity
            int remain = capacity - currWeight;
            totalValue += ((float)items[i].value / items[i].weight) * remain;
            break;
        }
    }
    return totalValue;
}

// Driver code
int main() {
    int n = 3;
    Item items[] = {{60, 10}, {100, 20}, {120, 30}};
    int capacity = 50;

    float maxValue = fractionalKnapsack(capacity, items, n);
    printf("Maximum value in Knapsack = %.2f\n", maxValue);

    return 0;
}
```

1.7.3 Output:

```
Maximum value in Knapsack = 240.00
```

1.8 Experiment - 8

1.8.1 Question:

From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

1.8.2 Code:

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

#define V 5

int minDistance(int dist[], int visited[]) {
    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;
}

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


void dijkstra(int graph[V][V], int src) {
    int dist[V];
    int visited[V];

    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        visited[i] = 0;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
       int u = minDistance(dist, visited);
        visited[u] = 1;
```

```
        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, src);
}

int main() {
   int graph[V][V] = {
        {0, 10, 0, 0, 5},
        {0, 0, 1, 0, 2},
        {0, 0, 0, 4, 0},
        {7, 0, 6, 0, 0},
        {0, 3, 9, 2, 0}
    };

    int source = 0;
    dijkstra(graph, source);

    return 0;
}
```

1.8.3 Output:

```
Vertex  Distance from Source 0
0               0
1               8
2               9
3               7
4               5
```

1.9 Experiment - 9

1.9.1 Question:

Implement “N-Queens Problem” using Backtracking.

1.9.2 Code:

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

int is_safe(int** board, int row, int col, int n) {
    int i, j;


    for (i = 0; i < row; i++) {
        if (board[i][col] == 1)
            return 0;
    }


    for (i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1)
            return 0;
    }


    for (i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
        if (board[i][j] == 1)
            return 0;
    }

    return 1;
}

void print_board(int** board, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%c ", board[i][j] ? 'Q' : '.');
        }
        printf("\n");
    }
    printf("\n");
}

int solve_n_queens_util(int** board, int row, int n) {
    if (row == n) {
```

```
        print_board(board, n);
        return 1;
    }

    int res = 0;
    for (int col = 0; col < n; col++) {
        if (is_safe(board, row, col, n)) {
            board[row][col] = 1;
            res += solve_n_queens_util(board, row + 1, n);
            board[row][col] = 0;
        }
    }
    return res;
}

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


    int** board = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        board[i] = (int*)calloc(n, sizeof(int));
    }

    int solutions = solve_n_queens_util(board, 0, n);
    printf("Total solutions: %d\n", solutions);


    for (int i = 0; i < n; i++) {
        free(board[i]);
    }
    free(board);

    return 0;
}
```

1.9.3 Output:

```
Enter the number of queens (N): 4
. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .

Total solutions: 2
```

1.10 Experiment - 10

1.10.1 Question:

Implement Johnson Trotter algorithm to generate permutations.

1.10.2 Code:

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


#define LEFT -1
#define RIGHT 1


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


int get_largest_mobile(int *arr, int *dir, int n) {
    int largest_mobile_index = -1;
    int largest_mobile = 0;

    for (int i = 0; i < n; i++) {
        int neighbor_index = i + dir[i];
        if (neighbor_index >= 0 && neighbor_index < n) {
            if (arr[i] > arr[neighbor_index] && arr[i] > largest_mobile) {
                largest_mobile = arr[i];
                largest_mobile_index = i;
            }
        }
    }
    return largest_mobile_index;
}

void johnson_trotter(int n) {
    int *arr = malloc(n * sizeof(int));
    int *dir = malloc(n * sizeof(int));


    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
        dir[i] = LEFT;
```

```
    }

    print_permutation(arr, n);

    while (1) {
        int largest_mobile_index = get_largest_mobile(arr, dir, n);
        if (largest_mobile_index == -1) break;

        int swap_index = largest_mobile_index + dir[largest_mobile_index];

        int temp = arr[largest_mobile_index];
        arr[largest_mobile_index] = arr[swap_index];
        arr[swap_index] = temp;

        int temp_dir = dir[largest_mobile_index];
        dir[largest_mobile_index] = dir[swap_index];
        dir[swap_index] = temp_dir;

        largest_mobile_index = swap_index;


        for (int i = 0; i < n; i++) {
            if (arr[i] > arr[largest_mobile_index]) {
                dir[i] = -dir[i];
            }
        }

        print_permutation(arr, n);
    }

    free(arr);
    free(dir);
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("All permutations generated by Johnson-Trotter algorithm:\n");
    johnson_trotter(n);
    return 0;
}
```

1.10.3 Output:

```
Enter number of elements: 3
All permutations generated by Johnson-Trotter algorithm:
1 2 3
1 3 2
3 1 2
3 2 1
2 3 1
2 1 3
```

1.11 Experiment - 11

1.11.1 Question:

Sort a given set of N integer elements using Heap Sort technique and compute its time taken..

1.11.2 Code:

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

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        heapify(arr, n, largest);
    }
}


void heapSort(int arr[], int n) {

    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);


    for (int i = n - 1; i >= 0; i--) {

        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        heapify(arr, i, 0);
```

```
    }
}

int main() {
    int n;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    int *arr = (int*)malloc(n * sizeof(int));
    if (!arr) {
        printf("Memory allocation failed!\n");
        return 1;
    }

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

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    heapSort(arr, n);
    end = clock();

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

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

    printf("Time taken for heap sort: %f seconds\n", cpu_time_used);

    free(arr);
    return 0;
}
```

1.11.3 Output:

```
Enter number of elements: 7
Enter 7 elements:
12 4 56 3 78 1 9
Sorted array:
1 3 4 9 12 56 78
Time taken for heap sort: 0.000000 seconds
```