List of Experiments

Sl. No.	Name of the Practical Assignment
1	Implement Binary Search using Divide and Conquer approach
2	Implement Merge Sort using Divide and Conquer approach
3	Implement Quick Sort using Divide and Conquer approach
4	Find Maximum and Minimum element from a array of integer using Divide and Conquer approach
5	Find the minimum number of scalar multiplication needed for chain of matrix
6	Implement all pair of Shortest path for a graph (Floyd-War shall Algorithm)
7	Implement Single Source shortest Path for a graph (Bellman Ford Algorithm)
8	Implement Single Source shortest Path for a graph (Dijkstra Algorithm)
9	Implement 8 Queen problem
10	Implement Travelling Salesman problem
11	Implement Graph Coloring problem
12	Implement Job sequencing with deadlines problem
13	Implement Longest Common Subsequence problem

1. Implement Binary Search using Divide and Conquer approach

Objective:

```
#include<stdio.h>
#incliude<conio.h>

int Binary_search(int arr[],int l,int u, int SrchElmnt)
 {
     if(l<=u)
     {
      int mid = (l+ u)/2;

      if(arr[mid]==SrchElmnt)
       return 1;

       else if(arr[mid]>SrchElmnt)
        return Binary_search(arr,l,mid-1,SrchElmnt);

        else
          return Binary_search(arr,mid+1,u,SrchElmnt);
   }

   return 0;
}

main()
{
 int arr[15]={3,4,5,6,7,8,4,5,2,8,9,13,14,17,19};

 int n=15;

  int S=79;

  if(Binary_search(arr,0,n-1,S)==1)
        printf("found");
  else
        printf("not found");
}
```

2. Implement Merge Sort using Divide and Conquer approach

```
#include<stdio.h>

void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

int main()
{
        int a[n0],n,i;
        printf("Enter no of elements:");
        scanf("%d",&n);
        printf("Enter array elements:");

        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        mergesort(a,0,n-1);

        printf("\nSorted array is :");
        for(i=0;i<n;i++)
                printf("%d ",a[i]);

        return 0;
}


void mergesort(int a[],int i,int j)
{
        int mid;
        if(i<j)
        {
                mid=(i+j)/2;
                mergesort(a,i,mid);          //left recursion
                mergesort(a,mid+1,j);        //right recursion
                merge(a,i,mid,mid+1,j);   //merging of two sorted sub-arrays
        }
}
```

```
void merge(int a[],int i1,int j1,int i2,int j2)
/*As we are using one array instead of taking 2 different array, one array, a[] is
taken, but a[] had to be divided into two part from the index i1 to j1 and from the
index i2 to j2 */
{
        int temp[50];                   //array used for merging
        int i,j,k;
        i=i1;                           //beginning of the first list
        j=i2;                           //beginning of the second list
        k=0;

        while(i<=j1 && j<=j2)   //while elements in both lists
        {
                if(a[i]<a[j])
                        temp[k++]=a[i++];
                else
                        temp[k++]=a[j++];
        }

        while(i<=j1)                    //copy remaining elements of the first list
                temp[k++]=a[i++];

        while(j<=j2)                    //copy remaining elements of the second list
                temp[k++]=a[j++];

        /*******    Transfer elements from temp[] back to a[]    ********/
        for(i=i1,j=0;i<=j2;i++,j++)
                a[i]=temp[j];
}
```

3. Implement Quick Sort using Divide and Conquer approach

```
#include <stdio.h>

void quick_sort(int[],int,int);
int partition(int[],int,int);

int main()
{
        int a[50],n,i;
        printf("How many elements?");
        scanf("%d",&n);
        printf("\nEnter array elements:");

        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        quick_sort(a,0,n-1);
        printf("\nArray after sorting:");

        for(i=0;i<n;i++)
                printf("%d ",a[i]);

        return 0;
}

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 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);      //Position of Pivot Element
}
```

4. Find Maximum and Minimum element from a array of integer using Divide and Conquer approach

```
#include <iostream>
#include <climits>
using namespace std;

// Divide & Conquer solution to find minimum and maximum number in an array
void findMinAndMax(int arr[], int low, int high, int& min, int& max)
{
   // if array contains only one element

   if (low == high)          // common comparison
   {
      if (max < arr[low])   // comparison 1
         max = arr[low];

      if (min > arr[high])    // comparison 2
         min = arr[high];

      return;
   }

   // if array contains only two elements

   if (high - low == 1)        // common comparison
   {
      if (arr[low] < arr[high])    // comparison 1
      {
         if (min > arr[low])         // comparison 2
            min = arr[low];

         if (max < arr[high])    // comparison 3
            max = arr[high];
      }
      else
      {
```

```
            if (min > arr[high])    // comparison 2
                min = arr[high];

            if (max < arr[low])        // comparison 3
                max = arr[low];
        }
        return;
    }

    // find mid element
    int mid = (low + high) / 2;

    // recur for left sub-array
    findMinAndMax(arr, low, mid, min, max);

    // recur for right sub-array
    findMinAndMax(arr, mid + 1, high, min, max);
}

int main()
{
    int arr[] = { 7, 2, 9, 3, 1, 6, 7, 8, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // initialize the minimum element by infinity and the
    // maximum element by minus infinity
    int max = INT_MIN, min = INT_MAX;

    findMinAndMax(arr, 0, n - 1, min, max);

    cout << "The minimum element in the array is " << min << '\n';
    cout << "The maximum element in the array is " << max;

    return 0;
}
```

5. Find the minimum number of scalar multiplication needed for chain of matrix

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

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n

int MatrixChainMultiplication(int p[], int n)
{
   int m[n][n];
   int i, j, k, L, q;

   for (i=1; i<n; i++)
      m[i][i] = 0;    //number of multiplications are 0(zero) when there is only one
                      matrix

 //Here L is chain length. It varies from length 2 to length n.
   for (L=2; L<n; L++)
   {
      for (i=1; i<n-L+1; i++)
      {
         j = i+L-1;
         m[i][j] = INT_MAX;  //INT_MAX is a macro which is defined in
                                limit.h.THrough it maximum integer value can be
                                assigned in to m[i][j]

         for (k=i; k<=j-1; k++)
         {
            q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
            if (q < m[i][j])
            {
               m[i][j] = q;    //if number of multiplications found less that number
                                will be updated.
            }
         }
      }
   }
```

```
    return m[1][n-1];   //returning the final answer which is M[1][n]

}

int main()
{
    int n,i;
    printf("Enter number of matrices\n");
    scanf("%d",&n);

    n++;

    int arr[n];

    printf("Enter dimensions \n");

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

    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
                    MatrixChainMultiplication(arr, size));

    return 0;
}
```

Output

```
Enter number of matrices
4
Enter dimensions
Enter d0 :: 10
Enter d1 :: 100
Enter d2 :: 20
Enter d3 :: 5
Enter d4 :: 80
```

Minimum number of multiplications is 19000

6. Implement all pair of Shortest path for a graph (Floyd- War shall Algorithm)

```
#include<stdio.h>
int min(int,int);
void floyds(int p[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)
     p[i][j]=0;
    else
     p[i][j]=min(p[i][j],p[i][k]+p[k][j]);
}
int min(int a,int b)
{
 if(a<b)
  return(a);
 else
  return(b);
}
void main()
{
 int p[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++)
   p[i][j]=999;
 }
 for(i=1;i<=e;i++)
 {
  printf("\n Enter the end vertices of edge%d with its weight \n",i);
```

```
 scanf("%d%d%d",&u,&v,&w);
 p[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",p[i][j]);
 printf("\n");
}
floyds(p,n);
printf("\n Transitive closure:\n");
for(i=1;i<=n;i++)
{
 for(j=1;j<=n;j++)
  printf("%d \t",p[i][j]);
 printf("\n");
}
printf("\n The shortest paths are:\n");
for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
 {
  if(i!=j)
   printf("\n <%d,%d>=%d",i,j,p[i][j]);
 }
                }
```

7. Implement Single Source shortest Path for a graph (Bellman Ford Algorithm)

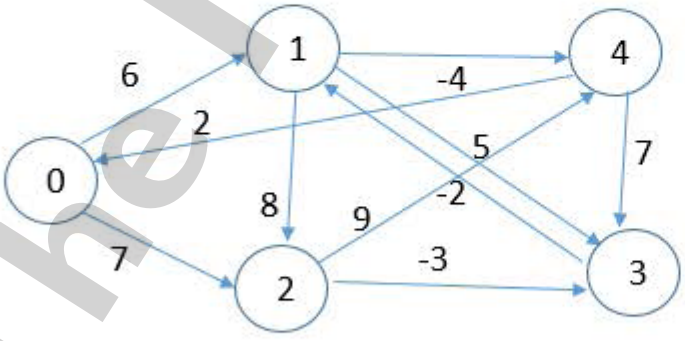
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

struct Edge
{
   // This structure is equal to an edge. Edge contains two end points. These edges
                are directed edges so they
                //contain source and destination and some weight. These 3 are
                elements in this structure
   int source, destination, weight;
};

// a structure to represent a connected, directed and weighted graph
struct Graph
{
   int V, E;
                // V is number of vertices and E is number of edges

   struct Edge* edge;
                // This structure contain another structure which we already created
                edge.
};

struct Graph* createGraph(int V, int E)
{
   struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph));
                //Allocating space to structure graph

   graph->V = V;   //assigning values to structure elements that taken form user.

   graph->E = E;

   graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );
```

```
                //Creating "Edge" type structures inside "Graph" structure, the
                number of edge type structures are equal to number of edges

    return graph;
}

void FinalSolution(int dist[], int n)
{
                // This function prints the final solution
    printf("\nVertex\tDistance from Source Vertex\n");
    int i;

    for (i = 0; i < n; ++i){
                        printf("%d \t\t %d\n", i, dist[i]);
                }
}

void BellmanFord(struct Graph* graph, int source)
{
    int V = graph->V;

    int E = graph->E;

    int StoreDistance[V];

    int i,j;

    // This is initial step that we know , we initialize all distance to infinity except
                source.
                // We assign source distance as 0(zero)

    for (i = 0; i < V; i++)
        StoreDistance[i] = INT_MAX;

    StoreDistance[source] = 0;

    //The shortest path of graph that contain V vertices, never contain "V-1" edges.
                So we do here "V-1" relaxations
    for (i = 1; i <= V-1; i++)
    {
```

```
    for (j = 0; j < E; j++)
    {
        int u = graph->edge[j].source;

        int v = graph->edge[j].destination;

        int weight = graph->edge[j].weight;

        if (StoreDistance[u] + weight < StoreDistance[v])
            StoreDistance[v] = StoreDistance[u] + weight;
    }
}

// Actually upto now shortest path found. But BellmanFord checks for negative
                edge cycle. In this step we check for that
// shortest distances if graph doesn't contain negative weight cycle.

// If we get a shorter path, then there is a negative edge cycle.
for (i = 0; i < E; i++)
{
    int u = graph->edge[i].source;

    int v = graph->edge[i].destination;

    int weight = graph->edge[i].weight;

    if (StoreDistance[u] + weight < StoreDistance[v])
        printf("This graph contains negative edge cycle\n");
}

FinalSolution(StoreDistance, V);

return;
}

int main()
{
    int V,E,S;  //V = no.of Vertices, E = no.of Edges, S is source vertex

                printf("Enter number of vertices in graph\n");
```

```
    scanf("%d",&V);

                printf("Enter number of edges in graph\n");
    scanf("%d",&E);

                printf("Enter your source vertex number\n");
                scanf("%d",&S);

    struct Graph* graph = createGraph(V, E);    //calling the function to allocate
                space to these many vertices and edges

    int i;
    for(i=0;i<E;i++){
       printf("\nEnter edge %d properties Source, destination, weight
                respectively\n",i+1);
       scanf("%d",&graph->edge[i].source);
       scanf("%d",&graph->edge[i].destination);
       scanf("%d",&graph->edge[i].weight);
    }

    BellmanFord(graph, S);
                //passing created graph and source vertex to BellmanFord Algorithm
                function

    return 0;
}
```

For the Graph



Output

```
Enter number of vertices in graph
5
Enter number of edges in graph
10
Enter your source vertex number
0
Enter edge 1 properties Source, destination, weight respectively
0 1 6
Enter edge 2 properties Source, destination, weight respectively
0 2 7
Enter edge 3 properties Source, destination, weight respectively
1 2 8
Enter edge 4 properties Source, destination, weight respectively
1 4 -4
Enter edge 5 properties Source, destination, weight respectively
1 3 5
Enter edge 6 properties Source, destination, weight respectively
3 1 -2
Enter edge 7 properties Source, destination, weight respectively
2 3 -3
Enter edge 8 properties Source, destination, weight respectively
2 4 9
Enter edge 9 properties Source, destination, weight respectively
4 0 2
Enter edge 10 properties Source, destination, weight respectively
4 3 7
Vertex  Distance from Source Vertex
0  0
1  2
2  7
3  4
4  -2
```

8. Implement Single Source shortest Path for a graph (Dijkstra Algorithm)

```
#include<stdio.h>

#define MAX 100
#define NIL -1
#define infinity 9999
#define TEMPORARY 0
#define PERMANENT 1

void Path_Finder(int source, int vertex);
void Dijkstra_function(int source);
int minimum_temp();
void make_graph();

int vertices;
int adjacent_matrix[MAX][MAX];
int predecessor[MAX];
int vertex_status[MAX];
int pathLength[MAX];

int main()
{
    int source, vertex;
    make_graph();
    printf("Enter Source Vertex:\t");
    scanf("%d", &source);
    Dijkstra_function(source);
    while(1)
    {
        printf("Enter destination vertex(-1 to Quit):\t");
        scanf("%d", &vertex);
        if(vertex == -1)
        {
            break;
        }
        if(vertex < 0 || vertex >= vertices)
        {
```

```
            printf("The Entered Vertex does not exist\n");
        }
        else if(vertex == source)
        {
            printf("Source Vertex and Destination Vertex are same\n");
        }
        else if(pathLength[vertex] == infinity)
        {
            printf("There is no path from Source vertex to Destination vertex\n");
        }
        else
        {
            Path_Finder(source, vertex);
        }
    }
    return 0;
}

void Dijkstra_function(int source)
{
    int count, current;
    for(count = 0; count < vertices; count++)
    {
        predecessor[count] = NIL;
        pathLength[count] = infinity;
        vertex_status[count] = TEMPORARY;
    }
    pathLength[source] = 0;
    while(1)
    {
        current = minimum_temp();
        if(current == NIL)
        {
            return;
        }
        vertex_status[current] = PERMANENT;
        for(count = 0; count < vertices; count++)
        {
            if(adjacent_matrix[current][count] != 0 && vertex_status[count] ==
              TEMPORARY)
```

```
            {
              if( pathLength[current] + adjacent_matrix[current][count] <
            pathLength[count])
              {
                  predecessor[count] = current;
                  pathLength[count] = pathLength[current] +
            adjacent_matrix[current][count];
              }
            }
        }
    }
}

int minimum_temp()
{
    int count;
    int min = infinity;
    int x = NIL;
    for(count = 0; count < vertices; count++)
    {
        if(vertex_status[count] == TEMPORARY && pathLength[count] < min)
        {
            min = pathLength[count];
            x = count;
        }
    }
    return x;
}

void Path_Finder(int source, int vertex)
{
    int count, u;
    int path[MAX];
    int shortest_distance = 0;
    int temp = 0;
    while(vertex != source)
    {
        temp++;
        path[temp] = vertex;
        u = predecessor[vertex];
```

```
            shortest_distance = shortest_distance + adjacent_matrix[u][vertex];
            vertex = u;
        }
        count++;
        path[temp] = source;
        printf("Shortest Path\n");
        for(count = temp; count >= 1; count--)
        {
            printf("%d  ", path[count]);
        }
        printf("\nShortest distance:\t%d\n", shortest_distance);
}

void make_graph()
{
        int count, maximum_edges, origin_vertex, destination_vertex, weight;
        printf("Enter total number of vertices:\t");
        scanf("%d", &vertices);
        maximum_edges = vertices * (vertices - 1);
        for(count = 0; count < maximum_edges; count++)
        {
            printf("Enter Edge [%d] Co-ordinates [-1 -1] to Quit\n", count + 1);
            printf("Enter Origin Vertex Point:\t");
            scanf("%d", &origin_vertex);
            printf("Enter Destination Vertex Point:\t");
            scanf("%d", &destination_vertex);
            if((origin_vertex == -1) && (destination_vertex == -1))
            {
                break;
            }
            printf("Enter the weight for this edge:\t");
            scanf("%d", &weight);
            if(origin_vertex >= vertices || destination_vertex >= vertices || origin_vertex
                    < 0 || destination_vertex < 0)
            {
                printf("Edge Co - ordinates are Invalid\n");
                count--;
            }
            else
            {
```

```
                adjacent_matrix[origin_vertex][destination_vertex] = weight;
            }
        }
    }
```

9. Implement 8 Queen problem

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

int board[20],count;

int main()
{
int n,i,j;
void queen(int row,int n);

printf(" - N Queens Problem Using Backtracking -");
printf("\n\nEnter number of Queens:");
scanf("%d",&n);
queen(1,n);
return 0;
}

//function for printing the solution
void print(int n)
{
int i,j;
printf("\n\nSolution %d:\n\n",++count);

for(i=1;i<=n;++i)
  printf("\t%d",i);

for(i=1;i<=n;++i)
{
  printf("\n\n%d",i);
  for(j=1;j<=n;++j) //for nxn board
  {
   if(board[i]==j)
    printf("\tQ"); //queen at i,j position
   else
    printf("\t-"); //empty slot
  }
```

```
}
}

/*funtion to check conflicts
If no conflict for desired postion returns 1 otherwise returns 0*/
int place(int row,int column)
{
int i;
for(i=1;i<=row-1;++i)
{
 //checking column and digonal conflicts
 if(board[i]==column)
  return 0;
 else
  if(abs(board[i]-column)==abs(i-row))
   return 0;
}

return 1; //no conflicts
}

//function to check for proper positioning of queen
void queen(int row,int n)
{
int column;
for(column=1;column<=n;++column)
{
 if(place(row,column))
 {
  board[row]=column; //no conflicts so place queen
  if(row==n) //dead end
   print(n); //printing the board configuration
  else //try queen with next position
   queen(row+1,n);
 }
}
}
```

Note: It is a n-quine problem,. In the place of n, if you put value as 8, it will be 8 quine problem.

10. Implement Travelling Salesman problem

```
#include<stdio.h>

int matrix[25][25], visited_cities[10], limit, cost = 0;

int tsp(int c)
{
    int count, nearest_city = 999;
    int minimum = 999, temp;
    for(count = 0; count < limit; count++)
    {
        if((matrix[c][count] != 0) && (visited_cities[count] == 0))
        {
            if(matrix[c][count] < minimum)
            {
                minimum = matrix[count][0] + matrix[c][count];
            }
            temp = matrix[c][count];
            nearest_city = count;
        }
    }
    if(minimum != 999)
    {
        cost = cost + temp;
    }
    return nearest_city;
}

void minimum_cost(int city)
{
    int nearest_city;
    visited_cities[city] = 1;
    printf("%d ", city + 1);
    nearest_city = tsp(city);
    if(nearest_city == 999)
    {
        nearest_city = 0;
```

```
        printf("%d", nearest_city + 1);
        cost = cost + matrix[city][nearest_city];
        return;
    }
    minimum_cost(nearest_city);
}

int main()
{
    int i, j;
    printf("Enter Total Number of Cities:\t");
    scanf("%d", &limit);
    printf("\nEnter Cost Matrix\n");
    for(i = 0; i < limit; i++)
    {
        printf("\nEnter %d Elements in Row[%d]\n", limit, i + 1);
        for(j = 0; j < limit; j++)
        {
            scanf("%d", &matrix[i][j]);
        }
        visited_cities[i] = 0;
    }
    printf("\nEntered Cost Matrix\n");
    for(i = 0; i < limit; i++)
    {
        printf("\n");
        for(j = 0; j < limit; j++)
        {
            printf("%d ", matrix[i][j]);
        }
    }
    printf("\n\nPath:\t");
    minimum_cost(0);
    printf("\n\nMinimum Cost: \t");
    printf("%d\n", cost);
    return 0;
}
```

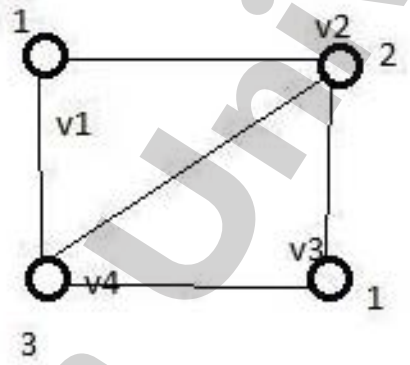
11. Implement Graph Coloring problem

```
#include<stdio.h>
int G[50][50],x[50];  //G:adjacency matrix,x:colors
void next_color(int k)
{
  int i,j;
  x[k]=1;  //coloring vertex with color1
  for(i=0;i<k;i++){ //checking all k-1 vertices-backtracking
                if(G[i][k]!=0 && x[k]==x[i])  //if connected and has same color
                  x[k]=x[i]+1;  //assign higher color than x[i]
  }
}

int main()
{
 int n,e,i,j,k,l;
 printf("Enter no. of vertices : ");
 scanf("%d",&n);  //total vertices
 printf("Enter no. of edges : ");
 scanf("%d",&e);  //total edges

 for(i=0;i<n;i++)
                for(j=0;j<n;j++)
                  G[i][j]=0;  //assign 0 to all index of adjacency matrix

 printf("Enter indexes where value is 1-->\n");
 for(i=0;i<e;i++)
 {
                scanf("%d %d",&k,&l);
                G[k][l]=1;
                G[l][k]=1;
 }

 for(i=0;i<n;i++)
                next_color(i);  //coloring each vertex

 printf("Colors of vertices -->\n");
 for(i=0;i<n;i++)  //displaying color of each vertex
```

```
            printf("Vertex[%d] : %d\n",i+1,x[i]);

 return 0;
}
```

NOTE : This code is written, compiled and run with GCC compiler under Linux environment Ubuntu12.04LTS Precise Pangolin.

Out put for the graph



Colored vertices of Graph G

```
Enter no. of vertices : 4
Enter no. of edges : 5
Enter indexes where value is 1-->
0 1
1 2
1 3
2 3
3 0
Colors of vertices -->
Vertex[1] : 1
Vertex[2] : 2
Vertex[3] : 1
Vertex[4] : 3
```

12. Implement Job sequencing with deadlines problem

```
#include <stdio.h>

#define MAX 100

typedef struct Job {       // Using typedef, we can add new name of a datatype
  char id[5];
  int deadline;
  int profit;
} Job;
void jobSequencingWithDeadline(Job jobs[], int n);

int minValue(int x, int y) {
  if(x < y) return x;
  return y;
}

int main(void) {
  //variables
  int i, j;

  //jobs with deadline and profit
  Job jobs[5] = {
    {"j1", 2,  60},
    {"j2", 1, 100},
    {"j3", 3,  20},
    {"j4", 2,  40},
    {"j5", 1,  20},
  };

  //temp
  Job temp;

  //number of jobs
  int n = 5;

  //sort the jobs profit wise in descending order
  for(i = 1; i < n; i++) {
    for(j = 0; j < n - i; j++) {
```

```
      if(jobs[j+1].profit > jobs[j].profit) {
        temp = jobs[j+1];
        jobs[j+1] = jobs[j];
        jobs[j] = temp;
      }
    }
  }

  printf("%10s %10s %10s\n", "Job", "Deadline", "Profit");
  for(i = 0; i < n; i++) {
    printf("%10s %10i %10i\n", jobs[i].id, jobs[i].deadline, jobs[i].profit);
  }

  jobSequencingWithDeadline(jobs, n);

  return 0;
}

void jobSequencingWithDeadline(Job jobs[], int n) {
  //variables
  int i, j, k, maxprofit;

  //free time slots
  int timeslot[MAX];

  //filled time slots
  int filledTimeSlot = 0;

  //find max deadline value
  int dmax = 0;
  for(i = 0; i < n; i++) {
    if(jobs[i].deadline > dmax) {
      dmax = jobs[i].deadline;
    }
  }

  //free time slots initially set to -1 [-1 denotes EMPTY]
  for(i = 1; i <= dmax; i++) {
    timeslot[i] = -1;
  }
```

```
printf("dmax: %d\n", dmax);

for(i = 1; i <= n; i++) {
  k = minValue(dmax, jobs[i - 1].deadline);
  while(k >= 1) {
    if(timeslot[k] == -1) {
      timeslot[k] = i-1;
      filledTimeSlot++;
      break;
    }
    k--;
  }

  //if all time slots are filled then stop
  if(filledTimeSlot == dmax) {
    break;
  }
}

//required jobs
printf("\nRequired Jobs: ");
for(i = 1; i <= dmax; i++) {
  printf("%s", jobs[timeslot[i]].id);

  if(i < dmax) {
    printf(" --> ");
  }
}

//required profit
maxprofit = 0;
for(i = 1; i <= dmax; i++) {
  maxprofit += jobs[timeslot[i]].profit;
}
printf("\nMax Profit: %d\n", maxprofit);
}
```

13. Implement Longest Common Subsequence problem

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

int i,j,m,n,c[20][20];
char x[20],y[20],b[20][20];

void print(int i,int j)
{
        if(i==0 || j==0)
                return;
        if(b[i][j]=='c')
        {
                print(i-1,j-1);
                printf("%c",x[i-1]);
        }
        else if(b[i][j]=='u')
                print(i-1,j);
        else
                print(i,j-1);
}

void lcs()
{
        m=strlen(x);
        n=strlen(y);
        for(i=0;i<=m;i++)
                c[i][0]=0;
        for(i=0;i<=n;i++)
                c[0][i]=0;

        //c, u and l denotes cross, upward and downward directions respectively
        for(i=1;i<=m;i++)
                for(j=1;j<=n;j++)
                {
                        if(x[i-1]==y[j-1])
                        {
                                c[i][j]=c[i-1][j-1]+1;
```

```
                    b[i][j]='c';
                }
                else if(c[i-1][j]>=c[i][j-1])
                {
                    c[i][j]=c[i-1][j];
                    b[i][j]='u';
                }
                else
                {
                    c[i][j]=c[i][j-1];
                    b[i][j]='l';
                }
            }
}
int main()
{
    printf("Enter 1st sequence:");
    scanf("%s",x);
    printf("Enter 2nd sequence:");
    scanf("%s",y);
    printf("\nThe Longest Common Subsequence is ");
    lcs();
    print(m,n);
      return 0;
}
```