Implement a graph using adjacency matrix representation

```
#include<stdio.h>
int no_vertices;
//implimenting the matrx
void printGraph(int adj[][no_vertices])
{
      for(int i=0;i<no_vertices;i++)
      {
            for(int j =0;j<no_vertices;j++)
            {
                  printf(" %d  ",adj[i][j]);
            }
            printf("\n");
      }

}
int main()
{
      int s,d;
      printf("\nEnter the number of vertices : ");
      scanf("%d",&no_vertices);
      int adj[no_vertices][no_vertices],visited[no_vert


      for(int i =0;i<no_vertices;i++)
            for(int j=0;j<no_vertices;j++)
                  adj[i][j] =0;

      while(s!=-1 && d!=-1)
      {
      printf("Enter an Edge fromnode(0 to %d) to node(0 to %d) :
",no_vertices,no_vertices);
      scanf("%d%d",&s,&d);
      adj[s][d] = 1;
      adj[d][s] = 1;
      }
    printGraph(adj);

    return 0;
}
```

Approach: The idea is to use a square Matrix of size **NxN** to create Adjacency Matrix. Below are the steps:
Create a 2D array(say **Adj[N+1][N+1]**) of size NxN and initialize all value of this matrix to zero.
For each edge in arr[][](say **X and Y**), Update value at **Adj[X][Y]** and **Adj[Y][X]** to 1, denotes that there is a edge between X and Y.
Display the Adjacency Matrix after the above operation for all the pairs in **arr[][]**.

The time complexity of printing the adjacency matrix is O(V^2), where V is the number of vertices, because we need to iterate over all the elements in the matrix.

Implement DFS traversal on the graph using matrix

```
#include<stdio.h>
#include<stdlib.h>
#define VER 5

int visited[VER],adjmat[VER][VER];


int stack[VER];
int top=-1;
//stack operations
void push(int item) {
   stack[++top] = item;
}

int pop() {
   return stack[top--];
```

```
}

int peek() {
   return stack[top];
}

int isStackEmpty() {
   if(top == -1)
      return 1;
   else
      return 0;
}

// INITIALIZE THE MATRIX TO ZERO
void init(int arr[][VER])
 {
  int i, j;
  for (i = 0; i < VER; i++)
    for (j = 0; j < VER; j++)
      arr[i][j] = 0;
}
 // ADD EDGES
void addEdge(int arr[][VER], int i, int j)
{
  arr[i][j] = 1;
  arr[j][i] = 1;
}
// PRINT THE MATRIX
void printadjmat(int arr[][VER])
{
  int i, j;
  for (i = 0; i < VER; i++)
  {
    for (j = 0; j < VER; j++)
    {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
}


int getUnvisitedVertex(int vertexIndex) {
   int i;

   for(i = 0; i < VER; i++) {
      if(adjmat[vertexIndex][i] == 1 && visited[i] == 0) {
         return i;
      }
   }

   return -1;
}

void dfs(int start_vertex)
{

    visited[start_vertex]=1;
    printf("%d\t",start_vertex);
```

Approach:

Create a matrix of size n*n where every element is 0 representing there is no edge in the graph.

Now, for every edge of the graph between the vertices i and j set mat[i][j] = 1.

After the adjacency matrix has been created and filled, call the recursive function for the source i.e. vertex 0 that will recursively call the same function for all the vertices adjacent to it.

Also, keep an array to keep track of the visited vertices i.e. visited[i] = true represents that vertex i has been visited before and the DFS function for some already visited node need not be called.

The **time complexity** of the above implementation of DFS on an adjacency matrix is **O(V^2)**,

```
    push(start_vertex);
    while(!isStackEmpty())
    {
        int unvisitedVertex = getUnvisitedVertex(peek());

      //no adjacent vertex found
        if(unvisitedVertex == -1)
        {
            pop();
        }
        else
        {
            visited[unvisitedVertex]= 1;
            printf("%d\t",unvisitedVertex);
            push(unvisitedVertex);
        }
   }
}
int main()
{

   int i,x,y,no_of_edges,first_v;
   printf("enter number of edges");
   scanf("%d",&no_of_edges);
   init(adjmat);
   for(i=0;i<no_of_edges;i++)
{
         printf("enter the VERtex from which edge is to be added-");
         scanf("%d",&x);
         printf("enter the VERtex from %d towards which edge is to be added-",x);
         scanf("%d",&y);
         addEdge(adjmat,x,y);
   }
   printadjmat(adjmat);

   for(int i=0;i<VER;i++)
   {
           visited[i] =0;
   }
   printf("enter vertex to start");
   scanf("%d",&first_v);
   dfs(first_v);
   return 0;
}
```

impliment bfs traversal

```
#include<stdio.h>
#include<stdlib.h>
int vertices=5;
struct queue
{
    int size;
    int f;
    int r;
    int* arr;
};
int isEmpty(struct queue *q){
    if(q->r==q->f){
        return 1;
```

Approach:

Create a matrix of size n*n where every element is 0 representing there is no edge in the graph.
Now, for every edge of the graph between the vertices i and j set mat[i][j] = 1.
After the adjacency matrix has been created and filled, find the BFS traversal of the graph

Time Complexity: O(N*N)

```
    }
    return 0;
}

int isFull(struct queue *q){
    if(q->r==q->size-1){
        return 1;
    }
    return 0;
}

void enqueue(struct queue *q, int val){
    if(isFull(q)){
        printf("This Queue is full\n");
    }
    else{
        q->r++;
        q->arr[q->r] = val;

    }
}

int dequeue(struct queue *q){
    int a = -1;
    if(isEmpty(q)){
        printf("This Queue is empty\n");
    }
    else{
        q->f++;
        a = q->arr[q->f];
    }
    return a;
}

// INITIALIZE THE MATRIX TO ZERO
void init(int arr[][vertices])
 {
  int i, j;
  for (i = 0; i < vertices; i++)
    for (j = 0; j < vertices; j++)
      arr[i][j] = 0;
}
 // ADD EDGES
void addEdge(int arr[][vertices], int i, int j)
{
  arr[i][j] = 1;
  arr[j][i] = 1;
}
// PRINT THE MATRIX
void printAdjMatrix(int arr[][vertices])
{
  int i, j;
  for (i = 0; i < vertices; i++)
  {
    for (j = 0; j < vertices; j++)
    {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
```

```
  }
}
int main(){
    struct queue q;
    int adjmat[vertices][vertices],visited[vertices];
    q.size = vertices;
    q.f = q.r = 0;
    q.arr = (int*) malloc(q.size*sizeof(int));
    int i,x,y,no_of_edges,first_v;
    printf("enter number of edges");
    scanf("%d",&no_of_edges);
    init(adjmat);
    for(i=0;i<no_of_edges;i++)
    {
        printf("enter the verticestex from which edge is to be added-");
        scanf("%d",&x);
        printf("enter the verticestex from %d towards which edge is to be
added-",x);
        scanf("%d",&y);
        addEdge(adjmat,x,y);
    }
    printAdjMatrix(adjmat);

    for(int i=0;i<vertices;i++)
    {
            visited[i] =0;
    }
    printf("enter vertex to start");
    scanf("%d",&first_v);
    //BFS TRAVERSAL
    int node;
    printf("%d\t",first_v);
    visited[first_v]=1;
    enqueue(&q,first_v);
    while(!isEmpty(&q))
    {
        node=dequeue(&q);
        for(int j=0;j<vertices;j++)
        {
            if(adjmat[node][j]==1 && visited[j]==0)
            {
                printf("%d\t",j);
                visited[j]=1;
                enqueue(&q,j);
            }
        }
    }
    return 0;
}
Implement a graph using adjacency list representation
#include<stdio.h>
#include<stdlib.h>
int no_vertices;
struct node
{
      int data;
      struct node *next;
};
void readgraph(struct node *ad[])
```

```
{
	struct node *newnode;
	int i,j,k,data;
	for(i=0;i<no_vertices;i++)
	{
		struct node *last =NULL;
		printf("\nHow many vertices adjacent to %d  :",i);
		scanf("%d",&k);

		for(j=1;j<=k;j++)
		{
			printf("Enter the value of %d neighbour of %d : ",j,i);
			scanf("%d",&data);

			newnode = (struct node*)malloc(sizeof(struct node*));
			newnode->data = data;
			newnode->next = NULL;
			if(ad[i]==NULL)
			{
				ad[i] = newnode;
			}
			else
				last->next = newnode;

			last = newnode;
		}
	}
}
void printgraph(struct node *adj[])
{
	struct node *ptr  = NULL;
	int i,j;
	for(i=0;i<no_vertices;i++)
	{
		ptr = adj[i];
		printf("\n vertices adjacent to %d are :",i);
		while(ptr != NULL)
		{
			printf("%d\t",ptr->data);
			ptr = ptr->next;
		}
	}
}
int main()
{
	int i,j,k;
	printf("Enter the number of vertices in the graph :");
	scanf("%d",&no_vertices);
	int visited[no_vertices];
	struct node *adj[no_vertices];
	for(i=0;i<no_vertices;i++)
	{
		adj[i] = NULL;
	}
	readgraph(adj);
	printgraph(adj);
}
BFS and DFS using list
#include <stdio.h>
```

```c
#include <stdlib.h>

// Define the maximum number of vertices in the graph
#define N 6
 int visited[N];
 // Data structure to store adjacency list nodes of the graph
struct node
{
    int dest;
    struct node* next;
};

// Data structure to store a graph object
struct Graph
{
    // An array of pointers to node to represent an adjacency list
    struct node* head[N];
};



// Data structure to store a graph edge
struct Edge {
    int src, dest;
};

// Function to create an adjacency list from specified edges
struct Graph* createGraph(struct Edge edges[], int n)
{
    // allocate storage for the graph data structure
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));

    // initialize head pointer for all vertices
    for (int i = 0; i < N; i++) {
        graph->head[i] = NULL;
    }

    // add edges to the directed graph one by one
    for (int i = 0; i < n; i++)
    {
        // get the source and destination vertex
        int src = edges[i].src;
        int dest = edges[i].dest;

        // 1. allocate a new node of adjacency list from src to dest

        struct node* newnode = (struct node*)malloc(sizeof(struct node));
        newnode->dest = dest;

        // point new node to the current head
        newnode->next = graph->head[src];

        // point head pointer to the new node
        graph->head[src] = newnode;

        // 2. allocate a new node of adjacency list from `dest` to `src`

        newnode = (struct node*)malloc(sizeof(struct node));
        newnode->dest = src;
```

```
        // point new node to the current head
        newnode->next = graph->head[dest];

        // change head pointer to point to the new node
        graph->head[dest] = newnode;
    }

    return graph;
}

// Function to print adjacency list representation of a graph
void printGraph(struct Graph* graph)
{
    for (int i = 0; i < N; i++)
    {
        // print current vertex and all its neighbors
        struct node* ptr = graph->head[i];
        while (ptr != NULL)
        {
            printf("(%d ->%d)\t", i, ptr->dest);
            ptr = ptr->next;
        }

        printf("\n");
    }
}
//QUEUE OPERATIONS
struct node *createnode(int v)
{
    struct node *newnode = (struct node*)malloc(sizeof(struct node));
    newnode->dest = v;
    newnode->next = NULL;
    return newnode;
}
int isEmpty(struct node *queue)
{
    return queue == NULL;
}

void enqueue(struct node **queue, int value)
{
    struct node *newnode = createnode(value);
    if (isEmpty(*queue))
    {
        *queue = newnode;
    }
    else
    {
        struct node *temp = *queue;
        while (temp->next)
        {
            temp = temp->next;
        }
        temp->next = newnode;
    }
}

int dequeue(struct node **queue)
```

```
{
    int nodeData = (*queue)->dest;
    struct node *temp = *queue;
    *queue = (*queue)->next;
    free(temp);
    return nodeData;
}

 //BFS TRAVERSAL
void bfs(struct Graph *graph, int startVertex)
{
    struct node *queue = NULL;
    visited[startVertex] = 1;
    enqueue(&queue, startVertex);
    printf("");
    while (!isEmpty(queue))
    {
        //printQueue(queue);
        int currentVertex = dequeue(&queue);
        printf("%d \t", currentVertex);

        struct node *temp = graph->head[currentVertex];

        while (temp)
        {
            int adjVertex = temp->dest;

            if (visited[adjVertex] == 0)
            {

                visited[adjVertex] = 1;

                enqueue(&queue, adjVertex);
            }
            temp = temp->next;

        }

    }
}

struct stack{
     int data;
     struct stack *next ;
};
typedef struct stack *stackpointer;
stackpointer top= NULL;


void push(int data){
        stackpointer temp = (stackpointer)malloc(sizeof(struct stack));
        temp->data= data;
        temp->next = top;
        top= temp;

}
int pop(){
     stackpointer temp ;
     int value;
```

```
     value= top->data;
     temp= top ;
     top= top->next;
     free(temp);
     return value;
}

void dfs(struct Graph *graph,int v){
     struct node* w;
     int vnext;

     push(v);

     while(top){
         vnext=pop();
         visited[vnext]=1;
         printf("%d\t", vnext);
        w=graph->head[vnext];
        while(w!=NULL){

              if(!visited[w->dest]){
                   push(w->dest);
              }
              w=w->next;
         }

     }
}


// MAIN FUNCTION
int main(void)
{
    int i;
    // (x, y) pair in the array represents an edge from x to y
    struct Edge edges[] =
    {
        {0, 1}, {1, 2}, {2, 0}, {2, 1}, {3, 2}, {3, 4}, {5, 4}
    };

    // calculate the total number of edges
    int n = sizeof(edges)/sizeof(edges[0]);

    // construct a graph from the given edges
    struct Graph *graph = createGraph(edges, n);

    // Function to print adjacency list representation of a graph
    printGraph(graph);
    //calling BFS function
    printf("BFS TRAVERSAL\t");
    bfs(graph,0);
    for(i=0;i<N;i++)
    {
        visited[i]=0;
    }
    printf("\nDFS TRAVESRAL\t");
    dfs(graph,0);
    return 0;
}
```

WEIGHTED GRAPH

```
#include <stdio.h>
#include <stdlib.h>
// Define the maximum number of vertices in the graph
#define N 6
// Data structure to store adjacency list nodes of the graph
struct Node
{
    int dest, weight;
    struct Node* next;
};
// Data structure to store graph
struct Graph
{
    // An array of pointers to `Node` for representing adjacency list
    struct Node* head[N];
};
// Data structure to store graph edges
struct Edge {
    int src, dest, weight;
};
// Function to create an adjacency list from specified edges
struct Graph* createGraph(struct Edge edges[], int n)
{
    int i;
    // allocate memory for the graph data structure
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    // initialize head pointer for all vertices
    for (i = 0; i < N; i++) {
        graph->head[i] = NULL;
    }
    // add edges to the  graph one by one
```

Find the shortest path from one vertex to another, where "shortest" is defined in terms of the *path weight* (i.e., the sum of all the weights of the edges in the path), rather than just the number of edges.
Find the *maximal flow* of a graph between one vertex and another, if we treat the weights as capacities.
Find a *minimum spanning tree* of an (undirected) weighted graph. The MST is a tree built from edges in the graph (i.e., a "subgraph") where the sum of all the edges is as small as possible. The MST is useful because it is essentially a graph in which *every* simple path is *the* shortest path between its two endpoints; it is not possible to construct a non-shortest path in a MST.

Time Complexity is O(V^2)

```
    for (i = 0; i < n; i++)
    {
        // get the source and destination vertex
        int src = edges[i].src;
        int dest = edges[i].dest;
        int weight = edges[i].weight;
        // allocate new node of adjacency list from `src` to `dest`
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->dest = dest;
        newNode->weight = weight;
        // point new node to current head
        newNode->next = graph->head[src];
        // point head pointer to a new node
        graph->head[src] = newNode;
        // allocate new node of adjacency list from `dest` to `src`
        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->dest = src;
        newNode->weight = weight;
        // point new node to current head
        newNode->next = graph->head[dest];

        // change head pointer to point to the new node
        graph->head[dest] = newNode;
    }
    return graph;
}
// Function to print adjacency list representation of a graph
void printGraph(struct Graph* graph)
{
    int i;
    for (i = 0; i < N; i++)
    {
        // print the current vertex and all its neighbors
```

```
        struct Node* ptr = graph->head[i];
        while (ptr != NULL)
        {
            printf("%d -> %d (%d)\t", i, ptr->dest, ptr->weight);
            ptr = ptr->next;
        }
        printf("\n");
    }
}
int main(void)
{
    int s,d,weigh,no_edg=7;
    struct Edge edges[no_edg] ;
    for(int i=0;i<no_edg;i++)
    {
        printf("enter source node of edge-%d:",i+1);
        scanf("%d",&s);
        edges[i].src=s;
        printf("enter destination node of edge-%d:",i+1);
        scanf("%d",&d);
        edges[i].dest=d;
        printf("enter weight  of edge-%d:",i+1);
        scanf("%d",&weigh);
        edges[i].weight=weigh;
    }
    struct Graph *graph = createGraph(edges, no_edg);
    printGraph(graph);
    return 0;
}
```

PRIM'S ALGORITHM

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
// number of vertices in the graph
#define V 6
int minkey(int key[], bool mstset[])
{
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      // this condition checks  whether the node being considered is not in the mst set
      if (mstset[v] == false && key[v] < min)
         min = key[v], min_index = v;
   return min_index;
}
int printMST(int parent[], int graph[V][V])
{
   printf("Edge \tWeight\n");
   for (int i = 1; i < V+1; i++)
   {
      printf("%d-%d\t%d\n", parent[i], i, graph[i][parent[i]]);
   }
}
void primMST(int graph[V][V])
{
   int parent[V];
   int key[V];
   bool mstset[V];
   for (int i = 0; i < V; i++)
   {
      key[i] = INT_MAX, mstset[i] = false;
      key[0] = 0;
```

ALGORITHM for Prim's Algorithm

First, we have to initialize an MST with the randomly chosen vertex.
Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
Repeat step 2 until the minimum spanning tree is formed.

The time complexity of the Prim's Algorithm is O ((V + E) l o g V) because each edge is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

```
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++)
    {
      // intially u will be 0 and then itw illl be included in the mstset
      int u = minkey(key, mstset);
      mstset[u] = true;
      for (int v = 0; v < V; v++)
      {
        // condition chcks that u anv are are connected  and v is not included in the mstset
        // and the edge weight is< than the key value(infinte)
        if (graph[u][v] && mstset[v] == false && graph[u][v] < key[v])
        {
          parent[v] = u, key[v] = graph[u][v];
        }
      }
    }
  printMST(parent, graph);
}
int main()
{
  int graph[V][V] = {{0, 2, 0, 1, 0, 4},
                     {2, 0, 8, 0, 0, 3},
                     {0, 8, 0, 6, 1, 0},
                     {0, 0, 6, 0, 12, 0},
                     {0, 0, 1, 0, 12, 0},
                     {4, 3, 0, 0, 2, 0}};

  primMST(graph);
  return 0;
}
```

KRUSKAL ALGORITHM

```
#include<stdio.h>
#define VERTICES 7

int visited[VERTICES];
typedef struct array_coor
{
 int row;
 int col;
 int wght;
}N;
// INITIALIZE THE MATRIX TO ZERO
void init(int arr[][VERTICES])
{
 int i, j;
 for (i = 0; i < VERTICES; i++)
   for (j = 0; j < VERTICES; j++)
     arr[i][j] = 999;
}
 // ADD EDGES
void addEdge(int arr[][VERTICES], int i, int j,int cost)
{

 arr[i][j] = cost;
 arr[j][i] = cost;
}
// PRINT THE MATRIX
void printadjmat(int arr[][VERTICES])
{
 int i, j;
 for (i = 0; i < VERTICES; i++)
 {
```

ALGORITHM for Kruskal Algorithm

First, sort all the edges from low weight to high.
Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

The time complexity of Kruskal's algorithm? The time complexity in Kruskal's algorithm is O(E log V), where \ is the number of vertices.

```
   for (j = 0; j < VERTICES; j++)
   {
     printf("%4d ", arr[i][j]);
   }
   printf("\n");
  }
}
//MINIMUM ELEMENT IN ARRAY
N min_ele(int arr[VERTICES][VERTICES])
{
 N new;
 int smallest=arr[0][0],row,column,i,j;
 for(i=0;i<VERTICES;i++)
 {
  for( j=0;j<VERTICES;j++)
  {
    if(smallest>arr[i][j])
    {
      smallest=arr[i][j];
      row=i;
      column=j;
    }
  }
 }
 new.row=row;
 new.col=column;
 new.wght=smallest;
 return new;
}

int main()
{
 int adjmat[VERTICES][VERTICES],mst[VERTICES][VERTICES];
```

```
int i,x,y,no_of_edges;
int cost;
N new;
//CREATING GRAPH
printf("enter number of edges");
scanf("%d",&no_of_edges);
init(adjmat);
for(i=0;i<no_of_edges;i++)
{
 printf("enter the VERTICEStex from which edge is to be added-");
 scanf("%d",&x);
 printf("enter the VERTICEStex from %d towards which edge is to be added-",x);
 scanf("%d",&y);
 printf("enter the cost ");
 scanf("%d",&cost);
 addEdge(adjmat,x,y,cost);
}

printadjmat(adjmat);

for(int i=0;i<VERTICES;i++)
{
 visited[i] =0;
}

int count=0;
init(mst);
for(i=0;i<VERTICES;i++)
{
 while(visited[i]==0)
 {
   new=min_ele(adjmat);
   if(visited[new.row]==0 && visited[new.col]==0)
```

```
{
 addEdge(mst,new.row,new.col,new.wght);
 count=count+1;
 visited[new.row]=visited[new.col]=count;
}
else if(visited[new.row]!=visited[new.col] )
{
 if(visited[new.row]==0)
 {
  addEdge(mst,new.row,new.col,new.wght);
  visited[new.row]=visited[new.col];
 }
 else if(visited[new.col]==0)
 {
  addEdge(mst,new.row,new.col,new.wght);
  visited[new.col]=visited[new.row];
 }
 else if(visited[new.row]!=0 && visited[new.col]!=0)
 {
  addEdge(mst,new.row,new.col,new.wght);
  if(visited[new.row]>visited[new.col])
  {
   for(int p=0;p<VERTICES;p++)
   {
    if(visited[p]!=0)
    {
     visited[p]=visited[new.row];
    }
   }
  }
  else
  {
   for(int p=0;p<VERTICES;p++)
```

```
        {
          if(visited[p]!=0)
          {
            visited[p]=visited[new.col];
          }
        }
       }
      }
     else if(visited[new.row]==visited[new.col] && visited[new.col]!=0 && visited[new.row]!=0)
     {
      adjmat[new.row][new.col]=999;
      adjmat[new.col][new.row]=999;
      break;
     }
     adjmat[new.row][new.col]=999;
     adjmat[new.col][new.row]=999;
   }
  }
  printf("\n MINIMUM SPANNING TREE USING KRUSKAL\n");
  printadjmat(mst);
  return 0;

}
```

Assignment 3

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include<conio.h>
typedef struct {
 double hi;
 double lo;
} two_val;
double hirec(int );
double lorec(int );
double hirec ( int n )
{
//complete the code
if (n==0){
 return 1.0000;
}
else {
 return (2*hirec(n-1)+lorec(n-1));
}
}
double lorec ( int n )
{
//complete the code
if (n==0){
 return 0.0000;
}
else{
 return (hirec(n-1)+lorec(n-1));
}
}
two_val hilorec ( int n )
{
 two_val N;
//complete the code
if (n==0){
 return (two_val){1,0};
}
else{
 N=hilorec(n-1);
 return (two_val){2*N.hi+N.lo,N.hi+N.lo};
}
}
two_val hiloformula ( int n )
{
 two_val N;
 N.hi =(5+sqrt(5))/10*pow((3-sqrt(5))/2,n+1)+(5-
sqrt(5))/10*pow((3+sqrt(5))/2,n+1);
 N.lo = (-5-(3*sqrt(5)))/10*pow((3-sqrt(5))/2,n+1)+ (-
5+(3*sqrt(5)))/10*pow((3+sqrt(5))/2,n+1);
 return N;
}
int main ( int argc, char *argv[] )
{
 two_val N1, N2, N3;
 int n;
 clock_t start, end;
 double cpu_time_used;
 scanf("%d", &n);
 printf("n = %d\n", n);
 printf("\n+++ Method 0\n");
 start = clock();
z
N1.hi = hirec(n); N1.lo = lorec(n);
 end = clock();
 printf(" hi(%d) = %.10le, lo(%d) = %.10le\n", n, N1.hi, n, N1.lo);
 cpu_time_used = ((double) (end - start))/CLOCKS_PER_SEC;
 printf("Method 0 took %f seconds to execute \n", cpu_time_used);
 printf("\n+++ Method 1\n");
 start = clock();
 N2 = hilorec(n);
 end = clock();
 printf(" hi(%d) = %.10le, lo(%d) = %.10le\n", n, N2.hi, n, N2.lo);
 cpu_time_used = ((double) (end - start))/CLOCKS_PER_SEC;
 printf("Method 1 took %f seconds to execute \n", cpu_time_used);
 printf("\n+++ Method 2\n");
 start = clock();
 N3 = hiloformula(n);
 end = clock();
 printf(" hi(%d) = %.10le, lo(%d) = %.10le\n", n, N3.hi, n, N3.lo);
 cpu_time_used = ((double) (end - start))/CLOCKS_PER_SEC;
 printf("Method 2 took %f seconds to execute \n", cpu_time_used);
 exit(0);
 getch();
}
```

ASSIGNMENT 4

Binary Search is an Example of DC technique, where we divide the input data (given in an indexed and sorted array) using its known size, and reduce the size of the problem

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i,a[9],key,point,low,high,mid;
for(i=1;i<=9;i++)
{
printf("Enter element");
scanf("%d",&a[i]);
}
printf("Enter key element");
scanf("%d",&key);
low=1;
high=9;
mid=(low+high)/2;
while(mid+1<high)
{
if(a[low]>a[mid])
{
low=low;
high=mid;
mid=(low+high)/2;
}
else if(a[high]<a[mid])
{
low=mid;
high=high;
mid=(low+high)/2;
}
}
if(a[mid]>a[mid+1])
point=mid;
else if(a[mid]<a[mid+1])
point=mid-1;
if(key<a[1])
{
low=point+1;
high=9;
}
else
{
low=1;
high=point;
}
mid=(low+high)/2;
while (low <= high) {
if(a[mid] < key)
low = mid + 1;
else if (a[mid] == key) {
printf("%d found at location %d", key, mid);
break;
}
else
high = mid - 1;
mid = (low + high)/2;
}
if(low>high)
printf("Not found! %d isn't present in the array", key);
getch();
return 0;
}
```

The time complexity of the code is O(log n), where n is the number of elements in the array.
The first loop to read in the array elements runs in O(n) time, as it iterates over each element in the array.
The second loop to find the pivot point also runs in O(log n) time, as it is a binary search algorithm that iteratively halves the search space until it finds the pivot point.
The third loop to search for the key also runs in O(log n) time, as it is another binary search algorithm that iteratively halves the search space until it finds the key or determines that the key is not present.
Overall, the time complexity of the algorithm is dominated by the binary search algorithms, which each have a time complexity of O(log n). Therefore, the time complexity of the entire algorithm is O(log n).

Tower of Hanoi problem using 4 pegs

ASSIGNMENT 4

```
#include <stdio.h>
#include <conio.h>
void towerOfHanoi(int n, char from_rod, char to_rod,char aux_rod1, char
aux_rod2)
{
if (n == 0)
return;
if (n == 1) {
printf("\n Move disk %d from rod %c to rod %c",n, from_rod, to_rod);
return;
}

towerOfHanoi(n - 2, from_rod, aux_rod1, aux_rod2,to_rod);
printf("\n Move disk %d from rod %c to rod %c ",n - 1, from_rod, aux_rod2);
printf("\n Move disk %d from rod %c to rod %c ",n, from_rod, to_rod);
printf("\n Move disk %d from rod %c to rod %c ",n - 1, aux_rod2, to_rod);
towerOfHanoi(n - 2, aux_rod1, to_rod, from_rod,aux_rod2);
}

// driver program
int main()
{
int n; // Number of disks
printf("Enter the number:");
scanf("%d",&n);
// A, B, C and D are names of rods
towerOfHanoi(n, 'A', 'D', 'B', 'C');
getch();
return 0;
}
```

The algorithm can be described as follows:
If n = 1, move the disk from the source peg to the destination peg using any available intermediate peg.
If n = 2, move the smallest disk from the source peg to an intermediate peg. Then move the largest disk from the source peg to the destination peg, using the remaining intermediate peg. Finally, move the smallest disk from the intermediate peg to the destination peg.
If n > 2, do the following: a. Move n-2 disks from the source peg to an intermediate peg, using the destination peg as the second intermediate peg. b. Move the remaining two disks from the source peg to the destination peg, using any available intermediate peg. c. Move the n-2 disks from the intermediate peg to the destination peg, using the source peg as the second intermediate peg.

The time complexity of the Tower of Hanoi problem using 4 pegs is O(2^n), where n is the number of disks.

ASSIGNMENT 5

Your second task is to write a fucntion that takes the randomly generated array from part A and sorts the array in linear time (i.e. in O(n)) and prints the sorted values.

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

// Generates and prints 'count' random
// numbers in range [lower, upper].
void printRandoms(int lower, int upper,
                  int count)
{
   int i;
   for (i = 0; i < count; i++) {
      int num = (rand() %
      (upper - lower + 1)) + lower;
      printf("%d ", num);
   }
}

int main()
{
   int lower, upper, count;

   // Read user input for lower, upper, and count
   printf("Enter lower limit: ");
   scanf("%d", &lower);

   printf("Enter upper limit: ");
   scanf("%d", &upper);

   printf("Enter number of random numbers to generate: ");
   scanf("%d", &count);

   // Use current time as
   // seed for random generator
   srand(time(0));

   printRandoms(lower, upper, count);

   return 0;
}
```

The time complexity of this code is O(n),

ASSIGNMENT 6

Implement the standard matrix chain multiplication problem

```
#include <stdio.h>
#include <stdlib.h>
 int s[7][7] = {0};
void print_optimal_parens(int i,int j){
   if(i==j){
      printf("A%d",i);
  }
else{
   printf("(");
   print_optimal_parens(i,s[i][j]);
   print_optimal_parens(s[i][j]+1,j);
   printf(")");
}
}
int main()
{
   int n =;
   printf("Enter the number of matrix");
   scanf("%d",&n);
   int p[n];
   for (int i = 0; i < n; i++)
   {
     scanf("%d", &p[i]);
   }
  int m[7][7] = {0};
  int j, min, q;
  for (int d = 1; d < n - 1; d++)
   {
      for (int i = 1; i < n - d; i++)
     {
        j = i + d;
        min = INT_MAX;
        for (int k = i; k < j; k++)
        {
          q = m[i][k] + m[k + 1][j] + p[i - 1] * p[j] * p[k];
           if (q < min)
             {
               min = q;
               s[i][j] = k;
              }
        }
         m[i][j] = min;
     }
  }
printf("%d", m[1][n - 1]);
printf(" \n   s matrix\n");
for (int i = 1; i < n-1 ; i++){
       for (int j = 1; j < n-1 ; j++){
          printf("%d", s[i][j]);
       }
       printf("\n");
      }
     for(int i = 1; i < n-1; i++){
      for(int j = i; j < n;j++){
          printf("%d", s[i][j]);
       }
      printf("\n");
    }
  print_optimal_parens(1,n-1);
   return 0;
}
```

The time complexity of this code is O(n^3), where n is the number of matrices.

ASSIGNMENT 7

time complexity of the given code is O(n * amount + 2^n),

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

int isAmountPossible(int amount, int i,
int *coins, int *counts, int **dp)
{
   if (amount < 0 || i < 0 || counts[i] < 0)
   {
      return 0;
   }
   if (!amount)
   {
      return 1;
   }
   if (dp[i][amount] != -1)
   {
      return dp[i][amount];
   }
   counts[i]--;
   int take = isAmountPossible(amount - coins[i], i, coins, counts, dp);
   counts[i]++;
   int notTake = isAmountPossible(amount, i - 1, coins, counts, dp);
   return dp[i][amount] = take || notTake;
}

int possibleWays(int amount, int i, int *coins, int *counts, int **dp)
{
   if (amount < 0 || i < 0 || counts[i] < 0)
   {
      return 0;
   }
   if (!amount)
   {
      return 1;
   }
   if (dp[i][amount] != -1)
   {
      return dp[i][amount];
   }
   counts[i]--;
   int take = possibleWays(amount - coins[i], i, coins, counts, dp);
   counts[i]++;
   int notTake = possibleWays(amount, i - 1, coins, counts, dp);
   return dp[i][amount] = take + notTake;
}

int leastCoins(int amount, int i, int *coins, int *counts, int **dp)
{
```

```
    if (amount < 0 || i < 0 || counts[i] < 0)
    {
        return 1e9;
    }
    if (!amount)
    {
        return 0;
    }
    if (dp[i][amount] != -1)
    {
        return dp[i][amount];
    }
    counts[i]--;
    int take = 1 + leastCoins(amount - coins[i], i, coins, counts, dp);
    counts[i]++;
    int notTake = leastCoins(amount, i - 1, coins, counts, dp);
    return dp[i][amount] = (take < notTake) ? take : notTake;
}

int greatestCoins(int amount, int i, int *coins, int *counts, int **dp)
{
    if (amount < 0 || i < 0 || counts[i] < 0)
    {
        return -1e9;
    }
    if (!amount)
    {
        return 0;
    }
    if (dp[i][amount] != -1)
    {
        return dp[i][amount];
    }
    counts[i]--;
    int take = 1 + greatestCoins(amount - coins[i], i, coins, counts, dp);
    counts[i]++;
    int notTake = greatestCoins(amount, i - 1, coins, counts, dp);
    return dp[i][amount] = (take > notTake) ? take : notTake;
}

void reassignDp(int **dp, int n1, int n2)
{
    for (int i = 0; i < n1; i++)
    {
        for (int j = 0; j < n2; j++)
        {
            dp[i][j] = -1;
        }
```

```
    }
}

int main()
{
    int n = 5, amount = 50 - 17;
    int coins[] = {1, 2, 5, 10, 20}, counts[] = {4, 2, 2, 2, 1};

    // printf("How many different coins are present? ");
    // int n;
    // scanf("%d", &n);

    // int *coins = (int *)malloc(sizeof(int) * n);
    // int *counts = (int *)malloc(sizeof(int) * n);
    // printf("What are them and their frequency?\n");
    // for (int i = 0; i < n; i++)
    // {
    //     scanf("%d", coins + i);
    //     scanf("%d", counts + i);
    // }

    // printf("What is the amount? ");
    // int amount;
    // scanf("%d", &amount);

    int **dp = (int **)malloc(sizeof(int *) * n);
    for (int i = 0; i < n; i++)
    {
        dp[i] = (int *)malloc(sizeof(int) * (amount + 1));
        for (int j = 0; j < amount + 1; j++)
        {
            dp[i][j] = -1;
        }
    }

    printf("The amount can%s be given\n", isAmountPossible(amount, n - 1, coins, counts, dp) ? "" :
"not");
    reassignDp(dp, n, amount + 1);
    printf("There are %d ways to give back the change\n", possibleWays(amount, n - 1, coins, counts,
dp));
    reassignDp(dp, n, amount + 1);
    printf("Least number of coins to give back the change is %d\n", leastCoins(amount, n - 1, coins,
counts, dp));
    reassignDp(dp, n, amount + 1);
    printf("Greatest number of coins to give back the change is %d\n", greatestCoins(amount, n - 1, coins,
counts, dp));

    return 0;
```