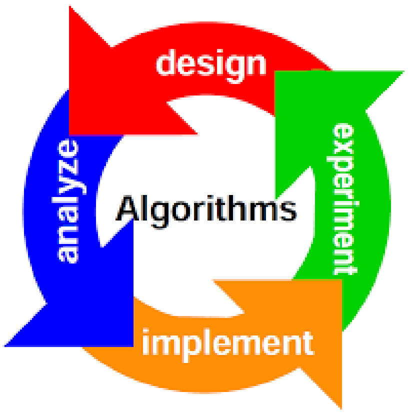
PES COLLEGE OF ENGINEERG MANDYA

DEPARTMENT

OF

COMPUTERSCIENCE AND ENGINEERING



DESIGN AND ANALYSIS OF ALGORITHMS LAB MANUAL

Year	:	2023-2024
Course Code	:	P21CSL406
Semester	:	IV
Branch	:	CSE

Preface

Algorithms play the central role both in the science and practice of computing. There are three principal reasons for emphasis on algorithm design techniques. First, these techniques provide a student with tools for designing algorithms for new problems. This makes learning algorithm design techniques a very valuable endeavor from a practical standpoint. Second, they seek to classify multitudes of known algorithms according to an underlying design idea. Learning to see such commonality among algorithms from different application areas should be a major goal of computer science education. Third, algorithm design techniques have utility as general problem solving strategies, applicable to problems beyond computing.

If you are going to be a computer professional, there are both practical and theoretical reasons to study algorithms. From a practical standpoint, you have to know a standard set of important algorithms from different areas of computing; in addition, you should be able to design new algorithms and analyze their efficiency. From the theoretical standpoint, the study of algorithms, as come to be recognized as the cornerstone of computer science.

This manual can serve as a handbook for implementing the various algorithm design techniques.

PES COLLEGE OF ENGINEERING
(Autonomous)

Department of Computer Science and Engineering

Mandya-571401

1. PROGRAM OUTCOMES:

B.E- PROGRAM OUTCOMES (POS)	
PO-1	Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems**(Engineering knowledge).**
PO-2	Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineeringsciences **(Problem analysis)**.
PO-3	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations **(Design/development of solutions)**.
PO-4	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions **(Conduct investigations of complex problems)**.
PO-5	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations **(Modern tool usage)**.
PO-6	Apply reasoning informed by the contextual knowledge to assesssocietal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice **(The engineer and society)**.
PO-7	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development **(Environmentand sustainability)**.
PO-8	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice **(Ethics)**.
PO-9	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings **(Individual and team work)**.
PO-10	Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions **(Communication)**.
PO-11	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and inmultidisciplinary environments **(Project management and finance)**.
PO-12	Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change **(Life-long learning)**.

2. PROGRAM SPECIFIC OUTCOMES

PSO 1	Ability to apply problem solving skills in developing solutions through fundamentals of Computer Science and Engineering.
PSO 2	Ability to apply Analytical Skills in the field of Data Processing Systems.
PSO 3	Ability to design and develop applications through Software Engineering methodologies and Networking Principles.

Course Outcomes:

Course Outcomes : On completion of this course, students are able to:	
CO 1	Implement the algorithms based on various algorithm design techniques
CO 2	Analyze the efficiency of various algorithms.

3. ATTAINMENT OF PROGRAM OUTCOMES AND PROGRAM SPECIFIC OUTCOMES:

Sl.No	Experiment	Program Outcomes Attained	Program Specific Outcomes Attained
WEEK-l	**BFS** Print all the nodes reachable from a given starting node in a digraph using BFS method.	2	1
WEEK-2	**Topological ordering (DFS Based).** Obtain the Topological ordering of vertices in a given digraph (DFS Based).	2	1
WEEK-3	**Merge sort** Sort a given set of elements using Merge sort method and determine the time taken to sort the elements. Repeat the experiment for different values of *n*, the number of elements in the list to be sorted and plot a graph of the time taken versus *n*.	2,3	1
WEEK-4	**Quick sort** Sort a given set of elements using Quick sort method and determine the time taken to sort the elements. Repeat the experiment for different values of *n*, the number of elements in the list to be sorted and plot a graph of the time taken versus *n*.	2,3	1
WEEK-5	**Horspool's String Matching** Find the Pattern string in a given Text string using Horspool's String Matching Algorithm.	3	1
WEEK-6	**Heap Sort** Sort a given set of elements using Heap Sort algorithm.	2,3	1
WEEK-7	**Knapsack problem** Implement 0/1 Knapsack problem using Dynamic Programming.	3	1
WEEK-8	**Dijikstra's algorithm** From a given vertex in a weighted connected graph, find shortest paths to other Vertices using Dijikstra's algorithm.	3	1
WEEK-9	**Kruskal's Algorithm** Find minimum cost spanning tree of a given undirected graph using Kruskal's Algorithm.	3	1
WEEK-10	**Sum-of-Subset problem** Implement Sum-of-Subset problem of a given set S = {s_1, s_2, ..., s_n} of 'n' Positive integers whose sum is equal to a given positive integer 'd'.	3	1

4. MAPPING COURSE OBJECTIVES LEADING TO THE ACHIEVEMENT OF PROGRAM OUTCOMES:

Course Objectives	Program Outcomes												Program Specific Outcomes		
	PO1	PO 2	PO 3	PO4	PO5	PO6	PO7	PO8	PO9	P O 10	P O 11	P O 12	PSO 1	PSO2	PSO 3
CO1	2	2	2		2							1	2	2	2
CO2	2	2											2	2	2

5. SYLLABUS:

DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY								
III Semester: CSE / IT								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
P21CSL406	**Core**	**L**	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
		-	-	2	1	50	50	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes:**				**Total Classes:**		

OBJECTIVES:
The course should enable the students to:
Learn how to analyze a problem and design the solution for the problem.
I. Design and implement efficient algorithms for a specified application.
II. Strengthen the ability to identify and apply the suitable algorithm for the given real world problem.

Sl.No	**LIST OF EXPERIMENTS**
1	Print all the nodes reachable from a given starting node in a digraph using BFS method.
2	Obtain the Topological ordering of vertices in a given digraph (DFS Based).
3	Sort a given set of elements using Merge sort method and determine the time taken to sort the elements. Repeat the experiment for different values of *n*, the number of elements in the list to be sorted and plot a graph of the time taken versus *n*.
4	Sort a given set of elements using Quick sort method and determine the time taken to sort the elements. Repeat the experiment for different values of *n*, the number of elements in the list to be sorted and plot a graph of the time taken versus *n*.
5	Find the Pattern string in a given Text string using Horspool's String Matching Algorithm.
6	Sort a given set of elements using Heap Sort algorithm.
7	Implement 0/1 Knapsack problem using Dynamic Programming.
8	From a given vertex in a weighted connected graph, find shortest paths to other Vertices using Dijikstra's algorithm.
9	` Find minimum cost spanning tree of a given undirected graph using Kruskal's Algorithm.
10	Implement Sum-of-Subset problem of a given set S = {s_1, s_2, ..., s_n} of 'n' Positive integers whose sum is equal to a given positive integer 'd'.

6. INDEX:

Program 1:

Print all the nodes reachable from a given starting node in a digraph using BFS method.

Breadth first search

It is a method used to traverse the given graph (i.e, visiting all the vertices in the given graph exactly once).

Method:

- It proceeds by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited.
- If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.

Procedure:

- Queue is a convenient data structure to trace the operation of breadth-first search.
- The queue is initialized with the traversal's starting vertex.
- Mark it as visited.
- On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the frontvertex of the queue, mark them as visited, and add them to the queue.
- Then remove the front vertex from the queue.

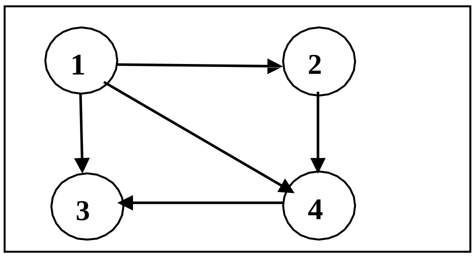
Algorithm for BFS traversalAlgorithm BFS(G)

```
 //Implements a breadth-first search traversal of a given graph
 //Input: Graph G =V,E
 //Output: Graph G with its vertices marked with consecutive integers in the order they are visited by the
         BFS traversal

mark each vertex in V with 0 as a mark of being "unvisited"count ← 0
for each vertex v in V do
    if v is marked with 0
    bfs(v)

Algorithm bfs(v)
//visits all the unvisited vertices connected to vertex v by a path and numbers them in the order they are
visited via global variable count
count ← count + 1;
mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex doif w is marked with 0
            count ← count + 1;
            mark w with count add w to the queue
remove the front vertex from the queue
```

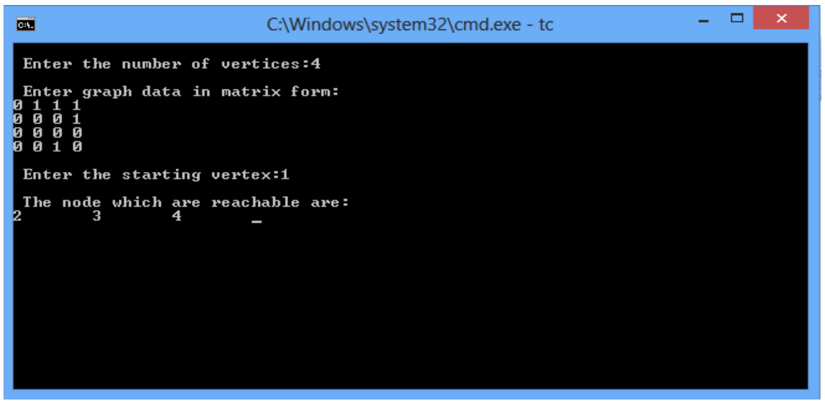
Program

```
#include<stdio.h>
#include<conio.h>
int a[20][20],q[20],visited[20],n, i, j,f=0,r=-1;

void bfs(int v)
{
for(i=1;i<=n;i++)
if(a[v][i]  && !visited[i])
q[++r]=i;
if(f<=r)
          {
          visited[q[f]]=1;
          bfs(q[f++]);
          }
}

void main()
{
int v; clrscr();
printf("\n Enter the number of vertices:");scanf("%d",&n);
for(i=1;i<=n;i++)
          {
          q[i]=0;
          visited[i]=0;
          }
printf("\n Enter graph data in matrix form:\n");
for(i=1;i<=n;i++)
          for(j=1;j<=n;j++)
                scanf("%d",&a[i][j]);

printf("\n Enter the starting vertex:");scanf("%d",&v);
bfs(v);
printf("\n The node which are reachable are:\n");
for(i=1;i<=n;i++)
      if(visited[i])
        printf("%d\t",i); getch();
}
```

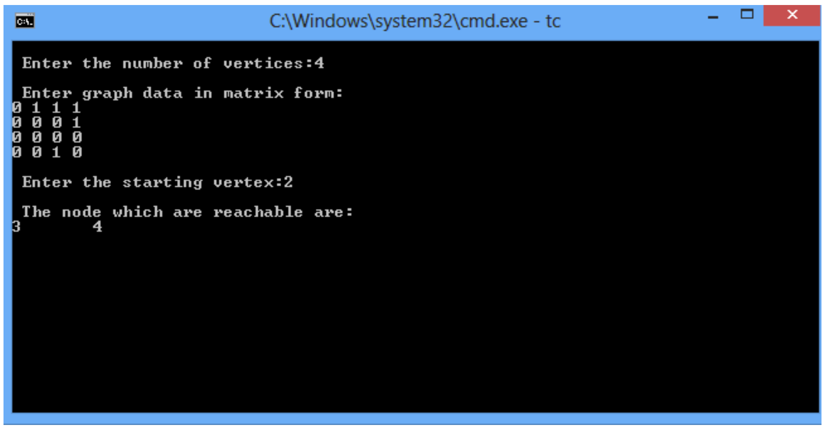
Input Graph:



Output:



```
 Enter the number of vertices:4
 Enter graph data in matrix form:
0 1 1 1
0 0 0 1
0 0 0 0
0 0 1 0
 Enter the starting vertex:1
 The node which are reachable are:
2       3       4       _
```



```
 Enter the number of vertices:4
 Enter graph data in matrix form:
0 1 1 1
0 0 0 1
0 0 0 0
0 0 1 0
 Enter the starting vertex:2
 The node which are reachable are:
3       4
```

Program 2:

Obtain the Topological ordering of vertices in a given digraph (DFS Based).

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering.

Note: Topological Sorting for a graph is not possible if the graph is not a DAG.

Procedure:

Maintain two time stamps for each node where first time stamp represents the time at which the node is visited and the second represents the time at which the node becomes the dead end.
Start from the first vertex in the adjacency matrix of the given graph.i.e.,0
At time 1 we visit the node ‘0’ and at time 2 node 0 becomes dead end as there is no adjacent vertex to node ‘0’.

The next unvisited vertex in the matrix is node ‘1’ ,so at time ‘3’ node ‘1’ is visited and at time ‘4’ node ‘1’ becomes dead end as there is no adjacent vertex to node ‘1’.

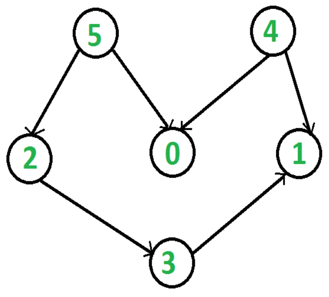
The next unvisited vertex in the matrix is node ‘2’ ,so at time ‘5’ node ‘2’ is visited ,as there is adjacent vertex to node ‘2’ i.e.,node ‘3’ at time ‘6’ node ‘3’ is visited .There is a adjacent vertex node ‘1’ to node ‘3’ ,but it is already been visited and also since there are no adjacent vertices to node ‘3’,it becomes dead end at time 7.Get back to node 2 from which node 3 was visited .as there is no adjacent vertex to node 2,it becomes dead end at time 8.

The next unvisited vertex in the matrix is node ‘4’ .So at time 9 node 4 is visited , Though there are adjacent vertices ‘1’ and ‘0’ to node ‘4’ and they are already been visited, at time 10 node ‘4’ becomes dead end .

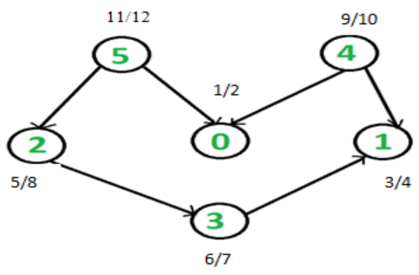
The next unvisited vertex in the matrix is node ‘5’ .So at time 11 node 5 is visited , Though there are adjacent vertices ‘2’ and ‘0’ to node ‘5’ and they are already been visited, at time 12 node ‘5’ becomes dead end .

Now as all the vertices have been visited, Arrange the nodes in decreasing order of their second time stamp, Then we obtain the topological sequence using DFS method.

Given graph is:



After applying the DFS traversal for the above graph,the resultant graphn will be as shown below:



Arrange the nodes in descending order according to second time stamp

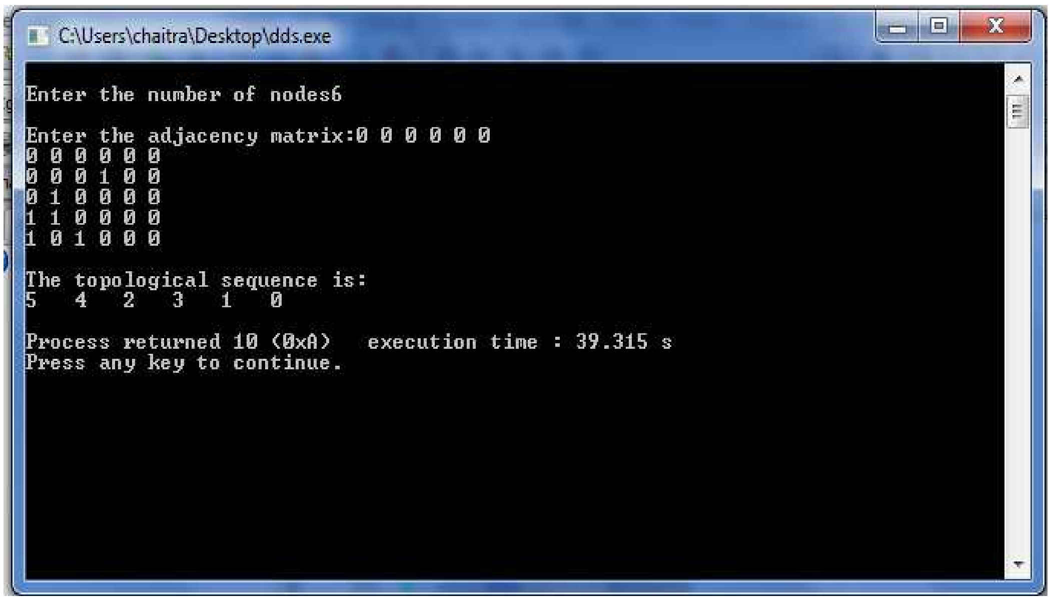
11/12	9/10	5/8	6/7	3/4	1/2
5	4	2	3	1	0

The topological sequence is: 5 4 2 3 1 0

Program:

```
#include<stdio.h>

int res[20];  //store the dead vertex or which is completely explored
int s[20];   //to know what nodes are visited and what nodes are not visited
int j=0;    //index variable for array res[]

void dfs(int u,int n,int cost[10][10])
{
int v;
//visit the vertex u
s[u]=1;
//traverse deeper into the graph till we get the dead end or till all the vertices are visited
for(v=0;v<n;v++)
{
if(cost[u][v]==1  && s[v]==0)
{
dfs(v,n,cost);
}
}
//store the dead vertex or which is completely explored
res[j++]=u;
}


void depth_first_traversal(int n,int a[10][10])
{
int i;
//initialisation to indicate that no vertex has been visited
```

```
for(i=0;i<n;i++)
s[i]=0;
```
//process each vertex in the graph
```
for(i=0;i<n;i++)
{
if(s[i]==0)
dfs(i,n,a);
}
}

void main()
{
int i,j,k,m,n,cost[10][10];
printf("\nEnter the number of nodes");
scanf("%d",&n);

printf("\nEnter the adjacency matrix:");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&cost[i][j]);

depth_first_traversal(n,cost);

printf("\nThe topological sequence is:\n");
for(i=n-1;i>=0;i--)
printf("%d   ",res[i]);
}
```


C:\Users\chaitra\Desktop\dds.exe

```
Enter the number of nodes6

Enter the adjacency matrix:0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 1 0 0
0 1 0 0 0 0
1 1 0 0 0 0
1 0 1 0 0 0

The topological sequence is:
5   4   2   3   1   0

Process returned 10 (0xA)   execution time : 39.315 s
Press any key to continue.
```

Program 3:

Sort a given set of elements using Merge sort method and determine the time taken to sort the elements. Repeat the experiment for different values of *n*, the number of elements in the list to be sorted and plot a graph of the time taken versus *n*.

Mergesort

```
Algorithm  Mergesort(A,low,high)
//Purpose: Sort the elements of the array between lower bound and upper bound
//Input: A-Unsorted array with low and high as lower bound and upper bound
//Output: A is Sorted array

if(low<high)
        Mid=(low+high)/2
        Mergesort(A,low,mid)
        Mergesort(A,mid+1,high)
        Simplemerge(A,low,mid,high)
```

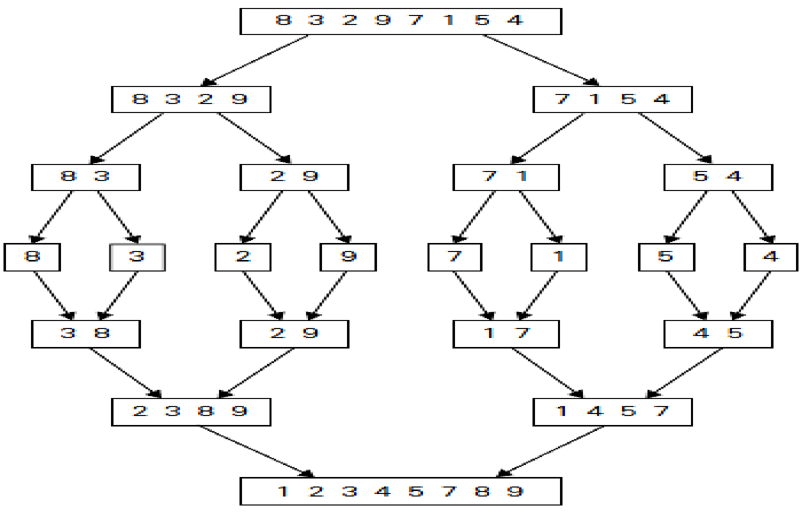
Mergesort is a perfect example of a successful application of the divide-and conquer technique. It sortsa given array A[0..n − 1] by dividing it into two halves A[0..n/2 − 1] and A[n/2..n− 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

The merging of two sorted arrays can be done as follows.
Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

```
Algorithm SimpleMerge(A,low,mid,high)
  //Purpose: Merges two sorted arrays where first array starts from low to mid and second array starts from
             mid+1 to high
  //Input: Arrays A[0..mid] and A[mid+1…high] both sorted
  //Output: Sorted array A[0..high]

  i←low; j ←mid+1; k←low
  while(  i<=mid and j <=high)
  if (A[i]≤ A[j]) then
      C[k++]←A[i++];
  else
      C[k++]←A[j ++];
end if
  end while
while(i<=mid)
  C[k++]←A[i++];
```

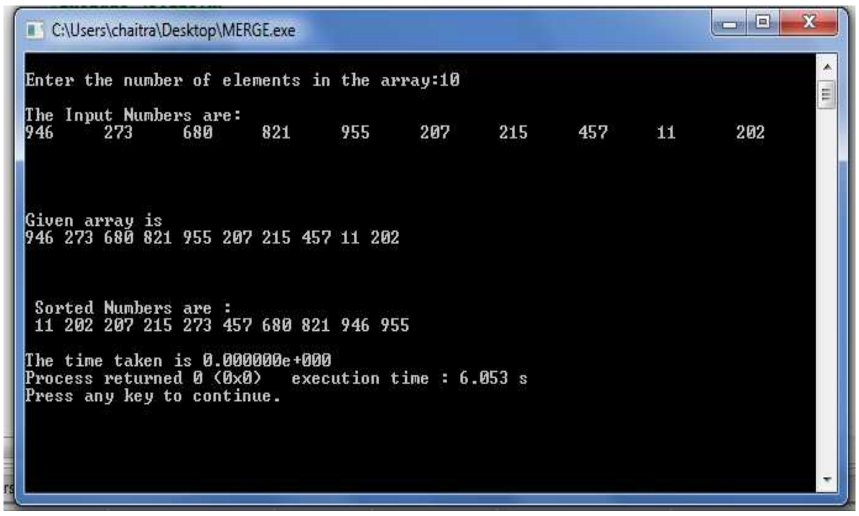
```
   while(j<=high)
         C[k++]←A[j++];
for(i=0;i<n;i++)
      A[i]=C[i]
Print Array A
```

Example:



Program:

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

int C[20];
void Merge(int a[ ], int low, int mid, int high)
{
int i, j, k;
i=low; j=mid+1; k=low;
while ( i<=mid && j<=high )
{
if( a[i] <= a[j] )
C[k++] = a[i++] ;
else
C[k++] = a[j++] ;
}
while (i<=mid)
C[k++] = a[i++] ;
while (j<=high)
C[k++] = a[j++] ;
for(k=low; k<=high; k++)
a[k] = C[k];
}

void MergeSort(int a[], int low, int high)
{
int mid;

if(low >= high)
return;
mid = (low+high)/2 ;
MergeSort(a, low, mid);
MergeSort(a, mid+1, high);
Merge(a, low, mid, high);
}

int main( )
{
int n, a[100],k;
clock_t st,et;
double ts;
printf("\n Enter the number of elements to be sorted:");
scanf("%d", &n);
srand(time(0));
printf("\nThe Random Numbers are:\n");
for(k=1; k<=n; k++)
{
a[k]=rand()% 100;
printf("%d\t", a[k]);
}
st=clock();
MergeSort(a, 1, n);
et=clock();
ts=(double)(et-st)/CLK_TCK;
printf("\n\n\n Sorted Numbers are : \n ");
for(k=1; k<=n; k++)
{
printf("%d\t", a[k]);
}
printf("\nThe time taken is %e",ts);
return 0;
}
```


C:\Users\chaitra\Desktop\MERGE.exe

```
Enter the number of elements in the array:10

The Input Numbers are:
946     273     680     821     955     207     215     457     11      202



Given array is
946 273 680 821 955 207 215 457 11 202



 Sorted Numbers are :
 11 202 207 215 273 457 680 821 946 955

The time taken is 0.000000e+000
Process returned 0 (0x0)   execution time : 6.053 s
Press any key to continue.
```

Program 4:

Sort a given set of elements using Quick sort method and determine the time taken to sort the elements. Repeat the experiment for different values of *n*, the number of elements in the list to be sorted and plot a graph of the time taken versus *n*.

Quicksort is the other important sorting algorithm that is based on the divide-and conquers approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value. A partition is an arrangement of the array’s elements so that all the elements to the left of some element *A*[*s*] are less than or equal to *A*[*s*], and all the elements to the right of *A*[*s*] are greater than or equal to it:

Obviously, after a partition is achieved, *A*[*s*] will be in its final position in the sorted array, and we cancontinue sorting the two subarrays to the left and to the right of *A*[*s*] independently (e.g., by the

$$\underbrace{A[0]\ldots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are } \ge A[s]}$$

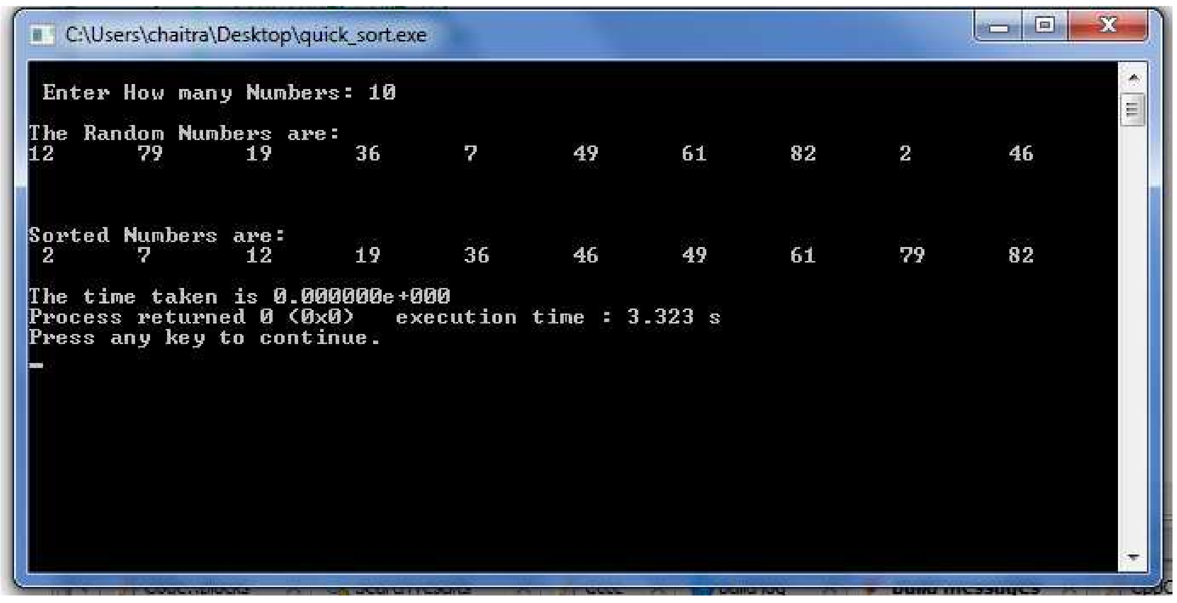
same method). Note the difference with mergesort: there, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

```
Algorithm  Quicksort(A[low..high])
//Purpose: Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right indices low and high
//Output: Subarray A[low..high] sorted in non decreasing order
if (low<high)
     mid←Partition(A[low..high])  //mid is a partitioned position
    Quicksort(A[low..mid-1])
     Quicksort(A[mid+1 ..high])


Algorithm  Partition(a[low..high])
//Purpose: Partitions a sub array  using the first element as a pivot
//Input:     Sub array of array a[0..n − 1], defined by its left and right indices low  and high (low<high)
//Output:  Partition of A[low..high], with the split position returned as this function’s value
              Key

key=a[low];
i=low; j=high+1;
while(i<=j)
        do    i←i+1       while(key>=a[i]);
        do    j←j+        while(key<a[j]);

    if(i<j)    exchange(a[i], a[j])
 end while

exchange(a[j], a[low]);
return j;
```

Program:

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

int partition(int a[], int low, int high)
{
   int i, j, key,temp;

   key=a[low];
   i=low; j=high+1;
   while(i<=j)
   {
      do
      {
         i++;
      }while(key>=a[i]);

      do
      {
         j--;
      }while(key<a[j]);

      if(i<j)
      {
         //swap the elements in position i and j
         temp=a[i];
         a[i]=a[j];
         a[j]=temp;
      }
   }
   //swap key element with element in position 'j'
   temp=a[low];
   a[low]=a[j];
   a[j]=temp;
   return j;
}



void  quick_sort(int a[],int low,int high)
{
   int mid;
   if(low<high)
   {
      mid=partition(a,low,high);
      quick_sort(a,low,mid-1);
```

```
        quick_sort(a,mid+1,high);
    }
}


int main()
{
    int n, a[100],k;
    clock_t st,et;
    double ts;
    printf("\n Enter How many Numbers: ");
    scanf("%d", &n);
    srand(time(0));
    printf("\nThe Random Numbers are:\n");
    for(k=0; k<n; k++)
    {
        a[k]=rand()%100;
        printf("%d\t",a[k]);
    }
    st=clock();
    quick_sort(a, 0, n-1);
    et=clock();
    ts=(double)(et-st)/CLK_TCK;
    printf("\n\n\nSorted Numbers are: \n ");
    for(k=0; k<n; k++)
    printf("%d\t", a[k]);
    printf("\nThe time taken is %e",ts);
}
```



```
 Enter How many Numbers: 10

The Random Numbers are:
12      79      19      36      7       49      61      82      2       46


Sorted Numbers are:
 2      7       12      19      36      46      49      61      79      82

The time taken is 0.000000e+000
Process returned 0 (0x0)   execution time : 3.323 s
Press any key to continue.
```

Program 5

Find the Pattern string in a given Text string using Horspool's String Matching Algorithm.

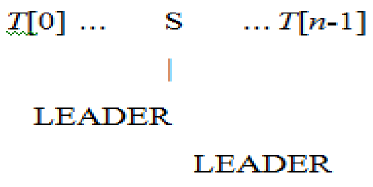
Horspool's Algorithm

Horspool's algorithm shifts the pattern by looking up shift value in the character of the text aligned with the last character of the pattern in table made during the initialization of the algorithm. The pattern is check with the text from right to left and progresses left to right through the text.

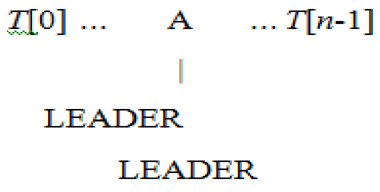
Let '*c*' be the character in the text that aligns with the last character of the pattern. If the pattern does notmatch there are 4 cases to consider.

The **mismatch occurs at the last character** of the pattern:

Case 1: ***c* does not exist** in the pattern (Not the mismatch occurred here) then shift pattern right by thesize of the pattern.

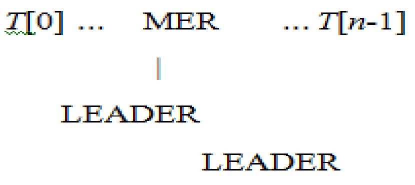


Case 2: The mismatch happens at the last character of the pattern and ***c* exists** in the pattern then theshift should be to the **right most *c*** in the *m*-1 remaining characters of the pattern.



The **mismatch happens in the middle** of the pattern:

Case 3: The mismatch happens in the middle (therefore *c* is in pattern) and there are **no other *c* in the pattern** then the shift should be the pattern length.



Case 4: The mismatch happens in the middle of the pattern but **there is other *c* in pattern** then the shift should be the **right most *c*** in the *m*-1 remaining characters of the pattern.

```
T[0] ...  EDER      ... T[n-1]
           |
        LEADER
           LEADER
```

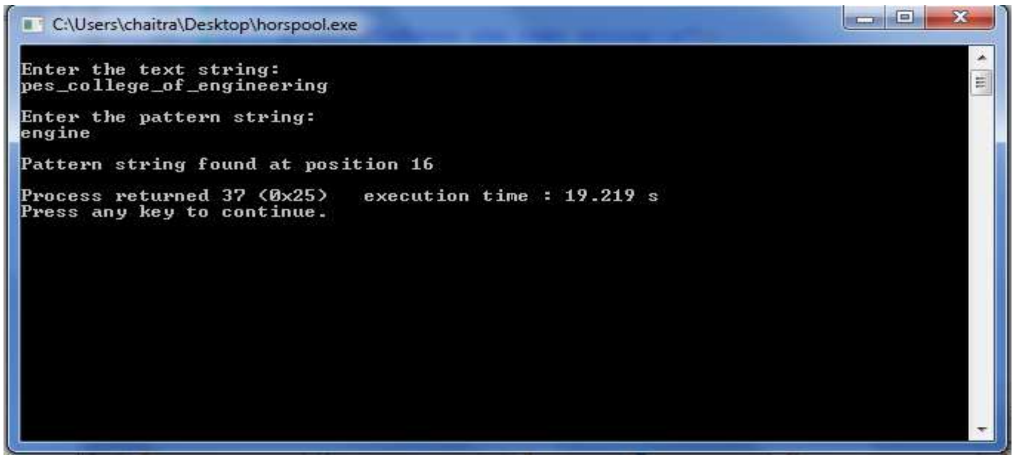
```
Algorithm  ShiftTable(P[0...m-1])
      Purpose:  To fill the shift table,based on the pattern string
      Input:     p-Pattern string to be searched
      Output:   t-Shift table is returned through parameter
    Step1:[obtain the length of the pattern string]
            m  ‹- length(p)
    step 2:Initialize the table with the length of the pattern string]
            for  i‹-0 to 127 do
                 s[i] ‹-m
            end for
    step 3:[Distance of the first m characters of string from the last character]
            for i‹-0 to m-2 do
                 s[p[j]] ‹- m-1-i
            end for
    step 4:finish
```

```
Algorithm HorspoolMatching(P[0...m-1], T[0...n-1])
//Purpose: To check if the pattern string is present in the text string
//Input: p-Pattern string
         t-text string
//Output:  Position of the pattern string ‘p’ in the text string ‘t’ is search
             is successful else   -1
Step1: [compute the shift table]
        Table ← ShiftTable(P[0...m-1])
Step 2: [Align the pattern string with the text string]
        n←length (t)      //compute the length of text string
        m←length (p)    //compute the length of pattern string
        i ← m-1 // position of the pattern string at the beginning of text string
Step 3: [Search for the pattern string in text string
            while i ≤ n-1 do
               k ← 0 // matching index
            while k ≤ m-1 and P[m-1-k]==T[i-k] // check matching
               k++
           end while
            if (k = m)
            return i-m+1 // return position of pattern string
        else
            i ← i + Table[T[i]]  // Shift the pattern from left to right
return -1
```

Program

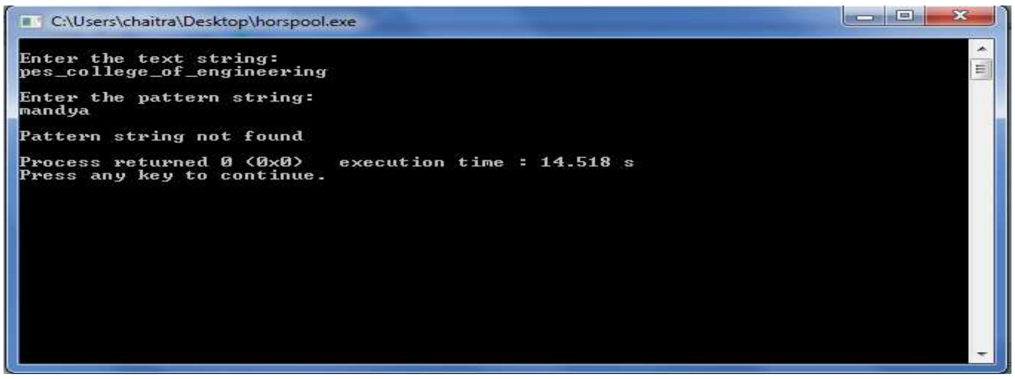
```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void shift_table(char p[],int t[])
{
   int i,m;
   /*length of pattern string*/
   m=strlen(p);
   /*initialize the table with pattern length*/
   for(i=0;i<128;i++)
      t[i]=m;
   /*find the distance of the first m-1 characters of the string from the last character*/
   for(i=0;i<=m-2;i++)
      {
         t[p[i]]=m-i-1;
      }

}

int horspool_pattern_match(char p[],char t[])
{
   int m,n,i,k,s[256];
   shift_table(p,s);

   m=strlen(p);
   n=strlen(t);
   i=m-1;

   while(i<=n-1)
   {
      k=0;
      while(k<=m-1 && t[i-k]==p[m-1-k])
      {
         k=k+1;
      }
      /*pattern found*/
      if(k==m)  return i-m+1;
      /*No match,so shift the pattern towards right*/
      i=i+s[t[i]];
   }
```

```
    return -1;
}

void main()
{
    char p[50],t[100];
    int pos;

    printf("\nEnter the text string:\n");
    scanf("%s",t);

    printf("\nEnter the pattern string:\n");
    scanf("%s",p);

    pos=horspool_pattern_match(p,t);

    if(pos==-1)
    {
        printf("\nPattern string not found\n");
        exit(0);
    }
    printf("\nPattern string found at position %d\n",pos+1);
}
```



C:\Users\chaitra\Desktop\horspool.exe

```
Enter the text string:
pes_college_of_engineering

Enter the pattern string:
engine

Pattern string found at position 16

Process returned 37 (0x25)   execution time : 19.219 s
Press any key to continue.
```



C:\Users\chaitra\Desktop\horspool.exe

```
Enter the text string:
pes_college_of_engineering

Enter the pattern string:
mandya

Pattern string not found

Process returned 0 (0x0)   execution time : 14.518 s
Press any key to continue.
```

Program 6:

Sort a given set of elements using Heap Sort algorithm.

A heap is a complete binary tree or an almost complete binary tree satisfying the parental dominance requirement (i.e., each item in the tree should be greater than their children or each item in the tree should be less than their children.)

A heap can be used to sort the numbers in ascending or descending order. This sorting technique which uses heap to arrange numbers in ascending/descending order is called heap sort.

The main advantage of this technique is that sorting can easily be implemented using arrays.
This sorting technique consists of two phases namely:

- Heap creation phase
- Delete root item and recreate the heap

Heap creation phase:

In heap creation phase, the unsorted array is transformed into heap and this process is called hepifying the array. If we want the items to be in ascending order it is required to construct the max-heap (root node contains highest value). If we want the items to be in descending order it is required to construct the min-heap (root node contains lowest value).

Sorting Phase:
In this phase, items are arranged in ascending order (if we use max heap) or descending order (if we use min heap).This is achieved by repeatedly performing the following activities.

- Exchange the root item to the end of the heap
- Decrement the size of the heap by 1 and recreate the heap by using the bottom_up_heapify()

Pseudo code for these two steps:

```
Exchange (a[0],a[i])
Bottom_up_heapify (i, a, 0);
```

Heap Sort Algorithm using Bottom up approach:

```
Algorithm heapsort (n,a[])
//purpose: To sort the items using heap
//Input:  a-the items to be arranged using sorting technique
          n-the number of items to be sorted.
//Output: a-Contains the sorted list

[create the heap]
for(i←(n-1)/2  down to 0)
    heapify (n,a,i)
end for
[Repeatedly exchange and recreate the heap]
for(i←n-1 down to 0)
    Exchange (a[0],a[i])
```

```
            Bottom_up_heapify (i, a, 0);
  end for
```

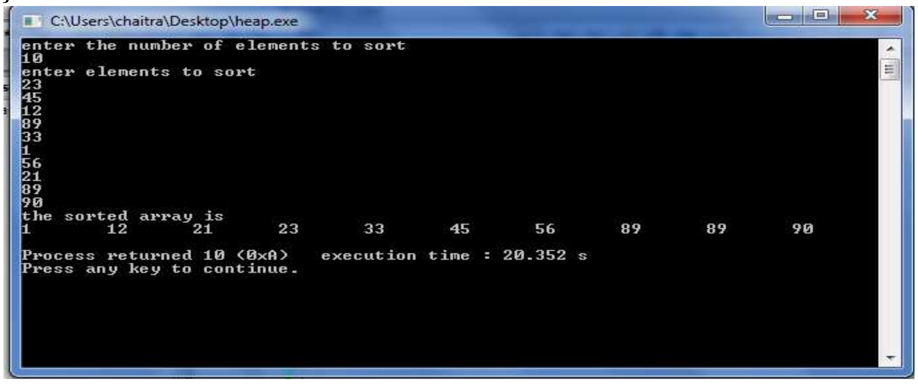
[The items are sorted and terminate the algorithm]

```
return
```

Program:

```
#include<stdio.h>
#include<conio.h>
void heapify(int a[],int n)
{
   int i,j,k,item;
   for(k=1;k<n;k++)
   {
      item=a[k];
      i=k;
      j=(i-1)/2;
      while(i>0 && item>a[j])
      {
         a[i]=a[j];
         i=j;
         j=(i-1)/2;
      }
      a[i]=item;
   }

}


void adjust(int a[],int n)
{
   int i,j,item;
   j=0;
   item=a[j];
   i=2*j+1;
   while(i<=n-1)
   {
      if(i+1<=n-1)
      if(a[i]<a[i+1])
      i++;
      if(item<a[i])
      {
         a[j]=a[i];
         j=i;
         i=2*j+1;
```

```
        }
        else
        break;
    }
    a[j]=item;
}

void heapsort(int a[],int n)
{
    int i,temp;
    heapify(a,n);
    for(i=n-1;i>0;i--)
    {
        temp=a[0];
        a[0]=a[i];
        a[i]=temp;
        adjust(a,i);
    }
}

void main()
{
int a[20],n,temp,i;
printf("enter the number of elements to sort\n");
scanf("%d",&n);
printf("enter elements to sort\n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
heapsort(a,n);
printf("the sorted array is\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
}
```



```
enter the number of elements to sort
10
enter elements to sort
23
45
12
89
33
1
56
21
89
90
the sorted array is
1       12      21      23      33      45      56      89      89      90

Process returned 10 (0xA)   execution time : 20.352 s
Press any key to continue.
```

Program 7:

Implement 0/1 Knapsack problem using Dynamic Programming.

In this problem we are given a knapsack of capacity M and ‘n’ objects of weights $w_1, w_2, ...w_n$ with ‘n’ profits $p_1, p_2, ...p_n$. The main objective is to place the objects into knapsack so that maximum profit is obtained and the weights of the objects chosen should not exceed the capacity of the knapsack.

The recurrence relation to get the solution to the knapsack problem can be written as shown below:

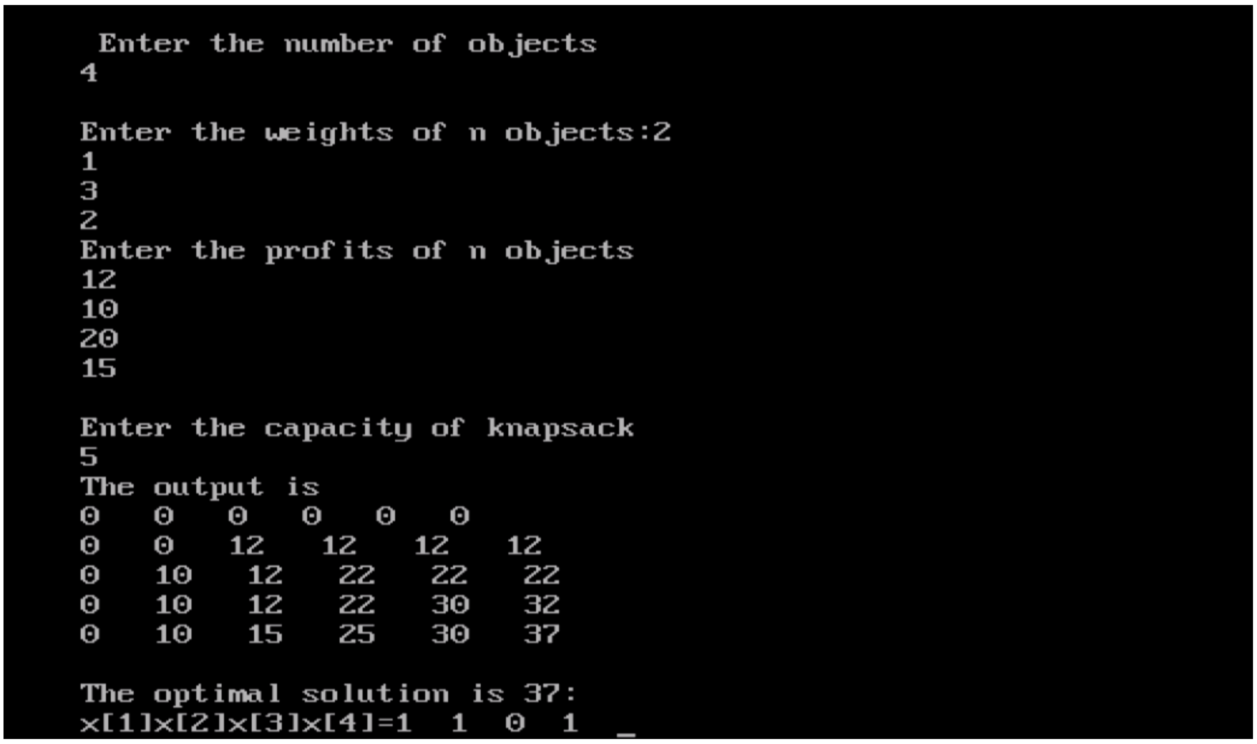
$$V[i, j] = \begin{cases} \max(V[i-1, j], V[i-1, j-w_i] + P_i) & \text{if } w_i \leq j \\ V[i-1, j] & \text{if } w_i > j \\ 0 & \text{if } i = j = 0 \end{cases}$$

Program:

```
#include<stdio.h>
int max(int a,int b)
{
        return a>b?a:b;
}

void knapsack(int n,int w[],int m,int v[][10],int p[])
{
int i,j;

for(i=0;i<=n;i++)
        {
        for(j=0;j<=m;j++)
                {
        if(i==0||j==0)
                v[i][j]=0;
        else if(j<w[i])
                v[i][j]=v[i-1][j];
        else
                v[i][j]=max(v[i-1][j],v[i-1][j-w[i]]+p[i]);

                }
        }
}

void print_optimal_sol(int n,int m,int w[],int v[10][10])
{
int i,j,x[10];
```

```
printf("\nThe optimal solution is %d:\n",v[n][m]);
/*Initially no object is selected*/

for(i=0;i<n;i++)
        x[i]=0;
i=n; /*number of objects*/
j=m;/*capacity of knapsack*/


while(i!=0  && j!=0)
        {
        if(v[i][j]!=v[i-1][j])
                {
                x[i]=1;
                j=j-w[i];
                }
        i=i-1;
        }
/*output the objects selected*/

for(i=1;i<=n;i++)
{
printf("x[%d]",i);
}
printf("=");
for(i=1;i<=n;i++)
        {
        printf("%d  ",x[i]);
        }
}

void main()
{
int m,n,i,j;
int p[10];
int w[10],v[10][10];
clrscr();
printf("\n Enter the number of objects\n");
scanf("%d",&n);
printf("\nEnter the weights of n objects:");
for(i=1;i<=n;i++)
        scanf("%d",&w[i]);
printf("Enter the profits of n objects\n");
for(i=1;i<=n;i++)
        scanf("%d",&p[i]);

printf("\nEnter the capacity of knapsack\n");
scanf("%d",&m);
```

```
knapsack(n,w,m,v,p);

printf("The output is\n");
for(i=0;i<=n;i++)
{
        for(j=0;j<=m;j++)
        {
                printf("%d   ",v[i][j]);
        }
                printf("\n");
}
print_optimal_sol(n,m,w,v);
}
```



```
 Enter the number of objects
4

Enter the weights of n objects:2
1
3
2
Enter the profits of n objects
12
10
20
15

Enter the capacity of knapsack
5
The output is
0   0   0   0   0   0
0   0   12   12   12   12
0   10   12   22   22   22
0   10   12   22   30   32
0   10   15   25   30   37

The optimal solution is 37:
x[1]x[2]x[3]x[4]=1  1  0  1  _
```

Program 8:

From a given vertex in a weighted connected graph, find shortest paths to other Vertices using Dijikstra’s algorithm.

Dijkstra's Algorithm (Single-Source Shortest Path Problem)

Method: For a given vertex called the source in a weighted connected graph, find shortest paths to all its other vertices. The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.
In other words, given a graph G = (V, E) and a source vertex vs, we find the shortest distance from vs to every other vertex vd in V.
This algorithm is applicable to undirected and directed graphs with nonnegative weights only.
This method is based on Greedy Technique.

Algorithm:

```
1. Read source vertex
2. Mark source vertex as visited
3. Initialize distance matrix d[ ] with sourceth row elements from the cost adjacency matrix
4. Find u and d[u] such that d[u] is minimum and uϵV-S
5. Add u to S
6. for every v which is not in S (i.e, v ϵV-S)
       Find d[v] = min(d[v], d[u]+cost[u][v])
   end for
7. repeat step 4 through 6 for remaining n-1 vertices
```

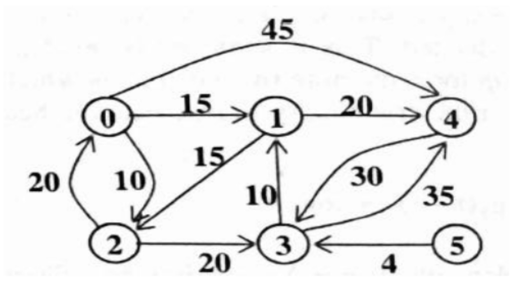
Program:

```
#include<stdio.h>
#include<conio.h>
int cost[10][10],d[10], p[10], n;
void dij(int source,int dest)
{
int i,j,u,v,min,S[10];
for(i=0;i<n;i++)
   {
   d[i]=cost[source][i];   it is used to have the shortest distence from the source vertex
   S[i]=0;
   p[i]=source;          it is used when we are printing the shortest paths to all vertex from source excluding the source
   }
S[source]=1;          its is used to notify that vertex is relaxed or visited
for(i=1;i<n;i++)
   {
   min=999;
   u=-1;
```

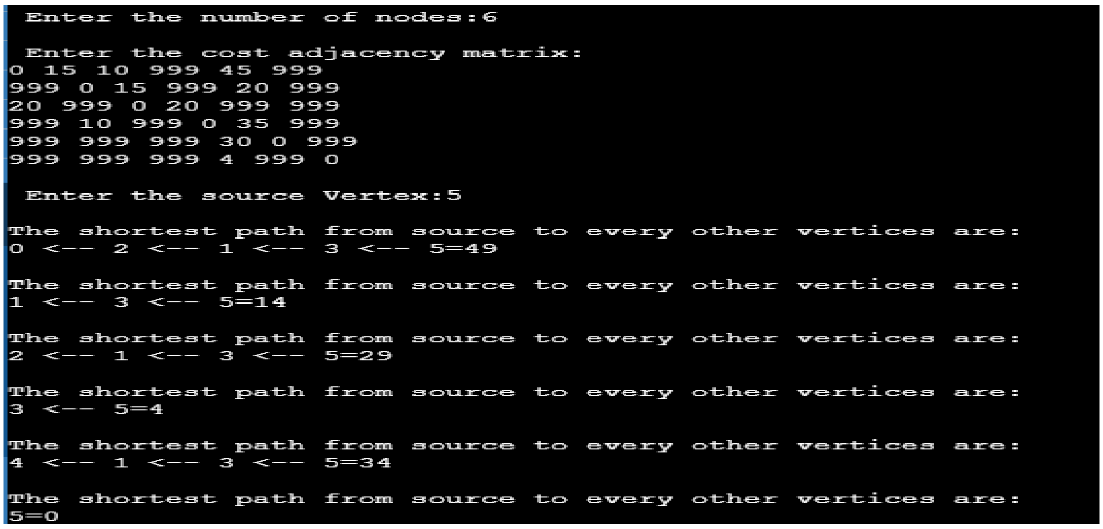
```
    for(j=0;j<n;j++)
        {
        if(d[j]<min &&S[j]==0)
        {
            min=d[j];      i think this min is not needed
            u=j;
        }
    }
if(u==-1)
    return;
S[u]=1;
if(u==dest)
    return;
for(v=0;v<n;v++)
    {
        if((d[u]+cost[u][v]<d[v]) &&S[v]==0)
        {
        d[v]=d[u]+cost[u][v];
        p[v]=u;
        }
    }
    }
}


void print_path(int source)
{
int destination,i;
for(destination=0;destination<n; destination++)
    {
    dij(source,destination);
        if(d[destination]==999)
            printf("\n %d is not reachable from %d",destination,source);
        else
            {
            printf("\nThe shortest path from source to every other vertices are:\n");
            i=destination;
        while(i!=source)
        {
        printf("%d <-- ",i);
        i=p[i];
        }
printf("%d=%d\n",i, d[destination]);
            }
    }
}
```

```
void main()
{
int i,j,source;
//clrscr();
printf("\n Enter the number of nodes:");
scanf("%d",&n);
printf("\n Enter the cost adjacency matrix:\n");
for(i=1;i<=n;i++)
    {
    for(j=1;j<=n;j++)
        {
        scanf("%d",&cost[i][j]);
        }
    }
printf("\n Enter the source Vertex:");
scanf("%d",&source);
print_path(source);
getch();
}
```

Input Graph:


$$\begin{tikzcd}
 & & \\
0 \arrow[r, "15"] \arrow[d, bend left, "10"] \arrow[rr, bend left, "45"] & 1 \arrow[r, "20"] \arrow[dl, "15"] & 4 \arrow[dl, bend left, "30"] \\
2 \arrow[u, bend left, "20"] \arrow[r, "20"] & 3 \arrow[u, "10"] \arrow[ur, bend right, "35"] & 5 \arrow[l, "4"]
\end{tikzcd}$$

Output


```
 Enter the number of nodes:6

 Enter the cost adjacency matrix:
0 15 10 999 45 999
999 0 15 999 20 999
20 999 0 20 999 999
999 10 999 0 35 999
999 999 999 30 0 999
999 999 999 4 999 0

 Enter the source Vertex:5

The shortest path from source to every other vertices are:
0 <-- 2 <-- 1 <-- 3 <-- 5=49

The shortest path from source to every other vertices are:
1 <-- 3 <-- 5=14

The shortest path from source to every other vertices are:
2 <-- 1 <-- 3 <-- 5=29

The shortest path from source to every other vertices are:
3 <-- 5=4

The shortest path from source to every other vertices are:
4 <-- 1 <-- 3 <-- 5=34

The shortest path from source to every other vertices are:
5=0
```

Problem 9:

Find minimum cost spanning tree of a given undirected graph using Kruskal’s Algorithm.

Method: Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weightedgraph.It finds a subset of the edges that forms a tree that includes every vertex,where the total weight of all the edges in the tree is minimized.
Let G=(V,E) be a directed graph. The algorithm selects n-1 edges one at a time. The algorithm selects the least cost edge (u,v) from E and delete the edge (u,v) from E. Then it checks whether the selected edge results in cycle when added to the list of edges to form MST. If the selected edge (u, v) does not form cycles, then it is added to the list of edges to form MST else the edge is neglected. This process is repeated until n-1 edges are selected.

Two functions which are essential for the design of Kruskal’s algorithm are:

1. **find(u):** if u is a node in a tree, this function returns the root of the corresponding tree.

2. **Union_ij(i, j):** let C1 and C2 be two trees. If i is a vertex in tree C1 and j is a vertex in the tree C2, this function merges the two trees C1 and C2 into single tree.

Algorithm

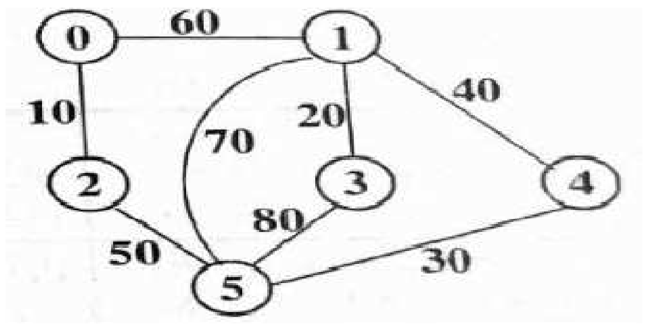
```
1. Arrange the edges of the given graph in non-increasing order of their weights
2. Create a forest with n vertices
3. Select an edge(u,v) with least cost
4. Find the roots of vertex u and vertex v
5. If the roots of vertex u and vertex v are different then
       Select the edge (u, v) as the edge of MST
       Update the number of edges selected for MST
       Update the cost of MST
       Merge the two trees with roots of vertex u and vertex v
6. Delete the edge (u, v) from the list
7. Repeat steps 3 through 6 until n-1 edges are selected
8. Finally print the edges selected for MST along the cost
```

Program

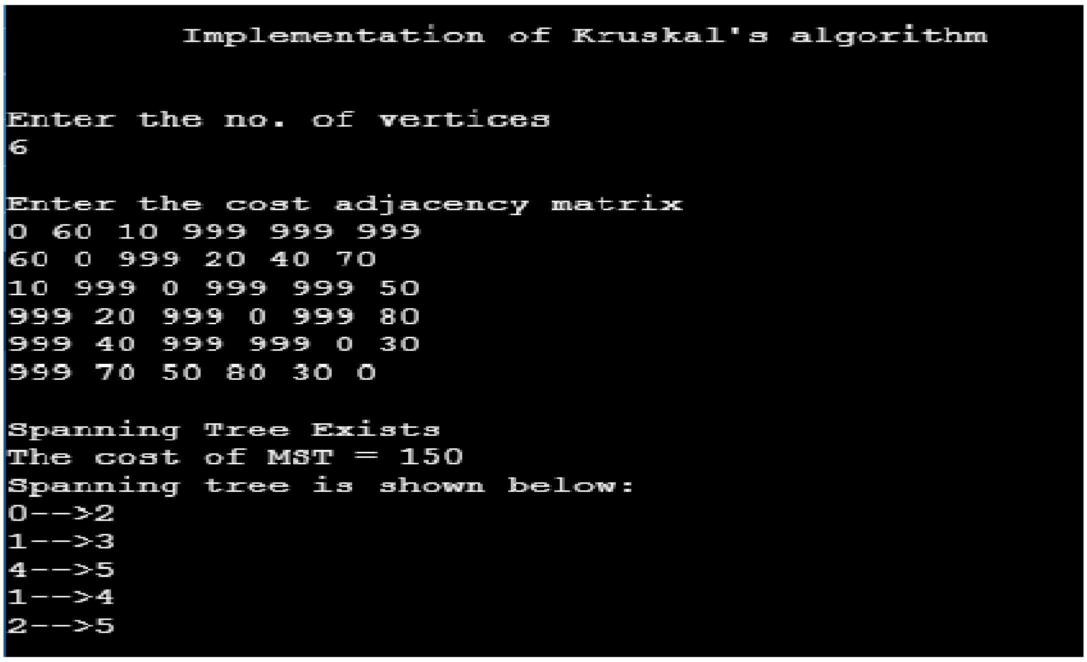
```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int cost[10][10],n;

int find(int node, int parent[10])
{
while(parent[node]!=node)
        node=parent[node];
return node;
```

```
}


int union_ij(int i,int j,int parent[10])
{
if(i<j)
        parent[j]=i;
else
        parent[i]=j;
}

void kruskal_mst()
{
int count, i, parent[10], min, u, v, k, t[10][2], sum;
count=0,sum=0,k=0;
/* create a forest with n vertices*/
for(i=0;i<n;i++)
        parent[i]=i;
/* repeat until n-1 edges are selected*/
while(count<n)
{
/* select an edge (u, v) with least cost*/
        min=999;
                for(i=0;i<n;i++)
                {
                for(j=0;j<n;j++)
                        {
                        if(cost[i][j]<min)
                                {
                                min=cost[i][j];
                                u=i;
                                v=j;
                                }
                        }
                }
if(min==999)break;
/* find the roots of vertex u and vertex v*/
        i=find(u,parent);
        j=find(v,parent);
/*if the roots of vertex u and vertex v are different then*/
if(i!=j)
{
/*select the edge (u, v) as the edge of MST*/
t[k][0]=u;
t[k][1]=v;
k++;
/* update the number of edges selected for MST*/
count++;
/* update the cost of MST*/
sum=sum+min;
/* merge the two trees with roots of vertex u and vertex v*/
unioun_ij(i,j,parent);
}
/*delete the edge (u, v) from the list*/
cost[u][v]=cost[v][u]=999;
}
if(count==n-1)
        {
        printf("\nSpanning Tree Exists");
        printf("\nThe cost of MST = %d",sum);
        printf("\nSpanning tree is shown below:");
        for(k=0;k<n-1;k++)
                printf("\n%d-->%d",t[k][0],t[k][1]);
        return;
        }
else
        printf("\nSpanning tree do not exist");
}

void main()
{
int i, j;
clrscr();
printf("\n\n\t Implementation of Kruskal's algorithm\n\n");
printf("\nEnter the no. of vertices\n");
scanf("%d",&n);
printf("\n Enter the cost 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;
                }
        }
kruskals_mst();
}
```

Input Graph



Output:



```
          Implementation of Kruskal's algorithm


Enter the no. of vertices
6

Enter the cost adjacency matrix
0 60 10 999 999 999
60 0 999 20 40 70
10 999 0 999 999 50
999 20 999 0 999 80
999 40 999 999 0 30
999 70 50 80 30 0

Spanning Tree Exists
The cost of MST = 150
Spanning tree is shown below:
0-->2
1-->3
4-->5
1-->4
2-->5
```

Program 10:

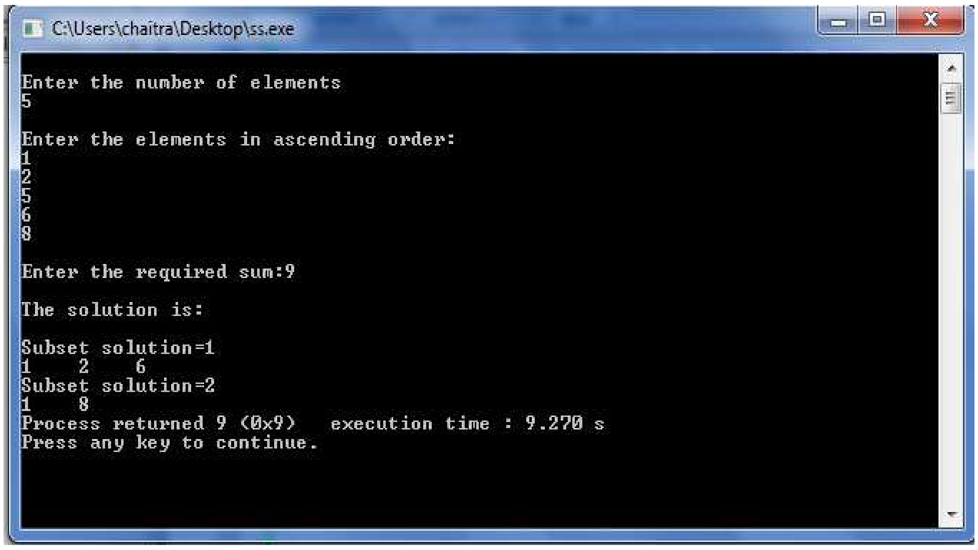
Implement Sum-of-Subset problem of a given set S = {s_1, s_2, ..., s_n} of ‘n’ Positive integers whose sum is equal to a given positive integer ‘d’.

In the subset sum problem, we have to find the subset of a set is such a way that the element of this subset-sum up to a given number K. All the elements of the set are positive and unique (no duplicate elements are present). For this, we will create subsets and check if their sum is equal to the given number k.
For example if S={1,2,5,6,8} and d=9,there are two solutions {1,2,6} and {1,8}.

Program:

```
#include<stdio.h>
int count=0;
int w[10];
int d;
int x[10];

void subset(int cs,int k,int r)
{
    int i;
    x[k]=1;
    if(cs+w[k]==d)
    {
        printf("\nSubset solution=%d\n",++count);
        for(i=0;i<=k;i++)
        {
            if(x[i]==1)
                printf("%d    ",w[i]);
        }
    }
    else if(cs+w[k]+w[k+1]<=d)
        subset(cs+w[k],k+1,r-w[k]);
    if((cs+r-w[k]>=d)&&(cs+w[k+1]<=d))
    {
        x[k]=0;
        subset(cs,k+1,r-w[k]);
    }
}

void main()
{

    int sum=0,i,n;
    printf("\nEnter the number of elements\n");
    scanf("%d",&n);
```

```
   printf("\nEnter the elements in ascending order:\n");
   for(i=0;i<n;i++)
      scanf("%d",&w[i]);

   printf("\nEnter the required sum:");
   scanf("%d",&d);

   for(i=0;i<n;i++)
         sum+=w[i];

   if(sum<d)
   {
      printf("\nNo solution exists\n");
      return;
   }
   printf("\nThe solution is:\n");
   subset(0,0,sum);
}
```



```
Enter the number of elements
5

Enter the elements in ascending order:
1
2
5
6
8

Enter the required sum:9

The solution is:

Subset solution=1
1    2    6
Subset solution=2
1    8
Process returned 9 (0x9)   execution time : 9.270 s
Press any key to continue.
```