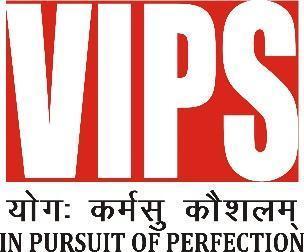
****

**VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS**

**Grade A++ Accredited Institution by NAAC**

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;

Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE

An ISO 9001:2015 Certified Institution

**SCHOOL OF ENGINEERING & TECHNOLOGY**

**BTECH Programme: AIDS-A**

**Course Title: Design and Analysis of Algorithms Lab**

**Course Code: AIDS353**

**Submitted To Submitted By**

**Dr. Sandhya Tarwani Name: Prachi Gupta**

**Enrollment No:** 02217711922

**VISION OF INSTITUTE**

To be an educational institute that empowers the field of engineering to build a sustainable future by providing quality education with innovative practices that supports people, planet and profit.

**MISSION OF INSTITUTE**

To groom the future engineers by providing value-based education and awakening students' curiosity, nurturing creativity and building  
capabilities to enable them to make significant contributions to the world.

**INDEX**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S.No** | **Experiment Name** | **Date** | **Marks** | | | **Remark** | **Updated Marks** | **Faculty Signature** |
| **Laboratory Assessment (15 Marks)** | **Class Participation (5 Marks)** | **Viva (5 Marks)** |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

**Experiment-1**

**Experiment Name:** Sort given set of elements using insertion sort and find time complexity for different values of n

**Description / Algorithm:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Source Code:**

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

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

int i;

for(i=1; i<n; i++) {

int key = arr[i];

int j = i-1;

while(j>=0 && arr[j]>key) {

arr[j+1] = arr[j];

j=j-1;

}

arr[j+1] = key;

}

}

void printArr(int arr[], int num) {

int i;

for(i=0; i<num; i++) {

printf("%d ", arr[i]);

}

}

int main() {

srand ( time(NULL) );

printf("Insertion Sort\n");

clock\_t time\_req;

int i,val;

int num = 50;

int arr[num];

for(i=0; i<num; i++){

val = rand();

arr[i] = val;

}

printf("Array Generated : \n");

printArr(arr,num);

time\_req = clock();

insertionSort(arr, num);

time\_req = clock() - time\_req;

printf("\n\nSorted Array : \n");

printArr(arr,num);

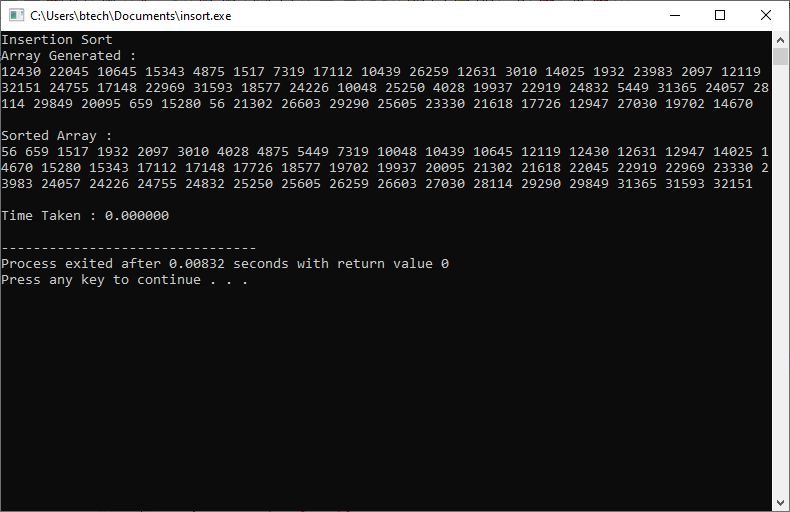
double finalTime = (double)time\_req/CLOCKS\_PER\_SEC;

printf("\n\nTime Taken : %f\n", finalTime);

return 0;

}

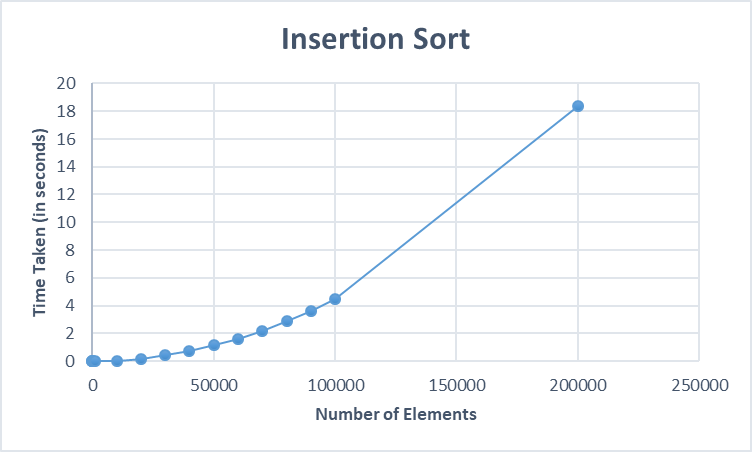
**Output :**

****

**Input (n) Vs Time Taken (t) Table:**

|  |  |
| --- | --- |
| **n** | **t** |
| 0 | 0 |
| 1 | 0 |
| 10 | 0 |
| 100 | 0 |
| 1000 | 0.001000 |
| 10000 | 0.031000 |
| 20000 | 0.171000 |
| 30000 | 0.406000 |
| 40000 | 0.721000 |
| 50000 | 1.124000 |
| 60000 | 1.611000 |
| 70000 | 2.202000 |
| 80000 | 2.874000 |
| 90000 | 3.640000 |
| 100000 | 4.499000 |
| 200000 | 18.395000 |

**Graph:**

****

**Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** |
| **Insertion Sort** |  |  |  |

**Learning Outcome:**

|  |
| --- |
|  |
|  |
|  |
|  |

**Experiment-2**

**Experiment Name:** Sort given set of elements using merge sort and find time complexity for different values of n

**Description / Algorithm:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Description**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

mergeSort(int arr[], int p, int r) {

    if (p < r) {

        int q = (p + r) / 2;

        mergeSort(arr, p, q);

        mergeSort(arr, q + 1, r);

        merge(arr, p, q, r);

    }

}

merge(int arr[], int p, int q, int r) {

    int n1 = q - p + 1;

    int n2 = r - q;

    int L[n1], R[n2];

    int i, j, k;

    for (i = 1; i <= n1; i++) {

        L[i] = arr[p + i - 1];

    }

    for (j = 1; j <= n2; j++) {

        R[j] = arr[q + j];

    }

    L[n1 + 1] = (int)INFINITY;

    R[n2 + 1] = (int)INFINITY;

    i = 1, j = 1;

    for (k = p; k <= r; k++) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

    }

}

void printArray(int A[], int size) {

    int i;

    for (i = 0; i < size; i++)

        printf("%d ", A[i]);

    printf("\n");

}

int main() {

    int arr\_size = 20;

    int arr[arr\_size];

    srand(time(NULL));

    int temp;

    for (temp = 0; temp < arr\_size; temp++) {

        arr[temp] = rand();

    }

    printf("Given array is \n");

    printArray(arr, arr\_size);

    clock\_t begin = clock();

    mergeSort(arr, 0, arr\_size - 1);

    clock\_t end = clock();

    printf("\nSorted array is \n");

    printArray(arr, arr\_size);

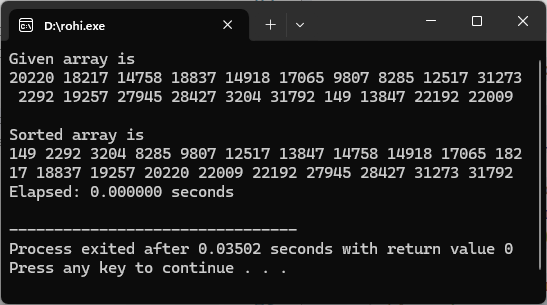
    double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

    printf("Elapsed: %f seconds\n", time\_spent);

    return 0;

}

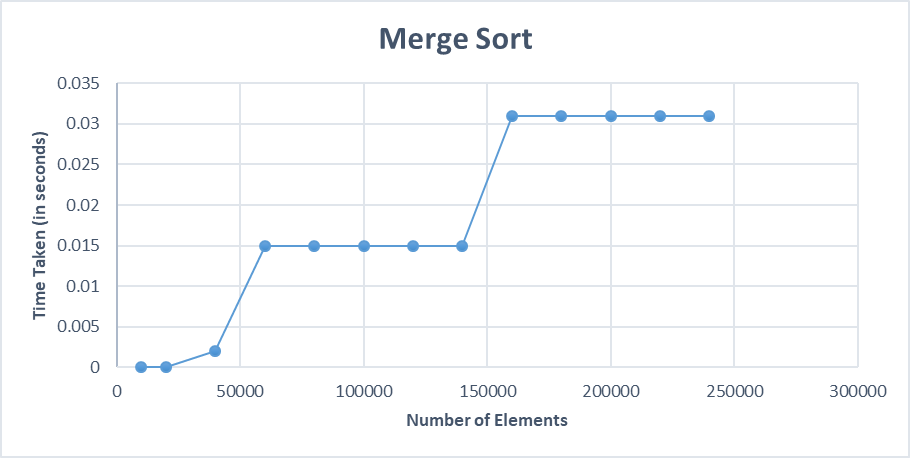
**Output :**

****

**Input (n) Vs Time Taken (t) Table:**

|  |  |
| --- | --- |
| **n** | **t** |
| 10000 | 0 |
| 20000 | 0 |
| 40000 | 0.002 |
| 60000 | 0.015 |
| 80000 | 0.015 |
| 100000 | 0.015 |
| 120000 | 0.015 |
| 140000 | 0.015 |
| 160000 | 0.031 |
| 180000 | 0.031 |
| 200000 | 0.031 |
| 220000 | 0.031 |
| 240000 | 0.031 |

**Graph:**

****

**Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** |
| **Merge Sort** |  |  |  |

**Learning Outcome:**

|  |
| --- |
|  |
|  |
|  |
|  |

**Experiment-3**

**Experiment Name:** Sort given set of elements using quick sort and find time complexity for different values of n

**Description / Algorithm:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Source Code:**

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include <math.h>

int partition(int arr[], int p, int r) {

int x = arr[r];

int j, i = p - 1;

for(j=p; j<=r-1; j++) {

if(arr[j] <= x) {

i+=1;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i+1];

arr[i+1] = arr[r];

arr[r] = temp;

return i+1;

}

void quickSort(int arr[], int p, int r) {

if(p < r) {

int q = partition(arr, p, r);

quickSort(arr, p, q-1);

quickSort(arr, q+1, r);

}

}

void printArray(int A[], int size) {

int i;

for (i = 0; i < size; i++)

printf("%d ", A[i]);

printf("\n");

}

int main() {

int i, size = 20;

int arr[size];

srand(time(NULL));

int temp;

for (temp = 0; temp < size; temp++) {

arr[temp] = rand();

}

printf("Before Swapping : \n");

printArray(arr, size);

clock\_t begin = clock();

quickSort(arr, 0, size-1);

clock\_t end = clock();

printf("\nAfter Swapping : \n");

printArray(arr, size);

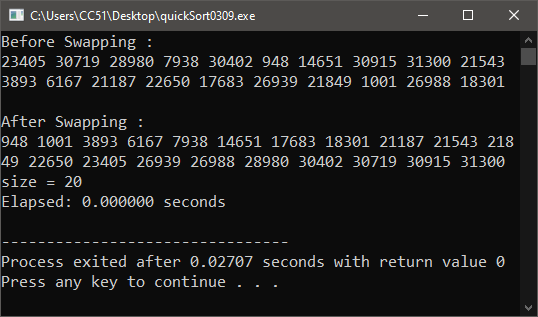
double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

printf("size = %d \nElapsed: %f seconds\n", size, time\_spent);

return 0;

}

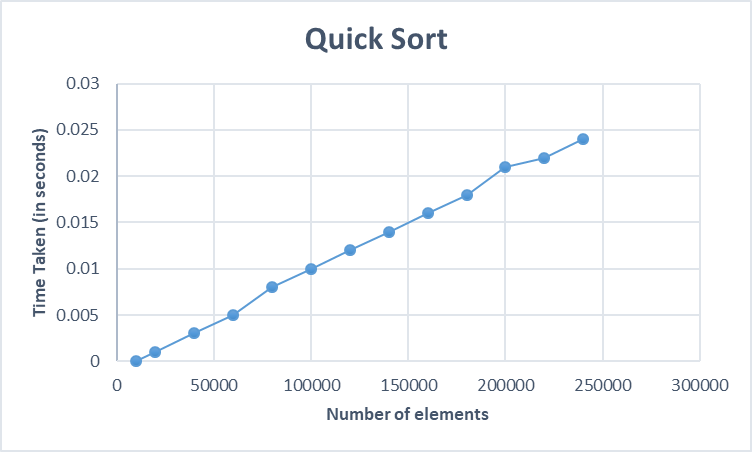
**Output :**

****

**Input (n) Vs Time Taken (t) Table:**

|  |  |
| --- | --- |
| **n** | **t** |
| 10000 | 0 |
| 20000 | 0.001 |
| 40000 | 0.003 |
| 60000 | 0.005 |
| 80000 | 0.008 |
| 100000 | 0.01 |
| 120000 | 0.012 |
| 140000 | 0.014 |
| 160000 | 0.016 |
| 180000 | 0.018 |
| 200000 | 0.021 |
| 220000 | 0.022 |
| 240000 | 0.024 |

**Graph:**

****

**Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** |
| **Quick Sort** |  |  |  |

**Learning Outcome:**

|  |
| --- |
|  |
|  |
|  |
|  |

**Experiment-4**

**Experiment Name:** Write a program to implement knapsack problem using greedy method

**Algorithm:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Description:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

typedef struct {

float weight;

float profit;

float ratio;

} Item;

void merge(Item arr[], int p, int q, int r) {

int n1 = q - p + 1;

int n2 = r - q;

Item \*L = (Item \*)malloc((n1 + 1) \* sizeof(Item));

Item \*R = (Item \*)malloc((n2 + 1) \* sizeof(Item));

for (int i = 0; i < n1; i++) {

L[i] = arr[p + i];

}

for (int j = 0; j < n2; j++) {

R[j] = arr[q + 1 + j];

}

L[n1].ratio = INFINITY; // Using ratio to signal end

R[n2].ratio = INFINITY; // Using ratio to signal end

int i = 0, j = 0;

for (int k = p; k <= r; k++) {

if (L[i].ratio >= R[j].ratio) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

}

free(L);

free(R);

}

void mergeSort(Item arr[], int p, int r) {

if (p < r) {

int q = (p + r) / 2;

mergeSort(arr, p, q);

mergeSort(arr, q + 1, r);

merge(arr, p, q, r);

}

}

float fractionalKnapsack(int num, float capacity, float weights[], float profits[]) {

Item \*items = (Item \*)malloc(num \* sizeof(Item));

for (int i = 0; i < num; i++) {

items[i].weight = weights[i];

items[i].profit = profits[i];

items[i].ratio = profits[i] / weights[i];

}

mergeSort(items, 0, num - 1);

float totalProfit = 0;

for (int i = 0; i < num; i++) {

if (capacity >= items[i].weight) {

capacity -= items[i].weight;

totalProfit += items[i].profit;

} else {

totalProfit += items[i].profit \* (capacity / items[i].weight);

break;

}

}

free(items);

return totalProfit;

}

int main() {

int numOfItems = 20;

float weights[numOfItems];

float profits[numOfItems];

srand(time(NULL));

for (int i = 0; i < numOfItems; i++) {

weights[i] = (float)(rand());

profits[i] = (float)(rand());

}

float capacity = (float)(rand() /5 );

clock\_t begin = clock();

float maxProfit = fractionalKnapsack(numOfItems, capacity, weights, profits);

clock\_t end = clock();

printf("Maximum Profit: %.2f\n", maxProfit);

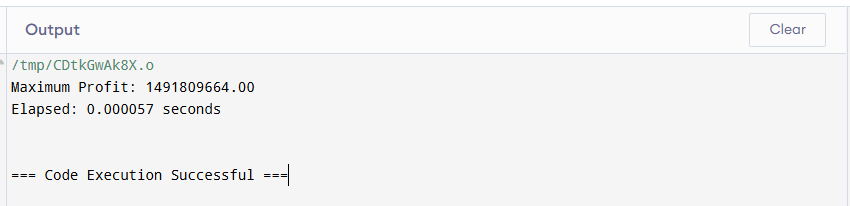
double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

printf("Elapsed: %f seconds\n", time\_spent);

return 0;

}

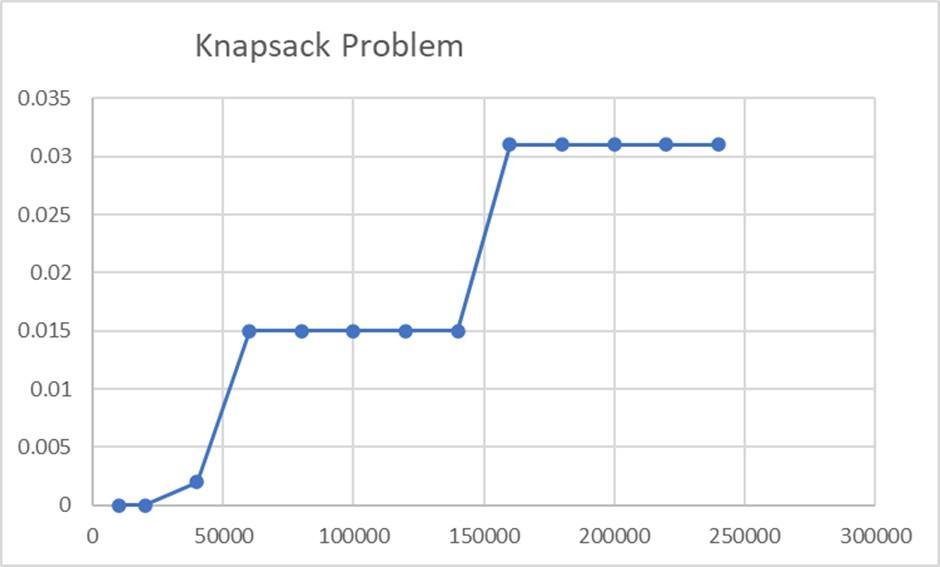
**Output :**



**Input (n) Vs Time Taken (t) Table:**

|  |  |  |
| --- | --- | --- |
| **n** | **Knapsack size** | **t** |
| 10000 | 2000 | 0 |
| 20000 | 400 | 0.001 |
| 40000 | 8000 | 0.003 |
| 60000 | 12000 | 0.005 |
| 80000 | 16000 | 0.008 |
| 100000 | 20000 | 0.01 |
| 120000 | 24000 | 0.012 |
| 140000 | 28000 | 0.014 |
| 160000 | 32000 | 0.016 |
| 180000 | 36000 | 0.018 |
| 200000 | 40000 | 0.021 |
| 220000 | 44000 | 0.022 |
| 240000 | 48000 | 0.024 |

**Graph:**



**Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** |
| **Knapsack Problem** |  |  |  |

**Learning Outcome:**

|  |
| --- |
|  |
|  |
|  |
|  |

**Experiment-5**

**Experiment Name:** Write a program to find minimum spanning tree using Prim's Algorithm

**Algorithm:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Description:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Source Code:**

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define V 10

int minKey(int key[], bool mstSet[]) {

    int min = INT\_MAX, min\_index;

    int v;

    for (v = 0; v < V; v++)

        if (mstSet[v] == false && key[v] < min) {

            min = key[v];

            min\_index = v;

        }

    return min\_index;

}

void printMST(int parent[], int graph[V][V]) {

    printf("\nEdge \tWeight\n");

    int i;

    for (i = 1; i < V; i++)

        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);

    int totW = 0;

    for (i = 1; i < V; i++)  // Start from 1 as 0 has no parent

        totW += graph[i][parent[i]];  // Use the edge weights

    printf("\nTotal Weight = %d \n", totW);

}

void primMST(int graph[V][V]) {

    int parent[V], key[V];

    bool mstSet[V];

    int i;

    for (i = 0; i < V; i++) {

        key[i] = INT\_MAX;

        mstSet[i] = false;

    }

    key[0] = 0;

    parent[0] = -1;

    int count;

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

        int u = minKey(key, mstSet);

        mstSet[u] = true;

        int v;

        for (v = 0; v < V; v++)

            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {

                parent[v] = u;

                key[v] = graph[u][v];

            }

    }

    printMST(parent, graph);

}

int main() {

    srand(time(NULL));

    int graph[V][V] = {0};

    int m, n;

    for (m = 0; m < V; m++) {

        for (n = m; n < V; n++) {

            if (m == n) {

                graph[m][n] = 0;

            } else {

                int weight = rand() % 100 + 1;

                graph[m][n] = weight;

                graph[n][m] = weight;

            }

        }

    }

    printf("Generated Random Graph (Adjacency Matrix):\n");

    int i, j;

    for (i = 0; i < V; i++) {

        for (j = 0; j < V; j++) {

            printf("%d\t", graph[i][j]);

        }

        printf("\n");

    }

    clock\_t begin = clock();

    primMST(graph);

    clock\_t end = clock();

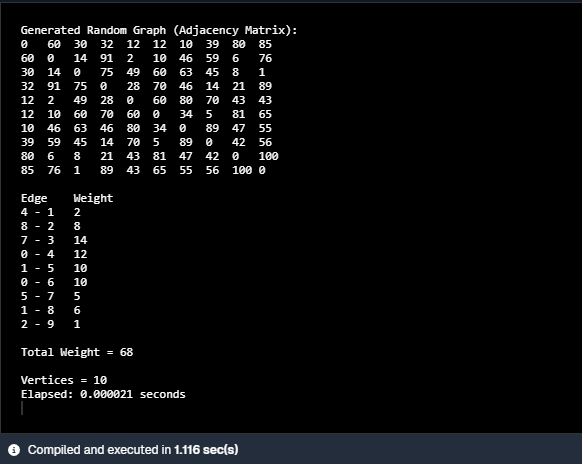
    double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

    printf("\nVertices = %d \nElapsed: %f seconds\n", V, time\_spent);

    return 0;

}

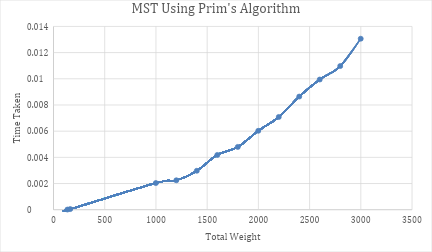
**Output :**

****

**Input (n) Vs Time Taken (t) Table:**

|  |  |  |
| --- | --- | --- |
| **Number of Vertices** | **Total Weight** | **Time taken** |
| 10 | 136 | 0.000013 |
| 100 | 163 | 0.000046 |
| 1000 | 999 | 0.002036 |
| 1200 | 1199 | 0.002248 |
| 1400 | 1399 | 0.002975 |
| 1600 | 1599 | 0.004185 |
| 1800 | 1799 | 0.004799 |
| 2000 | 1999 | 0.006026 |
| 2200 | 2199 | 0.007085 |
| 2400 | 2399 | 0.008641 |
| 2600 | 2599 | 0.009944 |
| 2800 | 2799 | 0.010971 |
| 3000 | 2999 | 0.013062 |

**Graph:**

****

**Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** |
| **Prim’s Algorithm** |  |  |  |

**Learning Outcome:**

|  |
| --- |
|  |
|  |
|  |
|  |

**Experiment-6**

**Experiment Name:** Write a program to find minimum spanning tree using Kruskal's Algorithm

**Algorithm:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Description:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define V 5

struct Edge {

    int src, dest, weight;

};

struct Graph {

    int vertices, edges;

    struct Edge\* edge;

};

struct subset {

    int parent;

    int rank;

};

struct Graph\* createGraph(int vertices, int edges) {

    struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

    graph->vertices = vertices;

    graph->edges = edges;

    graph->edge = (struct Edge\*)malloc(graph->edges \* sizeof(struct Edge));

    return graph;

}

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 compare(const void\* a, const void\* b) {

    struct Edge\* edgeA = (struct Edge\*)a;

    struct Edge\* edgeB = (struct Edge\*)b;

    return edgeA->weight > edgeB->weight;

}

void printMST(struct Edge result[], int e) {

    printf("\nEdge \tWeight\n");

    int totalWeight = 0;

    for (int i = 0; i < e; ++i) {

        printf("%d - %d \t%d\n", result[i].src, result[i].dest, result[i].weight);

        totalWeight += result[i].weight;

    }

    printf("\nTotal Weight = %d\n", totalWeight);

}

void KruskalMST(struct Graph\* graph) {

    int vertices = graph->vertices;

    struct Edge result[vertices];

    int e = 0;

    int i = 0;

    qsort(graph->edge, graph->edges, sizeof(graph->edge[0]), compare);

    struct subset\* subsets = (struct subset\*)malloc(vertices \* sizeof(struct subset));

    for (int v = 0; v < vertices; ++v) {

        subsets[v].parent = v;

        subsets[v].rank = 0;

    }

    while (e < vertices - 1 && i < graph->edges) {

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

        int x = find(subsets, next\_edge.src);

        int y = find(subsets, next\_edge.dest);

        if (x != y) {

            result[e++] = next\_edge;

            Union(subsets, x, y);

        }

    }

    printMST(result, e);

    free(subsets);

}

int main() {

    srand(time(NULL));

    int edges = V \* (V - 1) / 2;

    struct Graph\* graph = createGraph(V, edges);

    int edge\_count = 0;

    for (int m = 0; m < V; m++) {

        for (int n = m + 1; n < V; n++) {

            int weight = rand() % 100 + 1;

            graph->edge[edge\_count].src = m;

            graph->edge[edge\_count].dest = n;

            graph->edge[edge\_count].weight = weight;

            edge\_count++;

        }

    }

    printf("Generated Random Graph Edges:\n");

    for (int i = 0; i < edge\_count; i++) {

        printf("%d - %d \t%d\n", graph->edge[i].src, graph->edge[i].dest, graph->edge[i].weight);

    }

    clock\_t begin = clock();

    KruskalMST(graph);

    clock\_t end = clock();

    double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

    printf("\nVertices = %d \nElapsed: %f seconds\n", V, time\_spent);

    return 0;

}

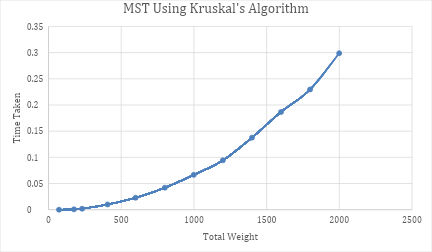
**Output :**

****

**Input (n) Vs Time Taken (t) Table:**

|  |  |  |
| --- | --- | --- |
| **Number of Vertices** | **Total Weight** | **Time taken** |
| 10 | 72 | 0.000018 |
| 100 | 175 | 0.000690 |
| 200 | 231 | 0.001942 |
| 400 | 406 | 0.010068 |
| 600 | 599 | 0.022869 |
| 800 | 800 | 0.042113 |
| 1000 | 999 | 0.066583 |
| 1200 | 1199 | 0.094346 |
| 1400 | 1399 | 0.137646 |
| 1600 | 1599 | 0.186651 |
| 1800 | 1799 | 0.229703 |
| 2000 | 1999 | 0.298829 |

**Graph:**

****

**Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** |
| **Kruskal’s Algorithm** |  |  |  |

**Learning Outcome:**

|  |
| --- |
|  |
|  |
|  |
|  |

**Experiment-7**

**Experiment Name:** Write a program to perform Single Source Shortest Path problem using Dijkstra's Algorithm.

**Algorithm:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Description:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Source Code:**

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define V 10

int minDistance(int dist[], bool sptSet[]) {

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < V; v++) {

        if (sptSet[v] == false && dist[v] <= min) {

            min = dist[v];

            min\_index = v;

        }

    }

    return min\_index;

}

void printSolution(int dist[]) {

    printf("\nVertex \t Distance from Source\n");

    for (int i = 0; i < V; i++) {

        printf("%d \t %d\n", i, dist[i]);

    }

}

void dijkstra(int \*\*graph, int src) {

    int dist[V];

    bool sptSet[V];

    for (int i = 0; i < V; i++) {

        dist[i] = INT\_MAX;

        sptSet[i] = false;

    }

    dist[src] = 0;

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

        int u = minDistance(dist, sptSet);

        sptSet[u] = true;

        for (int v = 0; v < V; 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];

            }

        }

    }

    printSolution(dist);

}

int main() {

    srand(time(NULL));

    int i, j, m, n;

    int \*\*graph = malloc(V \* sizeof(int \*));

    for (m = 0; m < V; m++) {

        graph[m] = malloc(V \* sizeof(int));

    }

    for (m = 0; m < V; m++) {

        for (n = 0; n < V; n++) {

            if (m == n) {

                graph[m][n] = 0;

            } else {

                graph[m][n] = rand() % 100 + 1;

            }

        }

    }

    printf("Generated Random Graph (Adjacency Matrix):\n");

    for (i = 0; i < V; i++) {

        for (j = 0; j < V; j++) {

            printf("%d\t", graph[i][j]);

        }

        printf("\n");

    }

    clock\_t begin = clock();

    dijkstra(graph, 0);

    clock\_t end = clock();

    double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

    printf("\nVertices = %d \nElapsed: %f seconds\n", V, time\_spent);

    for (m = 0; m < V; m++) {

        free(graph[m]);

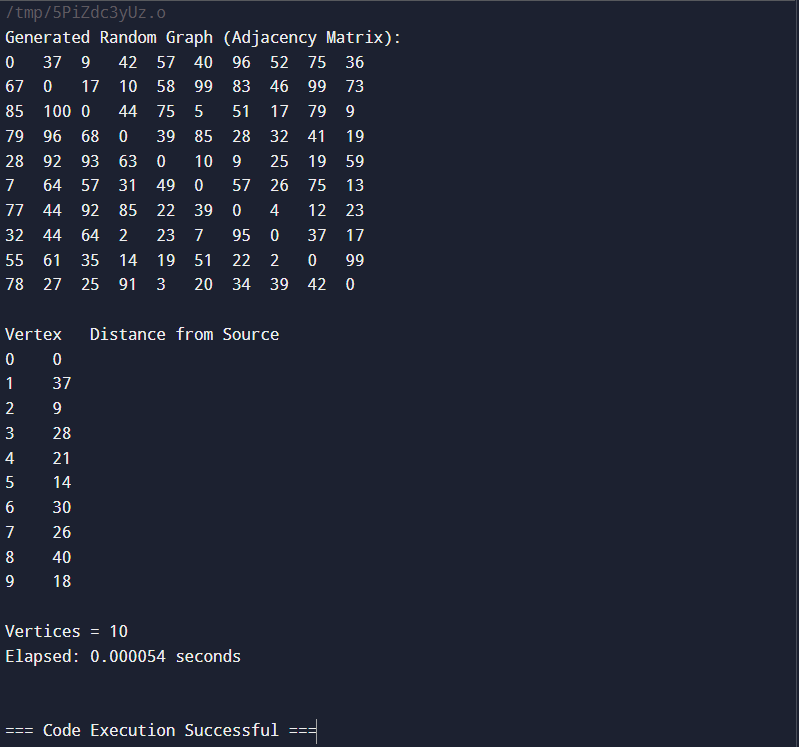
    }

    free(graph);

    return 0;

}

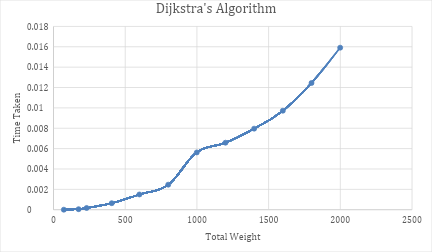
**Output :**

****

**Input (n) Vs Time Taken (t) Table:**

|  |  |
| --- | --- |
| **Number of Vertices** | **Time taken** |
| 10 | 0.000004 |
| 100 | 0.000057 |
| 200 | 0.000172 |
| 400 | 0.000642 |
| 600 | 0.001499 |
| 800 | 0.002461 |
| 1000 | 0.005628 |
| 1200 | 0.006582 |
| 1400 | 0.007963 |
| 1600 | 0.009723 |
| 1800 | 0.012445 |
| 2000 | 0.015913 |

**Graph:**

****

**Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** |
| **Dijkstra's Algorithm** |  |  |  |

**Learning Outcome:**

|  |
| --- |
|  |
|  |
|  |
|  |

**Experiment-8**

**Experiment Name:** Write a program to perform Single Source Shortest Path problem using Bellman Ford Algorithm.

**Algorithm:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Description:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <time.h>

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

#define E 10  // Number of edges in the graph

struct Edge {

    int src, dest, weight;

};

void printSolution(int dist[]) {

    printf("\nVertex \t Distance from Source\n");

    for (int i = 0; i < V; i++) {

        printf("%d \t %d\n", i, dist[i]);

    }

}

void bellmanFord(struct Edge\* edges, int edgeCount, int src) {

    int dist[V];

    for (int i = 0; i < V; i++) {

        dist[i] = INT\_MAX;

    }

    dist[src] = 0;

    for (int i = 1; i <= V - 1; i++) {

        for (int j = 0; j < edgeCount; j++) {

            int u = edges[j].src;

            int v = edges[j].dest;

            int weight = edges[j].weight;

            if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

                dist[v] = dist[u] + weight;

            }

        }

    }

    for (int i = 0; i < edgeCount; i++) {

        int u = edges[i].src;

        int v = edges[i].dest;

        int weight = edges[i].weight;

        if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

            printf("Graph contains negative weight cycle\n");

            return;

        }

    }

    printSolution(dist);

}

void ensureConnectivity(struct Edge\* edges, int\* edgeCount) {

    int connected[V] = {0};

    connected[0] = 1;

    int currentEdgeCount = 0;

    for (int i = 1; i < V; i++) {

        int src = rand() % i;   // Randomly connect to a vertex that's already connected

        int dest = i;

        int weight = rand() % 100;

        edges[currentEdgeCount].src = src;

        edges[currentEdgeCount].dest = dest;

        edges[currentEdgeCount].weight = weight;

        currentEdgeCount++;

    }

    \*edgeCount = currentEdgeCount;

}

int main() {

    srand(time(NULL));

    struct Edge\* edges = malloc(E \* sizeof(struct Edge));

    int edgeCount = 0;

    ensureConnectivity(edges, &edgeCount);

    for (int i = edgeCount; i < E; i++) {

        int src = rand() % V;

        int dest = rand() % V;

        int weight;

        // Assign random negative weights to a small percentage of edges

        if (rand() % 100 < 10) {  // 10% chance of a negative weight

            weight = -(rand() % 50 + 1);

        } else {

            weight = rand() % 100 + 1;

        }

        edges[i].src = src;

        edges[i].dest = dest;

        edges[i].weight = weight;

        edgeCount++;

    }

    printf("Generated Random Graph (Edge List with limited negative weights):\n");

    for (int i = 0; i < edgeCount; i++) {

        printf("Edge: %d -> %d, Weight: %d\n", edges[i].src, edges[i].dest, edges[i].weight);

    }

    clock\_t begin = clock();

    bellmanFord(edges, edgeCount, 0);

    clock\_t end = clock();

    double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

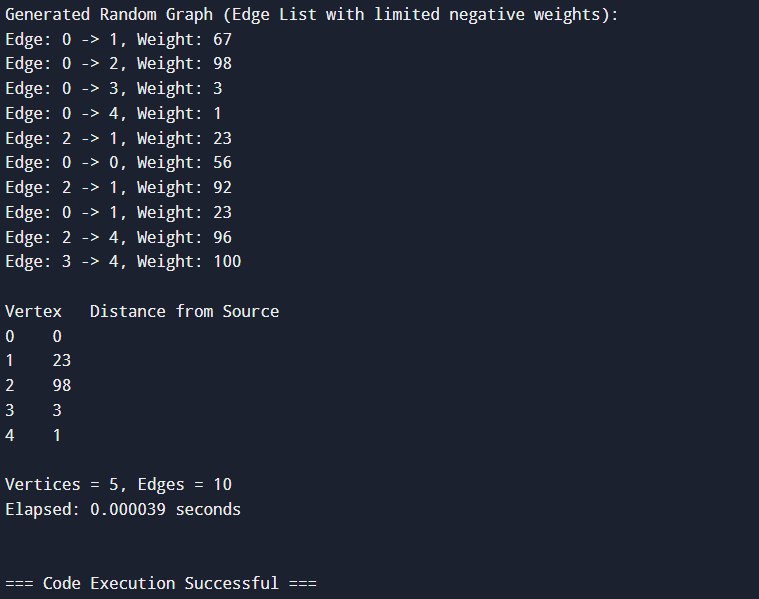
    printf("\nVertices = %d, Edges = %d \nElapsed: %f seconds\n", V, edgeCount, time\_spent);

    free(edges);

    return 0;

}

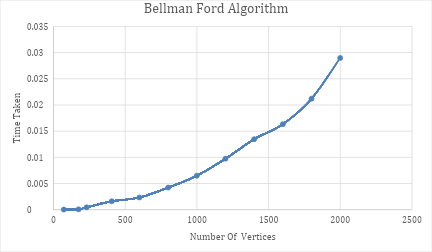
**Output :**

****

**Input (n) Vs Time Taken (t) Table:**

|  |  |  |
| --- | --- | --- |
| **Number of Vertices** | **Number Of Edges** | **Time taken** |
| 10 | 20 | 0.000032 |
| 100 | 200 | 0.000069 |
| 200 | 400 | 0.000450 |
| 400 | 800 | 0.001610 |
| 600 | 1200 | 0.002339 |
| 800 | 1600 | 0.004228 |
| 1000 | 2000 | 0.006500 |
| 1200 | 2400 | 0.009745 |
| 1400 | 2800 | 0.013473 |
| 1600 | 3200 | 0.016337 |
| 1800 | 3600 | 0.021196 |
| 2000 | 4000 | 0.028980 |

**Graph:**

****

**Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** |
| **Bellman Ford Algorithm** |  |  |  |

**Learning Outcome:**

|  |
| --- |
|  |
|  |
|  |
|  |

**Experiment-9**

**Experiment Name:** Write a program to perform All Pair Shortest Path Algorithm using Floyd Warshall Algorithm.

**Algorithm:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Description:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Source Code:**

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define V 10

void floydWarshall(int \*\*graph) {

    int \*\*dist = malloc(V \* sizeof(int \*));

    int i,j,k,m,n;

    for (i = 0; i < V; i++) {

        dist[i] = malloc(V \* sizeof(int));

    }

    for (i = 0; i < V; i++) {

        for (j = 0; j < V; j++) {

            dist[i][j] = graph[i][j];

        }

    }

    for (k = 0; k < V; k++) {

        for (i = 0; i < V; i++) {

            for (j = 0; j < V; j++) {

                if (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX &&

                    dist[i][j] > dist[i][k] + dist[k][j]) {

                    dist[i][j] = dist[i][k] + dist[k][j];

                }

            }

        }

    }

    printf("\nShortest distances between every pair of vertices:\n");

    for (i = 0; i < V; i++) {

        for (j = 0; j < V; j++) {

            if (dist[i][j] == INT\_MAX) {

                printf("INF\t");

            } else {

                printf("%d\t", dist[i][j]);

            }

        }

        printf("\n");

    }

    for (i = 0; i < V; i++) {

        free(dist[i]);

    }

    free(dist);

}

int main() {

    srand(time(NULL));

int i,j,m,n;

    int \*\*graph = malloc(V \* sizeof(int \*));

    for (m = 0; m < V; m++) {

        graph[m] = malloc(V \* sizeof(int));

    }

    for (m = 0; m < V; m++) {

        for (n = 0; n < V; n++) {

            if (m == n) {

                graph[m][n] = 0;

            } else {

                graph[m][n] = rand() % 100 + 1;

            }

        }

    }

    printf("Generated Random Graph (Adjacency Matrix):\n");

    for (i = 0; i < V; i++) {

        for (j = 0; j < V; j++) {

            printf("%d\t", graph[i][j]);

        }

        printf("\n");

    }

    clock\_t begin = clock();

    floydWarshall(graph);

    clock\_t end = clock();

    double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

    printf("\nVertices = %d \nElapsed: %f seconds\n", V, time\_spent);

    for (m = 0; m < V; m++) {

        free(graph[m]);

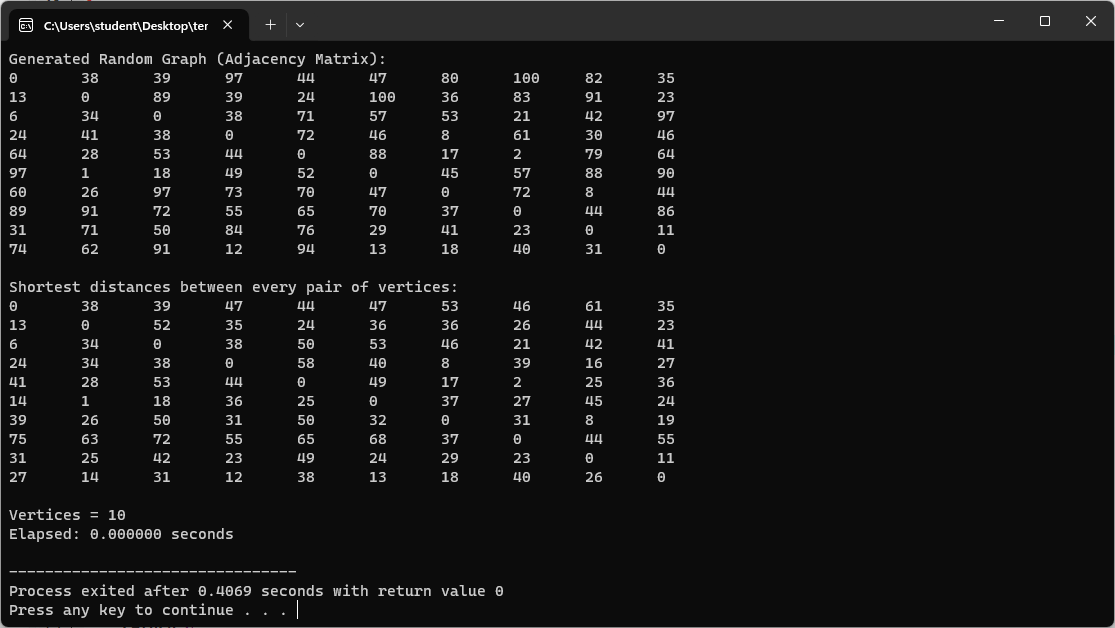
    }

    free(graph);

    return 0;

}

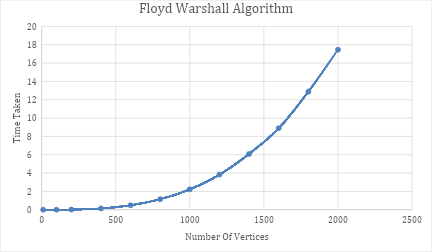
**Output :**

****

**Input (n) Vs Time Taken (t) Table:**

|  |  |
| --- | --- |
| **Number of Vertices** | **Time taken** |
| 10 | 0 |
| 100 | 0.004 |
| 200 | 0.015 |
| 400 | 0.139 |
| 600 | 0.487 |
| 800 | 1.151 |
| 1000 | 2.226 |
| 1200 | 3.832 |
| 1400 | 6.088 |
| 1600 | 8.9 |
| 1800 | 12.881 |
| 2000 | 17.463 |

**Graph:**

****

**Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** |
| **Floyd Warshall Algorithm** |  |  |  |

**Learning Outcome:**

|  |
| --- |
|  |
|  |
|  |
|  |

**Experiment-10**

**Experiment Name:** Write a program to implement N-Queens's Problem using Backtracking Approach.

**Algorithm:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Description:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Source Code:**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#include <time.h>

#define N 8

void printSolution(int \*\*board) {

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            printf("%s\t", board[i][j] ? "Q" : ".");

        }

        printf("\n");

    }

}

bool isSafe(int \*\*board, int row, int col) {

    int i, j;

    for (i = 0; i < col; i++)

        if (board[row][i])

            return false;

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

        if (board[i][j])

            return false;

    for (i = row, j = col; j >= 0 && i < N; i++, j--)

        if (board[i][j])

            return false;

    return true;

}

bool solveNQUtil(int \*\*board, int col) {

    if (col >= N)

        return true;

    for (int i = 0; i < N; i++) {

        if (isSafe(board, i, col)) {

            board[i][col] = 1;

            if (solveNQUtil(board, col + 1))

                return true;

            board[i][col] = 0;

        }

    }

    return false;

}

bool solveNQ() {

    int \*\*board = malloc(N \* sizeof(int \*));

    for (int i = 0; i < N; i++) {

        board[i] = malloc(N \* sizeof(int));

    }

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            board[i][j] = 0;

        }

    }

    if (!solveNQUtil(board, 0)) {

        printf("Solution does not exist.\n");

        return false;

    }

    printf("Solution to the %d-Queens Problem:\n", N);

    printSolution(board);

    for (int i = 0; i < N; i++) {

        free(board[i]);

    }

    free(board);

    return true;

}

int main() {

    clock\_t begin = clock();

    solveNQ();

    clock\_t end = clock();

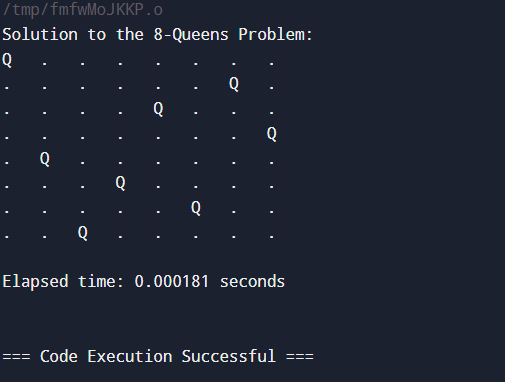
    double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

    printf("\nElapsed time: %f seconds\n", time\_spent);

    return 0;

}

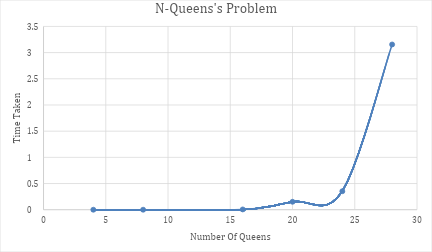
**Output :**

****

**Input (n) Vs Time Taken (t) Table:**

|  |  |
| --- | --- |
| **Number of Queens (N)** | **Time taken** |
| 4 | 0.000111 |
| 8 | 0.000181 |
| 16 | 0.006005 |
| 20 | 0.151236 |
| 24 | 0.353436 |
| 28 | 3.155545 |

**Graph:**

****

**Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** |
| **N-Queens's Problem** |  |  |  |

**Learning Outcome:**

|  |
| --- |
|  |
|  |
|  |
|  |