**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**on**

**Analysis and Design of Algorithms**

***Submitted by***

**ANSHA PRASHANTH (1BM20CS018)**

***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**

**May-2022 to July-2022**

**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 **ANSHA PRASHANTH (1BM20CS018),** 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 2022. The Lab report has been approved as it satisfies the academic requirements in respect of a **Analysis and Design of Algorithms - (19CS4PCADA)** work prescribed for the said degree.

**Dr. Nagarathna N**                **Dr. Jyothi S Nayak**

Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

`

**Index Sheet**

| **Sl. No.** | **Experiment Title** | **Page No.** |
| --- | --- | --- |
| **1** | Write a recursive program to Solve  **a)** Towers-of-Hanoi problem **b)** To find GCD | **5-7** |
| **2** | Implement Recursive Binary search and Linear search and determine the time required to search an element. Repeat the experiment for different values of N and plot a graph of the time taken versus N. | **8-13** |
| **3** | Sort a given set of N integer elements using Selection Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort. | **14-17** |
| **4** | Write program to do the following:  **a)** Print all the nodes reachable from a given starting node in a digraph using BFS method.  **b)** Check whether a given graph is connected or not using DFS method. | **18-23** |
| **5** | Sort a given set of N integer elements using Insertion Sort technique and compute its time taken. | **24-26** |
| **6** | Write program to obtain the Topological ordering of vertices in a given digraph. | **27-30** |
| **7** | Implement Johnson Trotter algorithm to generate permutations. | **31-36** |
| **8** | 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. | **37-40** |
| **9** | Sort a given set of N integer elements using Quick Sort technique and compute its time taken. | **41-44** |
| **10** | Sort a given set of N integer elements using Heap Sort technique and compute its time taken. | **45-48** |
| **11** | Implement Warshall’s algorithm using dynamic programming | **49-51** |
| **12** | Implement 0/1 Knapsack problem using dynamic programming. | **52-56** |
| **13** | Implement All Pair Shortest paths problem using Floyd’s algorithm. | **57-59** |
| **14** | Find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm. | **60-62** |
| **15** | Find Minimum Cost Spanning Tree of a given undirected graph using Kruskals algorithm. | **63-66** |
| **16** | From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra’s algorithm. | **67-70** |
| **17** | 1. Implement “Sum of Subsets” using Backtracking. “Sum of Subsets” problem: Find a subset of a given set S = {s1,s2,……,sn} of n positive integers whose sum is equal to a given positive integer d. For example, if S = {1,2,5,6,8} and d = 9 there are two solutions {1,2,6} and {1,8}. A suitable message is to be displayed if the given problem instance doesn’t have a solution. |  |
| **18** | Implement “N-Queens Problem” using Backtracking. |  |

**Course Outcome**

| **CO1** | Ability to **analyze** time complexity of Recursive and Non-Recursive algorithms using asymptotic notations. |
| --- | --- |
| **CO2** | Ability to **design** efficient algorithms using various design techniques. |
| **CO3** | Ability to **apply** the knowledge of complexity classes P, NP, and NP-Complete and prove certain problems are NP-Complete |
| **CO4** | Ability to **conduct** practical experiments to solve problems using an appropriate designing method and find time efficiency. |

**PROGRAM 1**

**TOWER OF HANOI AND GCD OF TWO NUMBERS**

| Write a recursive program to Solve  **a)** Towers-of-Hanoi problem **b)** To find GCD |
| --- |

1. **Program for Tower of Hanoi:**

#include <stdio.h>

void toh(int n, char s, char d, char i)

{

if (n == 1)

{

printf("move %c -> %c\n", s, d);

}

else

{

toh(n - 1, s, i, d);

toh(1, s, d, i);

toh(n - 1, i, d, s);

}

// s - source, d - destination, i - intermediate

}

int main(void)

{

int n;

printf("Enter number of disks: ");

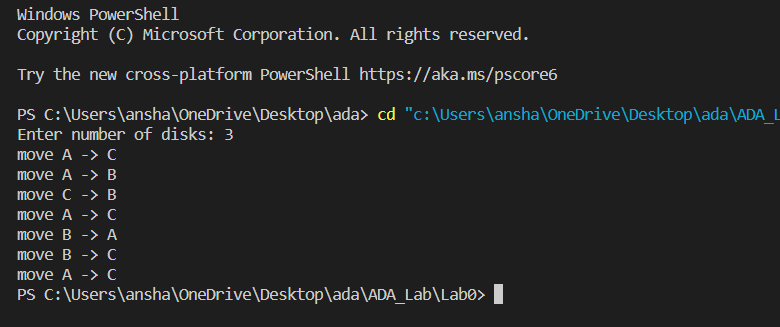
scanf("%d", &n);

toh(n, 'A', 'C', 'B');

return 0;

}

**Output:**



1. **Program for GCD:**

#include <stdio.h>

int gcd(int m, int n)

{

if (n == 0)

return m;

else

return gcd(n, m % n);

}

int main(void)

{

int m, n, hcf;

printf("Enter two numbers: ");

scanf("%d %d", &m, &n);

if (n > m)

{

int temp = m;

m = n;

n = temp;

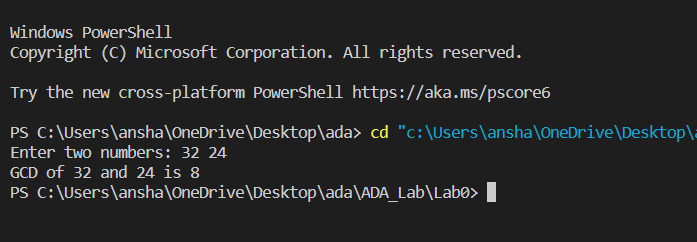
}

printf("GCD of %d and %d is %d\n", m, n, gcd(m, n));

return 0;

}

**Output:**

****

**PROGRAM 2**

**RECURSIVE BINARY SEARCH AND LINEAR SEARCH**

| Implement Recursive Binary search and Linear search and determine the time required to search an element. Repeat the experiment for different values of N and plot a graph of the time taken versus N. |
| --- |

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 100000

int linearSearch(int arr[], int n, int index, int key)

{

if (index > n)

return -1;

else if (arr[index] == key)

return index;

else

return linearSearch(arr, n, index + 1, key);

}

int binarySearch(int arr[], int l, int r, int key)

{

if (r >= l)

{

int mid = l + (r - l) / 2;

if (arr[mid] == key)

return mid;

if (arr[mid] > key)

return binarySearch(arr, l, mid - 1, key);

return binarySearch(arr, mid + 1, r, key);

}

return -1;

}

int main(void)

{

int arr[SIZE], key, choice;

clock\_t start, end;

int index = 0, n;

double time\_taken;

while (choice != 3)

{

printf("\nChoose\n"

"1. Linear Search\n"

"2. Binary search\n"

"3. Exit\n");

scanf("%d", &choice);

if (choice == 3)

break;

printf("Enter value for n:");

scanf("%d", &n);

for (int i = 0; i < n; i++)

{

arr[i] = i;

}

key = arr[n - 1];

start = clock();

switch (choice)

{

case 1:

for (int i = 0; i < 1000; i++)

{

index = linearSearch(arr, n, 0, key);

}

printf("Linear search executed for 1000 iterations");

break;

case 2:

for (int i = 0; i < 100000; i++)

{

index = binarySearch(arr, 0, n - 1, key);

}

printf("Binary search executed for 100000 iterations");

break;

default:

printf("Invalid choice\n");

break;

}

end = clock();

time\_taken = (end - start) / (double)CLOCKS\_PER\_SEC \* 1000;

if (index == -1)

printf("\nElement not found!\n");

else

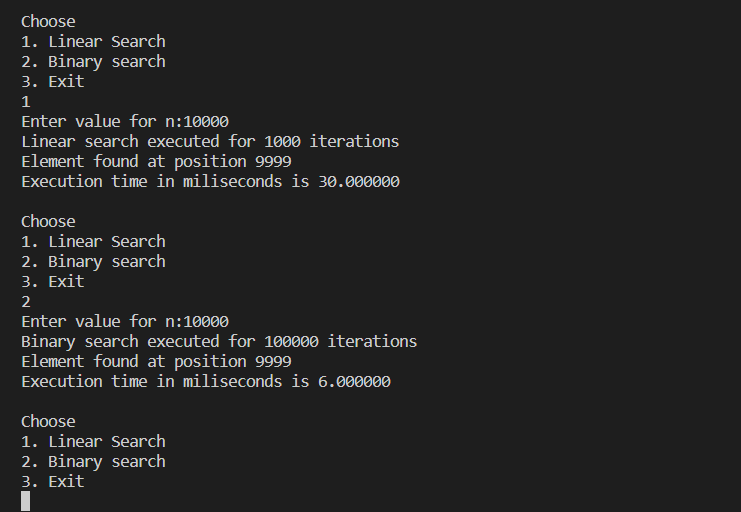
printf("\nElement found at position %d\n", index);

printf("Execution time in milliseconds is %lf\n", time\_taken);

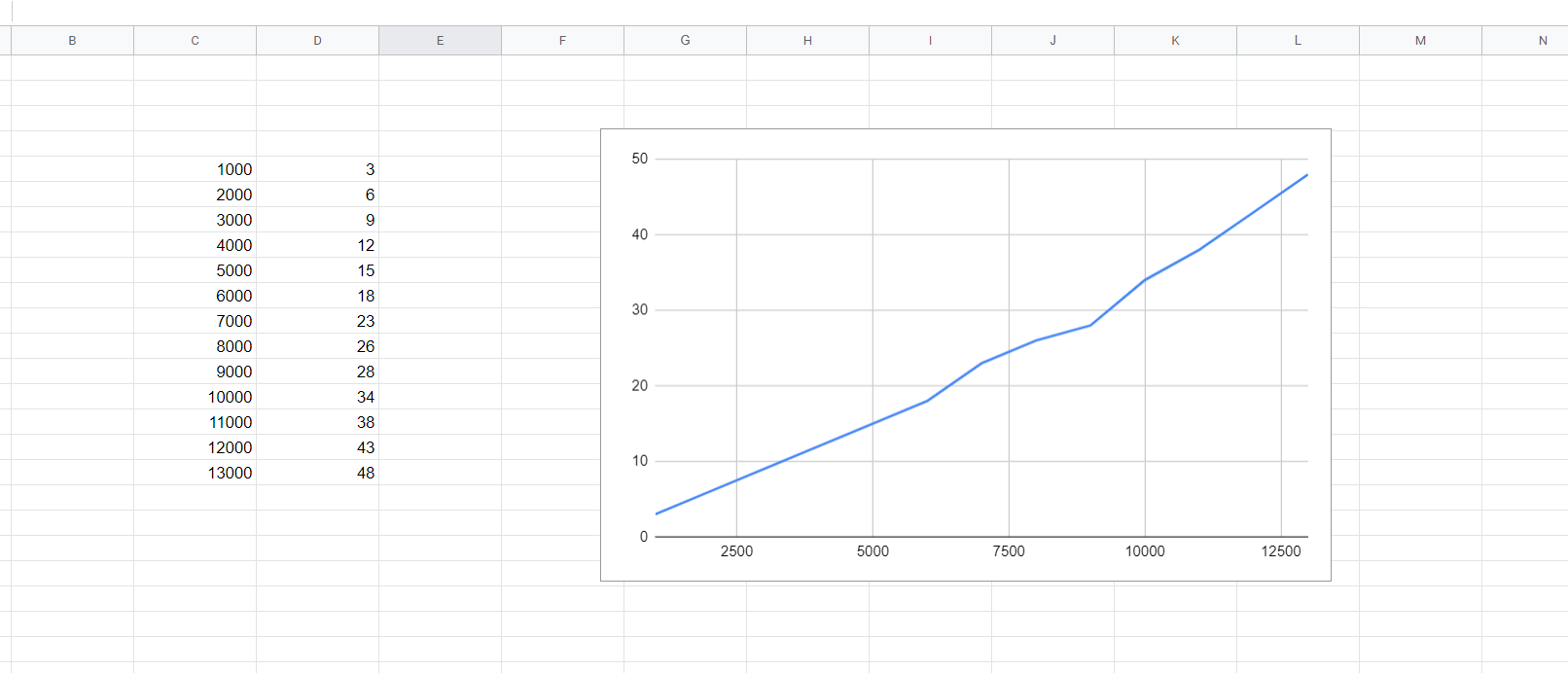
}

}

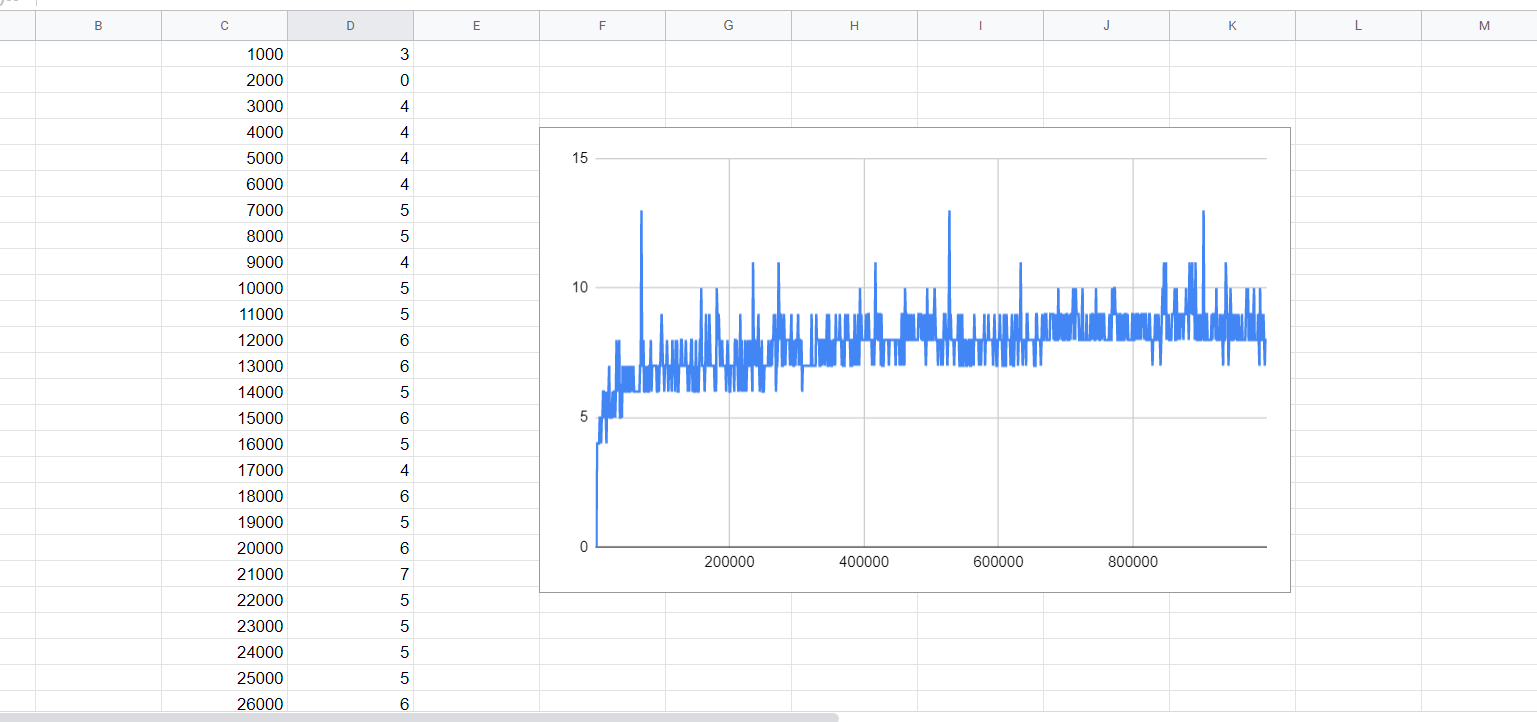
**Output:**

****

**Graph for linear search:**



**Graph for Binary Search:**



**PROGRAM 3**

**SELECTION SORT**

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

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

clock\_t start, end;

double time\_taken;

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void selection\_sort(int arr[], int n)

{

int i, j, min, temp;

for (i = 0; i < n - 1; i++)

{

min = i;

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

{

if (arr[j] < arr[min])

{

min = j;

}

}

swap(&arr[min], &arr[i]);

}

}

int main(void)

{

int n;

printf("Enter the number of elements: ");

scanf("%d", &n);

int arr[n];

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

{

arr[n - i] = i;

}

start = clock();

selection\_sort(arr, n);

end = clock();

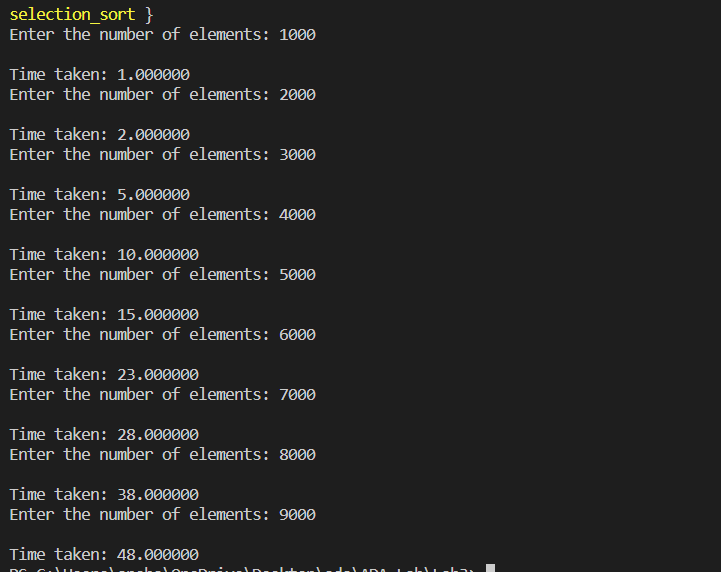
time\_taken = (end - start) / (double)CLOCKS\_PER\_SEC \* 1000;

printf("\nTime taken: %f\n", time\_taken);

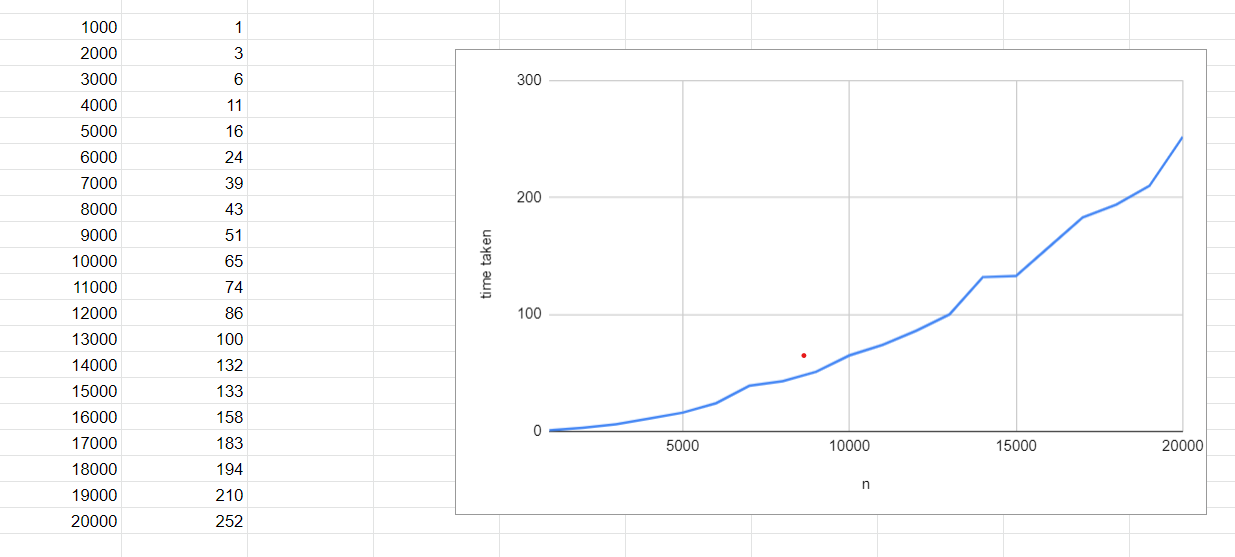
return 0;

}

**Output:**

****

**Graph for selection sort:**

****

**PROGRAM 4**

**BFS AND DFS**

| Write program to do the following:  **a)** Print all the nodes reachable from a given starting node in a digraph using BFS method.  **b)** Check whether a given graph is connected or not using DFS method. |
| --- |

**a)Program (BFS):**

#include <stdio.h>

#define size 20

#define true 1

#define false 0

int queue[size], visit[20], rear = -1, front = 0;

int n, s, adj[20][20], flag = 0;

void insertq(int v){

queue[++rear] = v;

}

int deleteq(){

return (queue[front++]);

}

int qempty(){

if (rear < front)

return 1;

else

return 0;

}

void bfs(int v)

{

int w;

visit[v] = 1;

insertq(v);

while (!qempty())

{

v = deleteq();

for (w = 1; w <= n; w++)

if ((adj[v][w] == 1) && (visit[w] == 0))

{

visit[w] = 1;

flag = 1;

printf("v%d\t", w);

insertq(w);

}

}

}

int main(void)

{

int v, w;

printf("Enter the no.of vertices:\n");

scanf("%d", &n);

printf("Enter adjacency matrix:");

for (v = 1; v <= n; v++)

{

for (w = 1; w <= n; w++)

scanf("%d", &adj[v][w]);

}

printf("Enter the start vertex:");

scanf("%d", &s);

printf("Reachability of vertex %d\n", s);

for (v = 1; v <= n; v++)

visit[v] = 0;

bfs(s);

if (flag == 0)

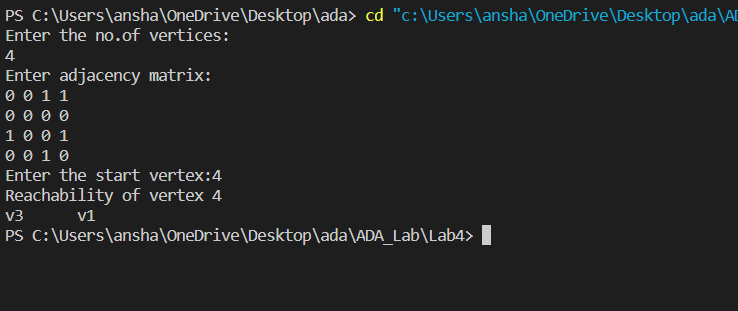
{

printf("No path found!!\n");

}

}

**Output for BFS:**

****

**b) Program (DFS):**

#include <stdio.h>

#include <stdlib.h>

int a[20][20], reach[20], n;

void dfs(int v)

{

int i;

reach[v] = 1;

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

if (a[v][i] && !reach[i])

{

printf("\n %d->%d", v, i);

dfs(i);

}

}

int main(void)

{

int i, j, count = 0;

printf("\nEnter number of vertices:");

scanf("%d", &n);

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

{

reach[i] = 0;

for (j = 1; j <= n; j++)

a[i][j] = 0;

}

printf("\nEnter the adjacency matrix:\n");

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

for (j = 1; j <= n; j++)

scanf("%d", &a[i][j]);

dfs(1);

printf("\n");

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

{

if (reach[i])

count++;

}

if (count == n)

printf("\n Graph is connected");

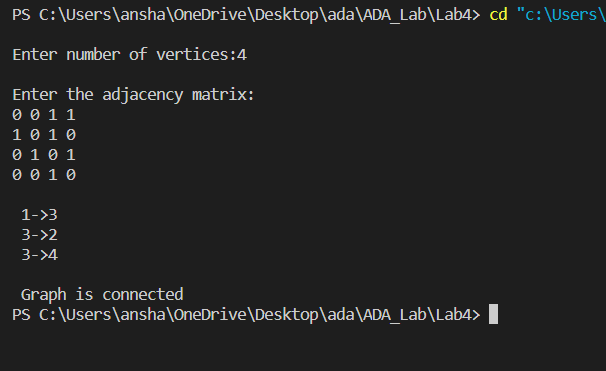
else

printf("\n Graph is not connected");

return 0;

}

**Output:**

****

**PROGRAM 5**

**INSERTION SORT**

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

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 10000

void insertionSort(int array[], int size)

{

for (int step = 1; step < size; step++)

{

int key = array[step];

int j = step - 1;

while (key < array[j] && j >= 0)

{

array[j + 1] = array[j];

--j;

}

array[j + 1] = key;

}

}

int main(void)

{

int n = 1000;

printf("\nTime taken each, for 10 observations:\n");

while (n < SIZE)

{

int a[n];

clock\_t start, end;

start = clock();

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

{

a[n - i] = i;

}

for (int i = 0; i < 1000; i++)

{

insertionSort(a, n);

}

end = clock();

printf("%f\n", (double)(end - start) / CLOCKS\_PER\_SEC \* 1000);

n = n + 1000;

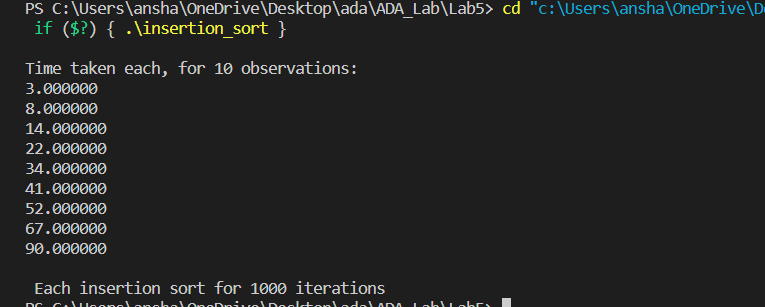
}

printf("\n Each insertion sort for 1000 iterations");

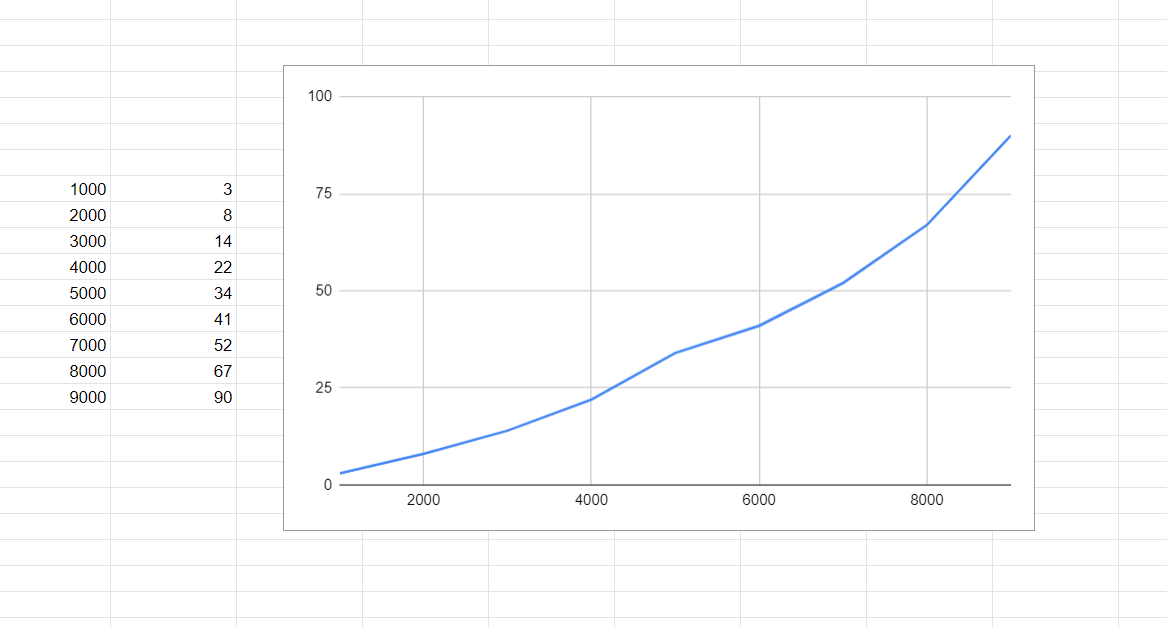
return 0;

}

**Output:**

****

**Graph:**

****

**PROGRAM 6**

**TOPOLOGICAL SORTING**

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

**Program:**

#include <stdio.h>

const int MAX = 10;

void topologicalOrder(int a[MAX][MAX], int n)

{

int in[MAX], out[MAX], stack[MAX], top=-1;

int i,j,k=0;

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

{

in[i] = 0;

for (j=0; j<n; j++)

if (a[j][i] == 1)

in[i]++;

}

while(1)

{

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

{

if (in[i] == 0)

{

stack[++top] = i;

in[i] = -1;

}

}

if (top == -1)

break;

out[k] = stack[top--];

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

{

if (a[out[k]][i] == 1)

in[i]--;

}

k++;

}

printf("Topological Sorting is: \n");

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

printf("%d ",out[i] + 1);

}

int main(void)

{

int a[MAX][MAX],n;

int i,j;

printf("\nEnter the number of vertices : ");

scanf("%d",&n);

printf("Enter the adjacency matrix: \n");

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

for (j=0; j<n; j++)

scanf("%d",&a[i][j]);

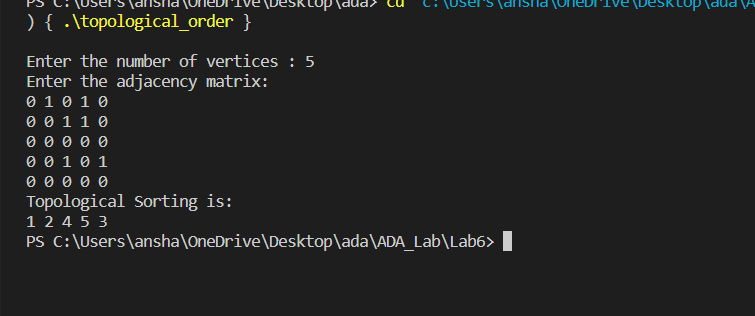
topologicalOrder(a,n);

printf("\n");

return 0;

}

**Output:**

****

**PROGRAM 7**

**JOHNSON TROTTER**

| Implement Johnson Trotter algorithm to generate permutations. |
| --- |

**Program:**

#include <stdio.h>

int LEFT\_TO\_RIGHT = 1;

int RIGHT\_TO\_LEFT = 0;

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int searchArr(int a[], int n, int mobile)

{

for (int i = 0; i < n; i++)

if (a[i] == mobile)

return i + 1;

}

int getMobile(int a[], int dir[], int n)

{

int mobile\_prev = 0, mobile = 0;

for (int i = 0; i < n; i++)

{

if (dir[a[i]-1] == RIGHT\_TO\_LEFT && i!=0)

{

if (a[i] > a[i-1] && a[i] > mobile\_prev)

{

mobile = a[i];

mobile\_prev = mobile;

}

}

if (dir[a[i]-1] == LEFT\_TO\_RIGHT && i!=n-1)

{

if (a[i] > a[i+1] && a[i] > mobile\_prev)

{

mobile = a[i];

mobile\_prev = mobile;

}

}

}

if (mobile == 0 && mobile\_prev == 0)

return 0;

else

return mobile;

}

int printOnePerm(int a[], int dir[], int n)

{

int mobile = getMobile(a, dir, n);

int pos = searchArr(a, n, mobile);

if (dir[a[pos - 1] - 1] == RIGHT\_TO\_LEFT)

swap(&a[pos-1], &a[pos-2]);

else if (dir[a[pos - 1] - 1] == LEFT\_TO\_RIGHT)

swap(&a[pos], &a[pos-1]);

for (int i = 0; i < n; i++)

{

if (a[i] > mobile)

{

if (dir[a[i] - 1] == LEFT\_TO\_RIGHT)

dir[a[i] - 1] = RIGHT\_TO\_LEFT;

else if (dir[a[i] - 1] == RIGHT\_TO\_LEFT)

dir[a[i] - 1] = LEFT\_TO\_RIGHT;

}

}

for (int i = 0; i < n; i++)

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

printf("\n");

}

int fact(int n)

{

int res = 1;

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

res = res \* i;

return res;

}

void printPermutation(int n)

{

int a[n];

int dir[n];

for (int i = 0; i < n; i++)

{

a[i] = i + 1;

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

}

printf("\n");

for (int i = 0; i < n; i++)

dir[i] = RIGHT\_TO\_LEFT;

for (int i = 1; i < fact(n); i++)

printOnePerm(a, dir, n);

}

int main()

{

int n = 4;

printPermutation(n);

return 0;

}

**Output:**

****

**PROGRAM 8**

**MERGE SORT**

| 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. |
| --- |

**Program:**

#include <stdio.h>

#include <time.h>

#define MAX 10000

void merge(int a[], int i1, int j1, int i2, int j2)

{

int temp[MAX];

int i, j, k;

i = i1;

j = i2;

k = 0;

while (i <= j1 && j <= j2)

{

if (a[i] < a[j])

temp[k++] = a[i++];

else

temp[k++] = a[j++];

}

while (i <= j1)

temp[k++] = a[i++];

while (j <= j2)

temp[k++] = a[j++];

for (i = i1, j = 0; i <= j2; i++, j++)

a[i] = temp[j];

}

void mergesort(int a[], int i, int j)

{

int mid;

if (i < j)

{

mid = (i + j) / 2;

mergesort(a, i, mid);

mergesort(a, mid + 1, j);

merge(a, i, mid, mid + 1, j);

}

}

int main(void)

{

int n = 1000;

printf("\nTime taken each, for 10 observations:\n");

while (n < MAX)

{

int a[n];

clock\_t start, end;

start = clock();

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

{

a[n - i] = i;

}

for (int i = 0; i < 100; i++)

{

mergesort(a, 0, n - 1);

}

end = clock();

printf("%f\n", (double)(end - start) / CLOCKS\_PER\_SEC \* 1000);

n = n + 1000;

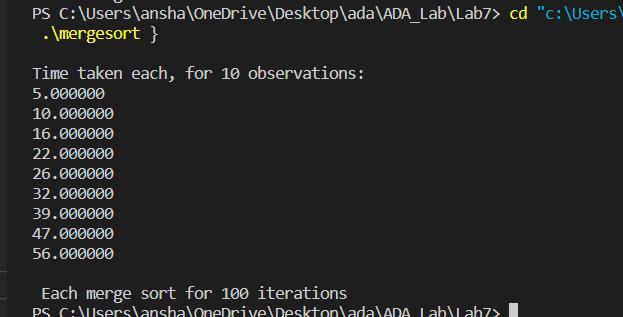
}

printf("\n Each merge sort for 100 iterations");

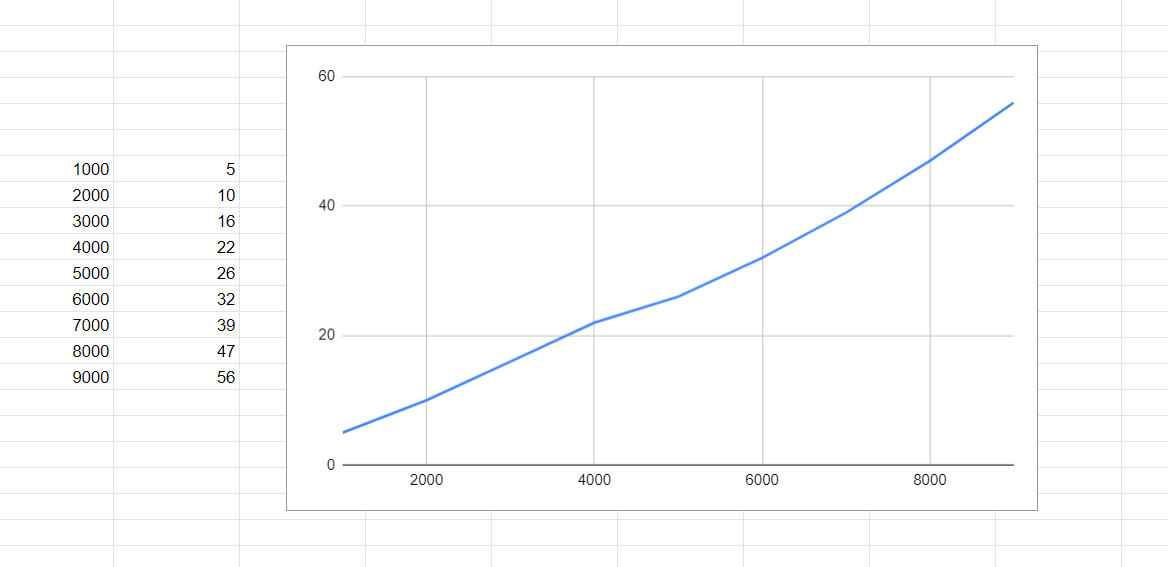
return 0;

}

**Output:**

****

**Graph:**

****

**PROGRAM 9**

**QUICK SORT**

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

**Program:**

#include <stdio.h>

#include <time.h>

#define MAX 10000

int partition(int a[], int l, int u)

{

int v, i, j, temp;

v = a[l];

i = l;

j = u + 1;

do

{

do

i++;

while (a[i] < v && i <= u);

do

j--;

while (v < a[j]);

if (i < j)

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

} while (i < j);

a[l] = a[j];

a[j] = v;

return (j);

}

void quick\_sort(int a[], int l, int u)

{

int j;

if (l < u)

{

j = partition(a, l, u);

quick\_sort(a, l, j - 1);

quick\_sort(a, j + 1, u);

}

}

int main(void)

{

int n = 1000;

printf("\nTime taken each, for 10 observations:\n");

while (n < MAX)

{

int a[n];

clock\_t start, end;

start = clock();

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

{

a[i] = i;

}

quick\_sort(a, 0, n - 1);

end = clock();

printf("%f\n", (double)(end - start) / CLOCKS\_PER\_SEC \* 1000);

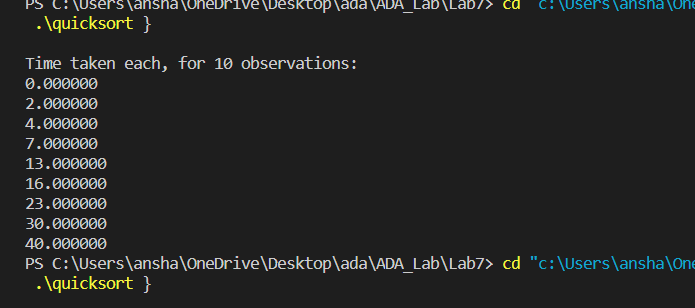
n = n + 1000;

}

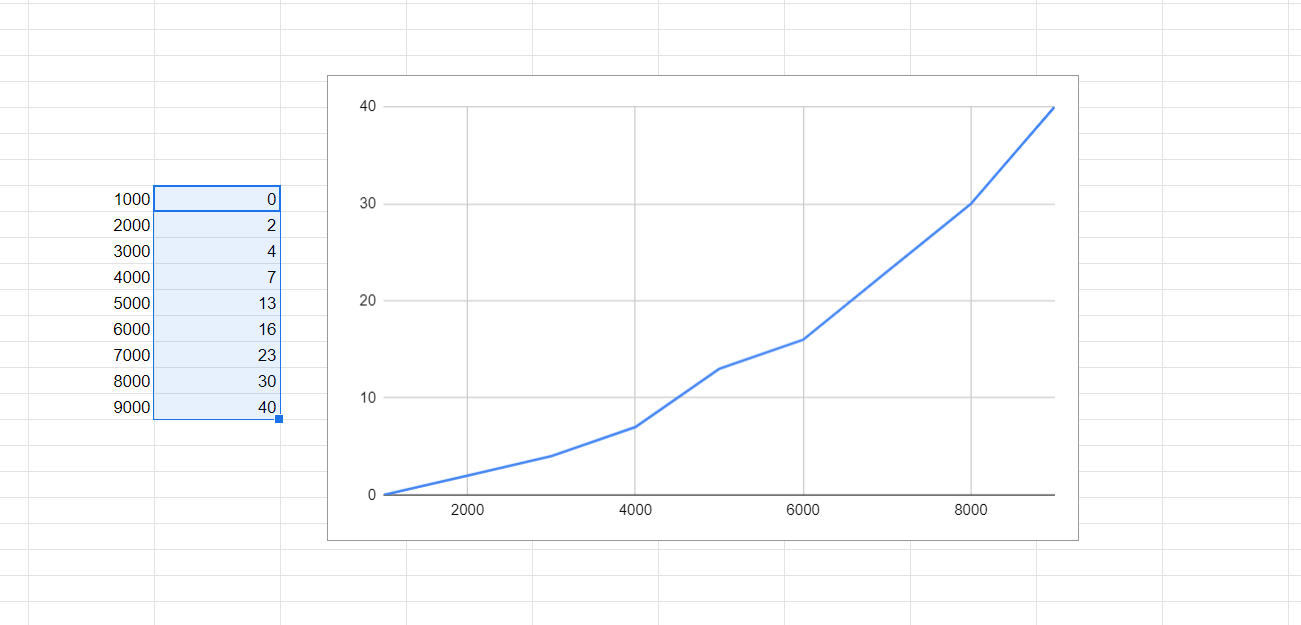
return 0;

}

**Output:**

****

**Graph:**

****

**PROGRAM 10**

**Heap Sort**

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

**Program:**

#include <stdio.h>

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void heapify(int arr[], int n, int i)

{

int largest = i;

int l = 2 \* i;

int r = (2 \* i) + 1;

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

largest = l;

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

largest = r;

if (largest != i)

{

swap(&arr[i], &arr[largest]);

heapify(arr, n, largest);

}

}

void heapsort(int arr[], int n)

{

for (int i = n / 2; i >= 1; i--)

{

heapify(arr, n, i);

}

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

{

swap(&arr[1], &arr[i]);

heapify(arr, i - 1, 1);

}

}

int main(void)

{

int arr[20];

int i, j, size;

printf("Enter the number of elements to sort : ");

scanf("%d", &size);

printf("Enter elements of array:\n");

for (i = 1; i <= size; i++)

{

scanf("%d", &arr[i]);

}

heapsort(arr, size);

printf("Sorted elements:\t");

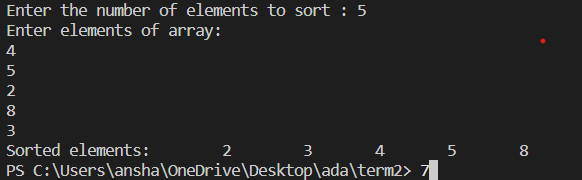
for (i = 1; i <= size; i++)

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

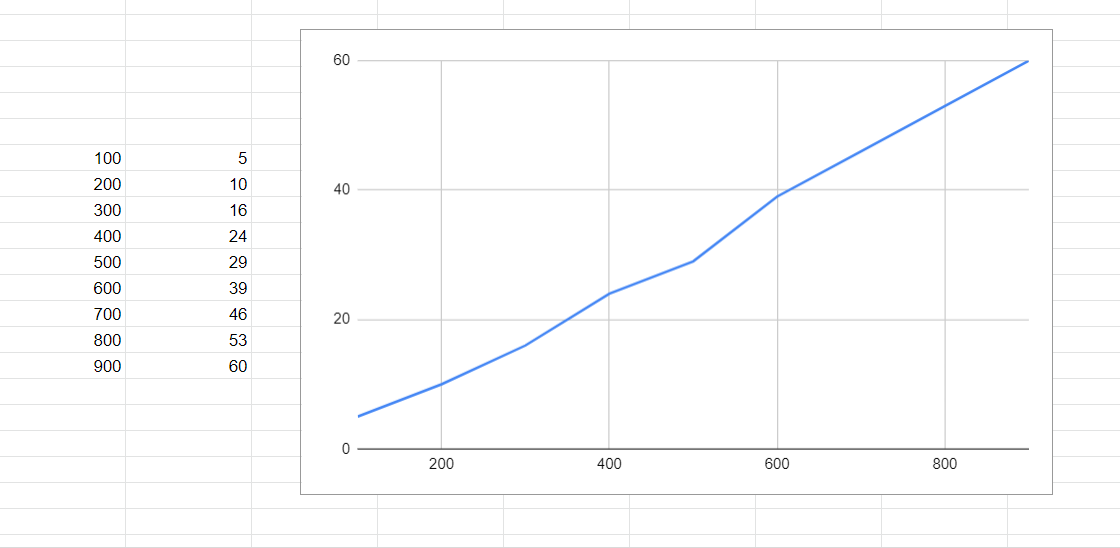
return 0;

}

**Output**:



**Graph**:

****

**PROGRAM 11**

**Warshall’s algorithm**

| Implement Warshall’s algorithm using dynamic programming |
| --- |

**Program:**

#include <stdio.h>

#define MAX 100

void WarshallTransitiveClosure(int graph[MAX][MAX], int v)

{

int i, j, k;

for (k = 0; k < v; k++)

{

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

{

for (j = 0; j < v; j++)

{

if (graph[i][j] || (graph[i][k] && graph[k][j]))

graph[i][j] = 1;

}

}

}

}

int main(void)

{

int i, j, v; // v - number of vertices

int graph[MAX][MAX];

printf("Warshall's Transitive Closure\n");

printf("Enter the number of vertices : ");

scanf("%d", &v);

printf("Enter the adjacency matrix :\n");

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

for (j = 0; j < v; j++)

scanf("%d", &graph[i][j]);

WarshallTransitiveClosure(graph, v);

printf("\nThe transitive closure for the given graph is :\n");

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

{

for (j = 0; j < v; j++)

{

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

}

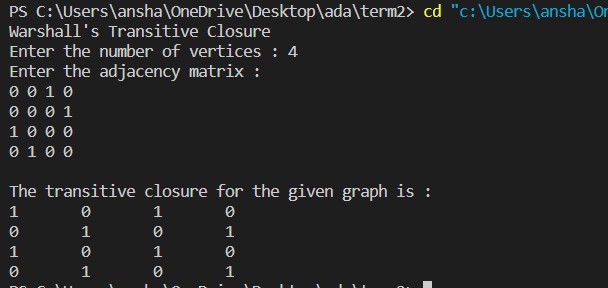
printf("\n");

}

return 0;

}

**Output**:



**PROGRAM 12**

**0/1 Knapsack problem**

| Implement 0/1 Knapsack problem using dynamic programming. |
| --- |

**Program:**

#include <stdio.h>

#define MAX 10

int max(int a, int b)

{

return a > b ? a : b;

}

// w - weight, p - profit

// n - number of items, c - knapsack capacity

void profitTable(int w[MAX], int p[MAX], int n, int c, int t[MAX][MAX])

{

int i, j;

for (j = 0; j <= c; j++)

t[0][j] = 0;

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

t[i][0] = 0;

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

{

for (j = 1; j <= c; j++)

{

if (j - w[i] < 0)

t[i][j] = t[i - 1][j];

else

t[i][j] = max(t[i - 1][j], p[i] + t[i - 1][j - w[i]]);

}

}

}

void backtrackSelect(int n, int c, int t[MAX][MAX], int w[MAX], int l[MAX])

{

int i, j;

i = n;

j = c;

while (i >= 1 && j >= 1)

{

if (t[i][j] != t[i - 1][j])

{

l[i] = 1;

j = j - w[i];

i--;

}

else

i--;

}

}

int main(void)

{

int i, j, totalProfit, num;

int weight[MAX], profit[MAX], loaded[MAX];

int capacity;

int table[MAX][MAX];

printf("Enter the maxium number of objects : ");

scanf("%d", &num);

printf("Enter the weights : \n");

for (i = 1; i <= num; i++)

{

printf("\nWeight of %d:", i);

scanf("%d", &weight[i]);

}

printf("\nEnter the profits : \n");

for (i = 1; i <= num; i++)

{

printf("\nProfit of %d:", i);

scanf("%d", &profit[i]);

}

printf("\nEnter the capacity : ");

scanf("%d", &capacity);

totalProfit = 0;

for (i = 1; i <= num; i++)

loaded[i] = 0;

profitTable(weight, profit, num, capacity, table);

backtrackSelect(num, capacity, table, weight, loaded);

printf("\nProfit matrix : \n");

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

{

for (j = 0; j <= capacity; j++)

{

printf("%d ", table[i][j]);

}

printf("\n");

}

printf("\nLoaded items : \n");

for (i = 1; i <= num; i++)

{

if (loaded[i])

{

printf("%d ", i);

totalProfit += profit[i];

}

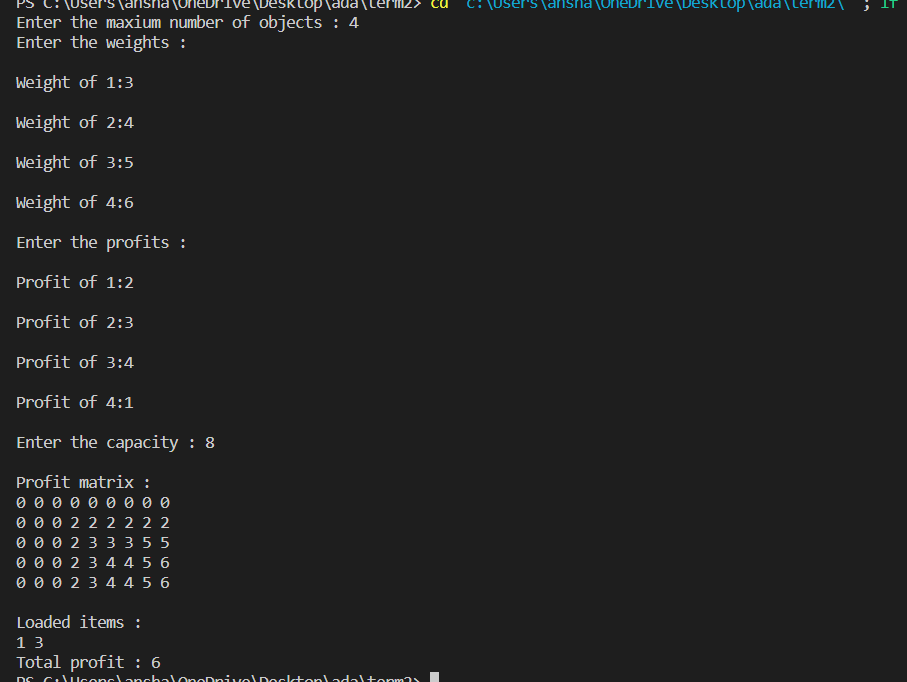
}

printf("\nTotal profit : %d", totalProfit);

return 0;

}

**Output**:



**PROGRAM 13**

**Floyd’s algorithm**

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

**Program:**

#include <stdio.h>

int min(int a, int b)

{

return (a < b ? a : b);

}

void floyd(int w[][10], int n)

{

int i, j, k;

for (k = 1; k <= n; k++)

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

for (j = 1; j <= n; j++)

w[i][j] = min(w[i][j], w[i][k] + w[k][j]);

}

int main()

{

int a[10][10];

int n, i, j;

printf("Enter the number of vertices:");

scanf("%d", &n);

printf("Enter the weighted matrix:\n");

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

for (j = 1; j <= n; j++)

scanf("%d", &a[i][j]);

floyd(a, n);

printf("\n All pairs shortest path\n");

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

{

for (j = 1; j <= n; j++)

{

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

}

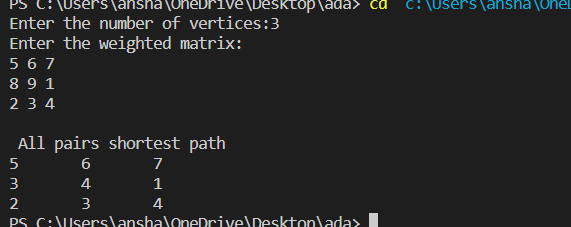
printf("\n");

}

return 0;

}

**Output**:



**PROGRAM 14**

**Prim’s algorithm**

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

**Program:**

#include <stdio.h>

int a, b, u, v, n, i, j, ne = 1;

int visited[10] = {0};

int min, mincost = 0, cost[10][10];

int main(void)

{

printf("\n Enter the number of nodes:");

scanf("%d", &n);

printf("\n Enter the adjacency matrix:\n");

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

for (j = 1; j <= n; j++)

{

scanf("%d", &cost[i][j]);

if (cost[i][j] == 0)

cost[i][j] = 999;

}

visited[1] = 1;

printf("\n");

while (ne < n)

{

for (i = 1, min = 999; i <= n; i++)

for (j = 1; j <= n; j++)

if (cost[i][j] < min)

if (visited[i] != 0)

{

min = cost[i][j];

a = u = i;

b = v = j;

}

if (visited[u] == 0 || visited[v] == 0)

{

printf("\n Edge %d:(%d %d) cost:%d", ne++, a, b, min);

mincost += min;

visited[b] = 1;

}

cost[a][b] = cost[b][a] = 999;

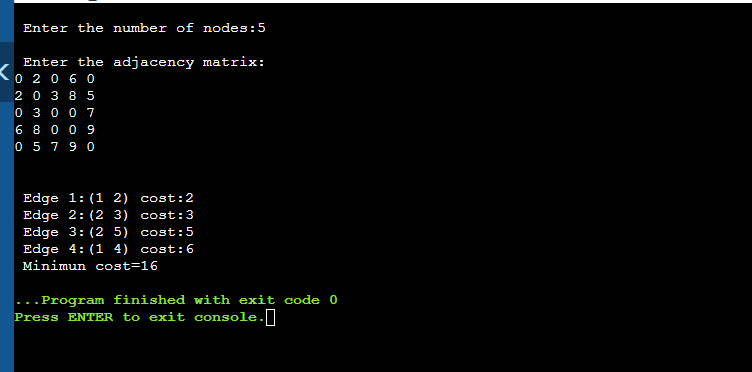
}

printf("\n Minimun cost=%d", mincost);

return 0;

}

**Output**:



**PROGRAM 15**

**Kruskals algorithm**

| Find Minimum Cost Spanning Tree of a given undirected graph using Kruskals algorithm. |
| --- |

**Program:**

#include <stdio.h>

#include <stdlib.h>

int i, j, k, a, b, u, v, n, ne = 1;

int min, mincost = 0, adj[9][9], parent[9];

int findparent(int i)

{

if (parent[i])

i = parent[i];

return i;

}

int uni(int i, int j)

{

if (i != j)

{

parent[j] = i;

return 1;

}

return 0;

}

int main(void)

{

printf("\nImplementation of Kruskal's algorithm\n");

printf("\nEnter the no. of vertices:");

scanf("%d", &n);

printf("\nEnter the adjacency matrix:\n");

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

{

for (j = 1; j <= n; j++)

{

scanf("%d", &adj[i][j]);

if (adj[i][j] == 0)

adj[i][j] = 999;

}

}

printf("\nThe edges of Minimum Cost Spanning Tree are\n");

while (ne < n) // ne:number of edges, n:number of vertices

{

for (i = 1, min = 999; i <= n; i++)

{

for (j = 1; j <= n; j++)

{

if (adj[i][j] < min)

{

min = adj[i][j];

a = u = i;

b = v = j;

}

}

}

u = findparent(u);

v = findparent(v);

if (uni(u, v))

{

printf("\n Edge %d:(%d %d) weight:%d", ne++, a, b, min);

mincost += min;

}

adj[a][b] = adj[b][a] = 999;

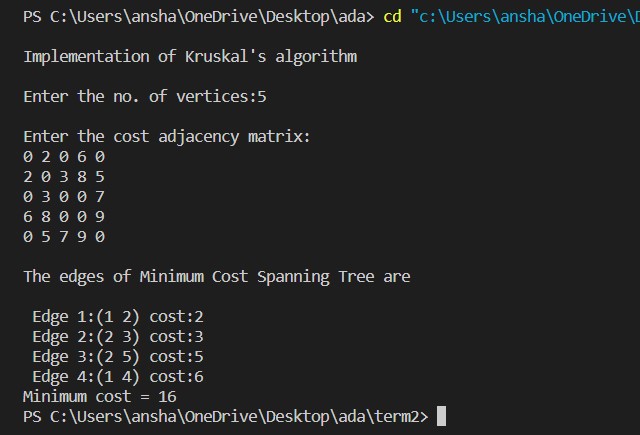
}

printf("\nMinimum cost = %d\n", mincost);

return 0;

}

**Output**:



**PROGRAM 16**

**Dijkstra’s algorithm**

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

**Program:**

#include <stdio.h>

#define INFINITY 9999

#define MAX 10

void dijikstra(int G[MAX][MAX], int n, int startnode)

{

int cost[MAX][MAX], distance[MAX], pred[MAX];

int visited[MAX], count, mindistance, nextnode, i, j;

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

for (j = 0; j < n; j++)

if (G[i][j] == 0)

cost[i][j] = INFINITY;

else

cost[i][j] = G[i][j];

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

{

distance[i] = cost[startnode][i];

pred[i] = startnode;

visited[i] = 0;

}

distance[startnode] = 0;

visited[startnode] = 1;

count = 1;

while (count < n - 1)

{

mindistance = INFINITY;

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

if (distance[i] < mindistance && !visited[i])

{

mindistance = distance[i];

nextnode = i;

}

visited[nextnode] = 1;

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

if (!visited[i])

if (mindistance + cost[nextnode][i] < distance[i])

{

distance[i] = mindistance + cost[nextnode][i];

pred[i] = nextnode;

}

count++;

}

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

if (i != startnode)

{

printf("\nDistance from %d : %d", i, distance[i]);

printf("\nPath : %d", i);

j = i;

do

{

j = pred[j];

printf(" <-%d", j);

} while (j != startnode);

}

}

int main(void)

{

int G[MAX][MAX], i, j, n, u;

printf("\nEnter the no. of vertices:: ");

scanf("%d", &n);

printf("\nEnter the adjacency matrix::\n");

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

for (j = 0; j < n; j++)

scanf("%d", &G[i][j]);

printf("\nEnter the starting node:: ");

scanf("%d", &u);

dijikstra(G, n, u);

return 0;

}

**Output**:

