VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY
Permanently Affiliated to JNTU Kakinada, Approved by AICTE
Accredited by NAAC with 'A' Grade, ISO 9001:2008 Certified
Nambur, Pedakakani (M), Guntur (Dt) - 522508
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
B.Tech Program is Accredited by NBA

Design and Analysis of Algorithms Lab Manual					
Course Code:	Year and Semester: III-I	L	T	P	C
Prerequisites: Prior knowledge of programming language(s) and basic Data Structures and Algorithms		0	0	3	1.5

Course Objectives:

1. To learn fundamental algorithmic problems.
2. To understand methods of designing and analysing algorithms.
3. To know various designing paradigms of algorithms for solving real world problems.

Course Outcomes:

At the end of the course student will be able to:

CO1: Identify and apply the suitable algorithm for the given problem.

CO2: Design and implement efficient algorithms for a specified application.

List of experiments:

1. Write a program to find the maximum and minimum element from the collection of elements using divide and conquer technique.

2. Write a program to find the optimal profit of a Knapsack using Greedy method.

3. Write a program for Optimal Merge Patterns problem using Greedy Method.

4. Write a program for Single Source Shortest Path for General Weights using Dynamic Programming.

5. Write a program to find all pair shortest path from any node to any other node within a graph.

6. Write a program to find the non-attacking positions of Queens in a given chess board using backtracking.

7. Find a subset of a given set S = {S_l, S_2, ..., S_n} 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.

8. Write a program to color the nodes in a given graph such that no two adjacent can have the same color using backtracking.

9. Design and implement to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using Backtracking principle.

Eperiment 1:

Write a program to find the maximum and minimum element from the collection of elements using divide and conquer technique.

A divide-and-conquer algorithm for this problem would proceed as follows:

Let P = (n,a[i],….,a[j]) denote an arbitrary instance of the problem. Here n is the number of elements in the list a[i],….,a[j] and we are interested in finding the maximum and minimum of this list.

Let small(P) be true when n ≤ 2.

If n = 1, the maximum and minimum are a[i].

If n = 2, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances.

For example, we might divide P into the two instances

P1 = (n/2,a[1],….,a[n/2]) and P2 = (n - n/2,a[n/2 + 1],….,a[n]).

After having divided P into two smaller sub problems, we can solve them by recursively invoking the same divide and conquer algorithm.

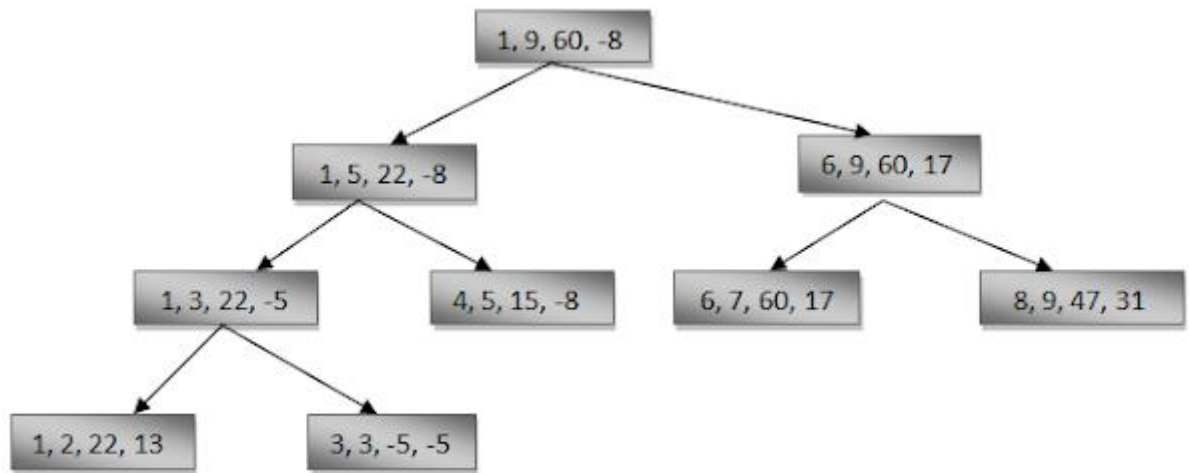We can combine the Solutions for P1 and P2 to obtain the solution for P as follows.

If MAX(P) and MIN(P) are the maximum and minimum of the elements of P, then MAX(P) is the larger of MAX(P1) and MAX(P2) also MIN(P) is the smaller of MIN(P1) and MIN(P2).

MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements {a(i),a(i+1),…,a(j)}. The situation of set sizes one (i=j) and two (i=j-1) are handled separately. For sets containing more than two elements, the midpoint is determined and two new sub problems are generated. When the maxima and minima of these sub problems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

Suppose we simulate MaxMin on the following nine elements:

Ex: a: [1] [2] [3] [4] [5] [6] [7] [8] [9]
       22   13   -5   -8   15   60   17   31   47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. On the array a[ ] above, the following tree is produced.

1, 9, 60, -8

1, 5, 22, -8

6, 9, 60, 17

1, 3, 22, -5

4, 5, 15, -8

6, 7, 60, 17

8, 9, 47, 31

1, 2, 22, 13

3, 3, -5, -5

We see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new call to MaxMin, where i and j have the values 1, 5 and 6, 9, and thus split the set into two subsets of the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call).

Algorithm:

```
Alogrithm MaxMin(i, j, max, min)
// a[1:n] is a global array. Parameters i and j are integers,
 // 1≤i≤j≤n. The effect is to set max and min to the largest and    // smallest values
in a[i:j].
{
        if (i=j) then max := min := a[i]; //Small(P)
        else if (i=j-1) then // Another case of Small(P)
            {
                    if (a[i] < a[j]) then max := a[j]; min := a[i];
                    else max := a[i]; min := a[j];
            }
        else
        {
                // if P is not small, divide P into sub-problems.
                // Find where to split the set.
                mid := ( i + j )/2;
                // Solve the sub-problems.
                MaxMin( i, mid, max, min );
                MaxMin( mid+1, j, max1, min1 );
                // Combine the solutions.
                if (max < max1) then max := max1;
                if (min > min1) then min := min1;
        }
```

```
 }
```

```c
#include<stdio.h>
int max,min;
int a[100];
void maxmin(int i,int j)
{
 int max1,min1,mid;
 if(i==j)
 {
  max=min=a[i];
 }
 else
 {
  if(i==j-1)
  {
   if(a[i]<a[j])
   {
    max=a[j];
    min=a[i];
   }
   else
   {
    max= a[i];
    min=a[j];
```

```c
        }
    }
    else
    {
      mid=(i+j)/2;
      maxmin(i,mid);
      max1=max; min1=min;
      maxmin(mid+1,j);
      if(max<max1)
        max=max1;
      if(min>min1)
        min=min1;
    }
  }
}
void main ()
{
 int i,s;
 printf("\n Enter size of input....");
 scanf("%d",&s);
 printf("Enter the numbers : \n");
 for(i=1;i<=s;i++)
  scanf("%d",&a[i]);
 max =-1;
```

```c
    min =9999;

  maxmin(1,s);

  printf ("maximum element is: %d\n",max);

  printf ("minimum element is: %d\n",min);

  }
```

Output:

```
 Enter size of input....5
Enter the numbers :
12
56
2
45
67
maximum element is: 67
minimum element is: 2
```

Experiment 2:
    Write a program to find the optimal profit of a Knapsack using Greedy method.

Procedure:
Greedy Method:
The greedy method is the most straight forward design technique, used to determine a feasible solution that may or may not be optimal.

Feasible solution:- Most problems have n inputs and its solution contains a subset of inputs that satisfies a given constraint(condition). Any subset that satisfies the constraint is called feasible solution.

Optimal solution: To find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution.

The greedy method suggests that an algorithm works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution.

Knapsack problem

The knapsack problem or rucksack (bag) problem is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible
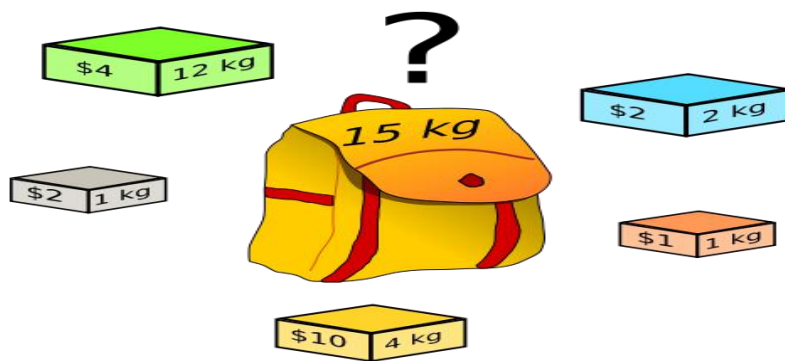
There are two versions of the problems

1. 0/1 knapsack problem
2. Fractional Knapsack problem
    a. Bounded Knapsack problem.
    b. Unbounded Knapsack problem.

Solutions to knapsack problems

Brute-force approach:-Solve the problem with a straight forward algorithm

Greedy Algorithm:- Keep taking most valuable items until maximum weight is reached or taking the largest value of each item by calculating $v_i = value_i/Size_i$

Fractional knapsack problem is solved using greedy method as follows

1. For each item, compute its value / weight ratio.
2. Arrange all the items in decreasing order of their value / weight ratio.
3. Start putting the items into the knapsack beginning from the item with the highest ratio.

   Put as many items as you can into the knapsack.

Example
Consider the following instance of the knapsack problem: n = 3, m = 20, (p1, p2, p3) = (25, 24, 15) and (w1, w2, w3) = (18, 15, 10).

1. First, we try to fill the knapsack by selecting the objects in some order:

x1	x2	x3	wi xi	pi xi
1/2	1/3	1/4	18 x 1/2 + 15 x 1/3 + 10 x 1/4 = 16.5	25 x 1/2 + 24 x 1/3 + 15 x 1/4 = 24.25

2. Select the object with the maximum profit first (p = 25). So, x1 = 1 and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit (p = 24). So, x2 = 2/15

x1	x2	x3	wi xi	pi xi
1	2/15	0	18 x 1 + 15 x 2/15 = 20	25 x 1 + 24 x 2/15 = 28.2

3. Considering the objects in the order of non-decreasing weights wi.

x1	x2	x3	wi xi	pi xi
0	2/3	1	15 x 2/3 + 10 x 1 = 20	24 x 2/3 + 15 x 1 = 31

4. Considered the objects in the order of the ratio pi / wi .

p1/w1	p2/w2	p3/w3
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio pi / xi. Select the object with the maximum pi / xi ratio, so, x2 = 1 and profit earned is 24. Now, only 5 units of space is left, select the object with next largest pi / xi ratio, so x3 = ½ and the profit earned is 7.5.

x1	x2	x3	wi xi	pi xi
0	1	1/2	15 x 1 + 10 x 1/2 = 20	24 x 1 + 15 x 1/2 = 31.5

optimal solution.

Algorithm
If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.
Algorithm GreedyKnapsack (m, n)
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so that p[i] / w[i] > p[i + 1] / w[i + 1].
// m is the knapsack size and x[1: n] is the solution vector.
{
for i := 1 to n do x[i] := 0.0 // initialize x U := m;
for i := 1 to n do
{
if (w(i) > U) then break;
x [i] := 1.0; U := U – w[i];
}
if (i < n) then x[i] := U / w[i];
}

```c
# include<stdio.h>
void knapsack(int n,float weight[],float profit[],float capacity)
{
    float x[20],pr=0;
    int i,j,u;
    u=capacity;
    for(i=0;i<n;i++)
        x[i]=0.0;
    for(i=0;i<n;i++)
    {
        if(weight[i]>u)
            break;
        else
         {
            x[i]=1.0;
            pr=pr+profit[i];
            u=u-weight[i];
         }
}
```

```c
    if(i<n)
        x[i]=u/weight[i];
pr=pr+(x[i]*profit[i]);
  printf("\n Result array x is...");
for(i=0;i<n;i++)
  printf("%f\t",x[i]);
printf("\n Profit is..... %f",pr);
}

void main()
{
    float weight[20],profit[20],capacity;
    int n,i,j;
    float ratio[20],t;
    printf("\n enter the capacityacity of knapsack...");
    scanf("%f",&capacity);
    printf("\n Toal no of objects...");
    scanf("%d",&n);
    printf("\n Enter the weights and profits of each object...");
    for (i=0;i<n;i++)
    {
        scanf("%f %f",&weight[i],&profit[i]);
    }
  for (i=0;i<n;i++)
   {
       ratio[i]=profit[i]/weight[i];
   }
for (i=0;i<n;i++)
{
    for (j=i + 1;j<n;j++)
     {
        if (ratio[i]<ratio[j])
         {
            t=ratio[j];
            ratio[j]=ratio[i];
            ratio[i]=t;
            t=weight[j];
            weight[j]=weight[i];
            weight[i]=t;
            t=profit[j];
```

```
                profit[j]=profit[i];
                profit[i]=t;
            }
        }
    }
}
knapsack(n,weight,profit,capacity);
}
```

```
 enter the capacityacity of knapsack...20

 Toal no of objects...3

 Enter the weights and profits of each object...18
25
15 24
10 15

 Result array x is...1.000000   0.500000        0.000000
 Profit is..... 31.500000
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY
Permanently Affiliated to JNTU Kakinada, Approved by AICTE
Accredited by NAAC with 'A' Grade, ISO 9001:2008 Certified
Nambur, Pedakakani (M), Guntur (Dt) - 522508
VASIREDDY VENKATADRI
INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
B.Tech Program is Accredited by NBA

Experiment 3:

Write a program for Optimal Merge Patterns problem using Greedy Method

❖ Given 'n' sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge 'n' sorted files together.

❖ This type of merging is called as 2-way merge patterns.

❖ To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice choice is, at each step merge the two smallest files together.

❖ The two-way merge patterns can be represented by binary merge trees.

The formula of external merging cost is:

$$\sum_{i=1}^{n} f(i)d(i)$$

Where, f (i) represents the number of records in each file and d (i) represents the depth.

Algorithm to Generate Two-way Merge Tree:

struct treenode
{
treenode *lchild;
treenode *rchild;
};

Algorithm TREE (n)

// list is a global of n single node binary trees
{
for i := 1 to n – 1 do
{
pt new treenode
(pt ->lchild)= least (list); // merge two trees with smallest lengths
(pt->rchild) least (list);
(pt-> weight)=((pt-> lchild)->weight)+((pt->rchild)->weight);
insert (list,pt);
}
return least (list);
}

❖ An optimal merge pattern corresponds to a binary merge tree with minimum weighted external path length.

❖ The function tree algorithm uses the greedy rule to get a two- way merge tree for n files. The algorithm contains an input list of n trees.

❖ There are three field child, rchild, and weight in each node of the tree. Initially, each tree in a list contains just one node.

❖ This external node has lchild and rchild field zero whereas weight is the length of one of the n files to be merged.

- For any tree in the list with root node t, t = it represents the weight that gives the length of the merged file.
- There are two functions least (list) and insert (list, t) in a function tree.
- Least (list) obtains a tree in lists whose root has the least weight and return a pointer to this tree.
- This tree is deleted from the list.
- Function insert (list, t) inserts the tree with root t into the list.
- The main for loop in this algorithm is executed in n-1 times.
- If the list is kept in increasing order according to the weight value in the roots, then least (list) needs only O(1) time and insert (list, t) can be performed in O(n) time. Hence, the total time taken is $O(n^2)$.
- If the list is represented as a minheap in which the root value is less than or equal to the values of its children, then least (list) and insert (list, t) can be done in O (log n) time. In this condition, the computing time for the tree is O (n log n).

Example:

Suppose we are having three sorted files X_1, X_2 and X_3 of length 30, 20, and 10 records each. Merging of the files can be carried out as follows:

| S.No	First Merging	Record moves in first merging	Second merging	Record moves in second merging	Total no. of records moves
1.	X_1 & X_2 = T1	50	T_1 & X_3	60	50 + 60 = 110
2.	X_2 & X_3 = T1	30	T_1 & X_1	60	30 + 60 = 90

The Second case is optimal.

Example 2:

Given five files (X1, X2, X3, X4, X5) with sizes (20, 30, 10, 5, 30). Apply greedy rule to find optimal way of pair wise merging to give an optimal solution using binary merge tree representation.
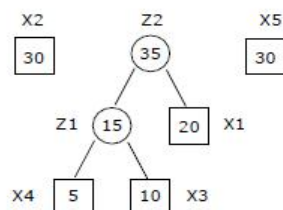
Solution:
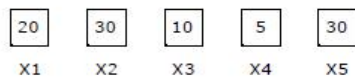
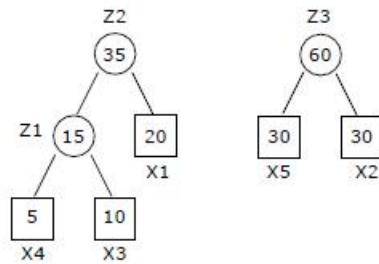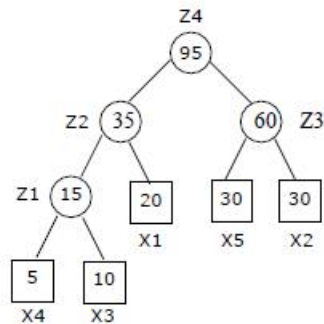Merge X_4 and X_3 to get 15 record moves. Call this Z_1.
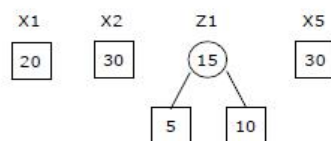
Merge Z_1 and X_1 to get 35 record moves. Call this Z_2.

Merge X_2 and X_5 to get 60 record moves. Call this Z_3.

Merge Z_2 and Z_3 to get 90 record moves. This is the answer. Call this Z_4.

Therefore the total number of record moves is $15 + 35 + 60 + 95 = 205$. This is an optimal merge pattern for the given problem.

Sample implementation

```c
//programme to generate optimal merge patterns
#include <stdio.h>
#include <stdlib.h>
int ops;
struct node
{
int weight;
struct node *left,*right;
};
//min heap for storing nodes
struct heapn
{
    int size;
    int capacity;
    struct node** array;
};
//function to crete new node
struct node* newnode(int weight)
{
    struct node* temp = (struct node*)malloc(
      sizeof(struct node));
    temp->left = temp->right = NULL;
    temp->weight = weight;
    return temp;
```

```c
}

// function to create a minimum heap
struct heapn* createminheap(int capacity)
{
    struct heapn* minheap= (struct heapn*)malloc(sizeof(struct heapn));
    minheap->size = 0;
    minheap->capacity = capacity;
    minheap->array = (struct node**)malloc(
        minheap->capacity * sizeof(struct node*));
    return minheap;
}
void swapnode(struct node** n1,struct node** n2)
{
    struct node *t=*n1;
    *n1=*n2;
    *n2=t;
}

//function to heapify
void heapify(struct heapn* minheap, int idx)
{
    int min=idx;
    int left=2*idx+1;
    int right=2* idx+ 2;
    if (left<minheap->size&&minheap->array[left]->weight
            < minheap->array[min]->weight)
        min=left;
    if (right<minheap->size
        && minheap->array[right]->weight
            < minheap->array[min]->weight)
        min=right;
    if (min!=idx) {
        swapnode(&minheap->array[min],
                &minheap->array[idx]);
        heapify(minheap,min);
    }
}
//extract minimum node
struct node* extractmin(struct heapn* minheap)
{
    struct node* temp = minheap->array[0];
    minheap->array[0] = minheap->array[minheap->size - 1];
    --minheap->size;
    heapify(minheap,0);
    return temp;
}

// insert new node to Min Heap
void insert(struct heapn* minheap,struct node* minh)
```

```c
{
    ++minheap->size;
    int i=minheap->size-1;
    while (i&&minh->weight< minheap->array[(i-1)/2]->weight)
    {
        minheap->array[i]=minheap->array[(i-1)/2];
        i=(i-1)/2;
    }
    minheap->array[i]=minh;
}

//function to initialize heap
void buildheap(struct heapn* minheap)
{
    int n=minheap->size-1;
    int i;
    for(i=(n-1)/2;i>=0;--i)
        heapify(minheap,i);
}

// Creates a min heap of capacity
struct heapn* create(int d[],int size)
{
    struct heapn* minheap=createminheap(size);
    for(int i=0;i<size;++i)
        minheap->array[i]=newnode(d[i]);
    minheap->size=size;
    buildheap(minheap);
    return minheap;
}
// The main function that builds Huffman tree
struct node* optt(int d[],int size)
{
    struct node *left,*right,*top;
    struct heapn* minheap= create(d,size);
    int i=0,t,tot=0;
    printf("\n");
    while (!(minheap->size==1))
    {
        left=extractmin(minheap);
        right=extractmin(minheap);
        t=left->weight+right->weight;
        tot=tot+t;
        printf("\n merge %d and %d cost is =%d",left->weight,right->weight,t);
        top=newnode(left->weight+right->weight);
        top->left=left;
        top->right=right;
        insert(minheap,top);
    }
    printf("\n total merge cost is...:%d",tot);
```

```
    return extractmin(minheap);
}

void main()
{
    int fr[10],n,i;
    printf("\n Enter the no of files:");
    scanf("%d",&n);
    printf("enter the records of %d file",n);
    for(i=0;i<n;i++)
    scanf("\n %d",&fr[i]);
    struct node* root=optt(fr,n);
}
```

Output:
Enter the no of files:4
enter the records of 4 file20
10
15
5

  merge 5 and 10 cost is =15
  merge 15 and 15 cost is =30
  merge 20 and 30 cost is =50
  total merge cost is...:95

Experiment 4:
Write a program for Single Source Shortest Path for General Weights using Dynamic Programming
❖ Bellman Ford algorithm helps to find the shortest path from a vertex to all other vertices of a weighted graph.
❖ Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

Procedure:
❖ Maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.
❖ We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.
❖ Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

Algorithm:
Algorithm bellmanFord(G,S)
   for each vertex V in G
       distance[V] <- infinite
```

```
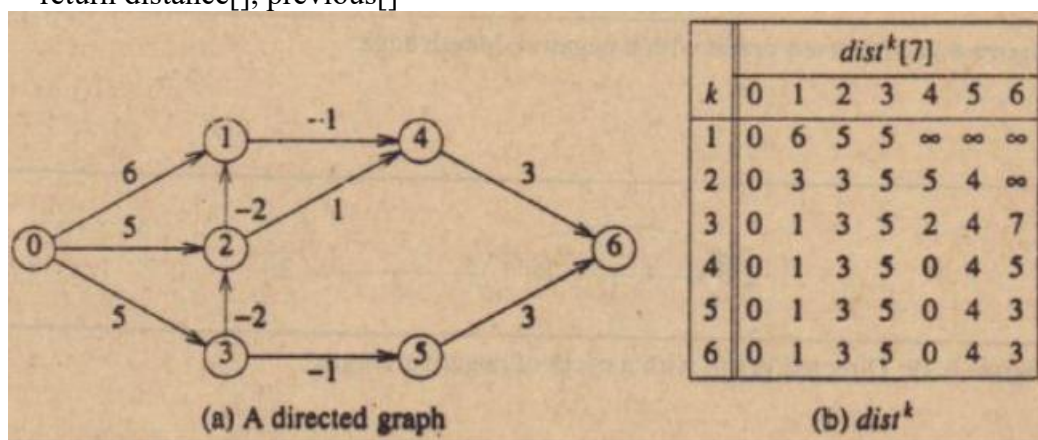            previous[V] <- NULL
        distance[S] <- 0

    for each vertex V in G
        for each edge (U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    for each edge (U,V) in G
        If distance[U] + edge_weight(U, V) < distance[V}
            Error: Negative Cycle Exists

    return distance[], previous[]
```


(a) A directed graph   (b) $dist^k$

Sample implementation:

```c
#include <stdio.h>
#include <stdlib.h>
#define INF 9999
struct edge
{
    int u;
    int v;
    int w;
};
struct graph
{
    int vertices;
    int edges;
    struct edge *edge;
};
void bellmanford(struct graph *g, int source);
void display(int arr[], int size);
int main(void)
{
    int v,e,i;
    //create graph
    struct graph *g = (struct graph *)malloc(sizeof(struct graph));
```

```c
    printf("\n enter the total vertices and edges:");
    scanf("%d%d",&v,&e);
    g->vertices=v;
    g->edges=e;
g->edge=(struct edge *)malloc(g->edges*sizeof(struct edge));
printf("\n enter edge u,v and weight w..:");
for(i=0;i<e;i++)
{
 scanf("%d%d%d",&g->edge[i].u,&g->edge[i].v,&g->edge[i].w);
}
bellmanford(g,0);
return 0;
}
void bellmanford(struct graph *g, int source)
{
    int i, j, u, v, w;
    int ve = g->vertices; //total no of vertices
    int ee = g->edges; // total no of edges
    int d[ve];
    int p[ve];     //predecessor array
    //initialize distance and predessor array
    for(i=0;i<ve;i++)
    {
      d[i]=INF;
      p[i]=0;
    }
    d[source] = 0;
    for(i=1; i<=ve-1;i++) // relax edges
    {
      for(j=0;j<ee;j++)
       {
         u=g->edge[j].u;
         v=g->edge[j].v;
         w=g->edge[j].w;
         if(d[u]!=INF&&d[v]>d[u]+w)
          {
            d[v]=d[u]+w;
            p[v]=u;
          }
       }
    }
    for(i=0;i<ee;i++)     // check for negative cycle
    {
      u=g->edge[i].u;
      v=g->edge[i].v;
      w=g->edge[i].w;
      if (d[u]!=INF&&d[v]>d[u]+w)
       {
         printf("\n Negative Cycle...:");
         return;
```

```
    }
  }
  printf("distance array: ");
  display(d,ve);
  printf("predecessor array: ");
  display(p,ve);
}
void display(int arr[], int size)
{
  int i;
  for(i=0;i<size;i++)
  {
    printf("\n %d",arr[i]);
  }
  printf("\n");
}
```

Ouput:

```
enter the total vertices and edges:7
10

 enter edge u,v and weight w..:0 1 6
0 2 5
0 3 5
1 4 -1
2 1 -2
3 2 -2
2 4 1
3 5 -1
4 6 3
5 6 3
distance array:
 0  1  3  5  0  4  3
predecessor array:
 0  2  3  0  1  3  4
```

Experiment 5:
Write a program to find all pair shortest path from any node to any other node within a graph

❖ Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph.
❖ This algorithm works for both the directed and undirected weighted graphs.
❖ It does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).
❖ This algorithm follows the dynamic programming approach to find the shortest paths.

<u>Algorithm:</u>
Algorithm Floydwarshal()
{
//Initialize the cost matrix
for i := 1 to N
    for j := 1 to N
        if there is an edge from i to j
            dist[0][i][j]:= the length of the edge from i to j
        else
            dist[0][i][j] := INFINITY

for k:= 1 to N
    for I:= 1 to N
        for j:= 1 to N
            dist[k][i][j]:= min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j])
}

❖ Initialize the cost matrix same as the input graph matrix.
❖ update the solution matrix by considering all vertices as an intermediate vertex.
❖ consider vertex k as an intermediate vertex.
❖ For every pair (i, j) of the source and destination vertices respectively.
1) k is not an intermediate vertex in shortest path from i to j. no change in the value of dist[i][j].
2) k is an intermediate vertex in shortest path from i to j.
    Update the value of dist[i][j] as
    dist[k][i][j]:= min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j])

<u>Sample implementation:</u>

```
#include<stdio.h>
#include<conio.h>
int min(int,int);
void floyds(int c[10][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++)
          if(i==j)
```

```c
            c[i][j]=0;
            else
            c[i][j]=min(c[i][j],c[i][k]+c[k][j]);
}
int min(int a,int b)
{
    if(a<b)
     return a;
     else
     return b;
}
void main()
{
    int c[10][10],w,n,e,u,v,i,j;
    printf("\n enter the number of vertices:");
    scanf("%d",&n);
    printf("\n enter the number of edges:\n");
    scanf("%d",&e);
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=n;j++)
            c[i][j]=999;
    }
    printf("\n enter the edge,weight for %d edges \n",e);
    for (i=1;i<=e;i++)
    {
        scanf("%d%d%d",&u,&v,&w);
        c[u][v]=w;
    }
    printf("\n Matrix of input data:\n");
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=n;j++)
            printf("%d \t",c[i][j]);
        printf("\n");
    }
    floyds(c,n);
    printf("\n cost matrix:\n");
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=n;j++)
            printf("%d \t",c[i][j]);
        printf("\n");
    }
    printf("\n shortest paths are:\n");
    for (i=1;i<=n;i++)
      for (j=1;j<=n;j++)
      {
        if(i!=j)
         printf("\nshortest path from %d to %d is %d",i,j,c[i][j]);
```

```
        }
        getch();
    }
```

Output:

```
 enter the number of vertices:4

 enter the number of edges:
6

 enter the edge,weight for 6 edges
1 2 3
2 1 2
1 4 5
2 4 4
3 2 1
4 3 2

 Matrix of input data:
999    3     999    5
2      999   999    4
999    1     999    999
999    999   2      999

 cost matrix:
0      3     7     5
2      0     6     4
3      1     0     5
5      3     2     0

 shortest paths are:

shortest path from 1 to 2 is 3
shortest path from 1 to 3 is 7
shortest path from 1 to 4 is 5
shortest path from 2 to 1 is 2
shortest path from 2 to 3 is 6
shortest path from 2 to 4 is 4
shortest path from 3 to 1 is 3
shortest path from 3 to 2 is 1
shortest path from 3 to 4 is 5
shortest path from 4 to 1 is 5
shortest path from 4 to 2 is 3
shortest path from 4 to 3 is 2
```