Linear search!

```c
#include <stdio.h>
int main()
{
  int array[100], search, c, n;

  printf("Enter number of elements in array\n");
  scanf("%d", &n);

  printf("Enter %d integer(s)\n", n);

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

  printf("Enter a number to search\n");
  scanf("%d", &search);

  for (c = 0; c < n; c++)
  {
    if (array[c] == search)
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d isn't present in the array.\n", search);

  return 0;
}
```

Binary search!

```c
#include <stdio.h>
int main()
{
int i, low, high, mid, n, key, array[100];
printf("Enter number of elementsn");
scanf("%d",&n);
printf("Enter %d integersn", n);
for(i = 0; i < n; i++)
scanf("%d",&array[i]);
printf("Enter value to findn");
```

```
scanf("%d", &key);
low = 0;
high = n - 1;
mid = (low+high)/2;
while (low <= high) {
if(array[mid] < key)
low = mid + 1;
else if (array[mid] == key) {
printf("%d found at location %d.n", key, mid+1);
break;
}
else
high = mid - 1;
mid = (low + high)/2;
}
if(low > high)
printf("Not found! %d isn't present in the list.n", key);
return 0;
}
```

Merge sort!

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

// merge function
void Merge(int arr[], int left, int mid, int right)
{
    int i, j, k;
    int size1 = mid - left + 1;
    int size2 = right - mid;

    // created temporary array
    int Left[size1], Right[size2];

    // copying the data from arr to temporary array
    for (i = 0; i < size1; i++)
        Left[i] = arr[left + i];
```

```
        for (j = 0; j < size2; j++)
                Right[j] = arr[mid + 1 + j];

    // merging of the array
        i = 0;  // intital index of first subarray
    j = 0;     // inital index of second subarray
    k = left; // initial index of parent array
    while (i < size1 && j < size2)
    {
                if (Left[i] <= Right[j])
        {
            arr[k] = Left[i];
            i++;
        }
        else
        {
            arr[k] = Right[j];
            j++;
        }
        k++;
    }

    // copying the elements from Left[], if any
    while (i < size1)
    {
        arr[k] = Left[i];
        i++;
        k++;
    }

    // copying the elements from Right[], if any
    while (j < size2)
    {
        arr[k] = Right[j];
        j++;
        k++;
    }
}

//merge sort function
void Merge_Sort(int arr[], int left, int right)
{
    if (left < right)
    {

        int mid = left + (right - left) / 2;
```

```
        // recursive calling of merge_sort
        Merge_Sort(arr, left, mid);
        Merge_Sort(arr, mid + 1, right);

        Merge(arr, left, mid, right);
    }
}

// driver code
int main()
{
    int size;
    printf("Enter the size: ");
    scanf("%d", &size);

    int arr[size];
    printf("Enter the elements of array: ");
    for (int i = 0; i < size; i++)
    {
        scanf("%d", &arr[i]);
    }

    Merge_Sort(arr, 0, size - 1);

    printf("The sorted array is: ");
    for (int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Quick sort!

```
/*
 * C Program to sort an array of integers using Quick Sort without recursion
 */

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

int quickSort(int *arr, int low, int high)
{
```

```
    int i = low, j = high;
    int pivot = arr[(low + high) / 2];
    while (i <= j)
    {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j)
        {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
            j--;
        }
    }
    if (low < j)
        quickSort(arr, low, j);
    if (i < high)
        quickSort(arr, i, high);
    return 0;
}

int main(void)
{
    puts("Enter the number of elements in the array: ");
    int n;
    scanf("%d", &n);
    int arr[n];
    puts("Enter the elements of the array: ");
    for (int i = 0; i < n; i++)
    {
        printf("arr[%d]: ", i);
        scanf("%d", &arr[i]);
    }
    int low = 0;
    int high = n - 1;
    int pivot = arr[high];
    int k = low - 1;
    for (int j = low; j < high; j++)
    {
        if (arr[j] <= pivot)
        {
            k++;
            int temp = arr[k];
```

```
            arr[k] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[k + 1];
    arr[k + 1] = arr[high];
    arr[high] = temp;
    int pi = k + 1;
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
    puts("The sorted array is: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Min and max using divide and conquer!

```
#include<stdio.h>
#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];
  }
 }
 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;
 }
}
}
int main ()
{
 int i, num;
 printf ("\nEnter the total number of numbers : ");
 scanf ("%d",&num);
 printf ("Enter the numbers : \n");
 for (i=1;i<=num;i++)
```

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


 max = a[0];
 min = a[0];
 maxmin(1, num);
 printf ("Minimum element in an array : %d\n", min);
 printf ("Maximum element in an array : %d\n", max);
 return 0;
}
```

DFS!

```
#include <stdio.h>
#include <stdlib.h>
intsourceV,Vertex,Edge,time,visited[100],Graph[100][100];
void DepthFirstSearch(inti)
{
int j;
visited[i]=1;
printf(" %d->",i++);
for(j=0;j<Vertex;j++)
{
if(Graph[i][j]==1&&visited[j]==0)
DepthFirstSearch(j);
}
}
intmain()
{
inti,j,vertex1,vertex2;
printf("\t\t\tGraphs\n");
printf("Enter no. of edges:");
scanf("%d",&Edge);
printf("Enter no. of vertices:");
scanf("%d",&Vertex);
for(i=0;i<Vertex;i++)
{
for(j=0;j<Vertex;j++)
Graph[i][j]=0;
}
```

```
for(i=0;i<Edge;i++)
{
printf("Enter the edges in V1 V2 : ");
scanf("%d%d",&vertex1,&vertex2);
Graph[vertex1-1][vertex2-1]=1;
}
for(i=0;i<Vertex;i++)
{
for(j=0;j<Vertex;j++)
printf(" %d ",Graph[i][j]);
printf("\n");
}
printf("Enter source Vertex: ");
scanf("%d",&sourceV);
DepthFirstSearch(sourceV-1);
return 0;
}
```

BFS!

```
#include <stdio.h>

int n, i, j, visited[10], queue[10], front = -1, rear = -1;
int adj[10][10];

void bfs(int v)
{
    for (i = 1; i <= n; i++)
        if (adj[v][i] && !visited[i])
            queue[++rear] = i;
    if (front <= rear)
    {
        visited[queue[front]] = 1;
        bfs(queue[front++]);
    }
}

void main()
{
    int v;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
```

```
    {
        queue[i] = 0;
        visited[i] = 0;
    }
    printf("Enter graph data in matrix form:    \n");
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            scanf("%d", &adj[i][j]);
    printf("Enter the starting vertex: ");
    scanf("%d", &v);
    bfs(v);
    printf("The node which are reachable are:    \n");
    for (i = 1; i <= n; i++)
        if (visited[i])
            printf("%d\t", i);
        else
            printf("BFS is not possible. Not all nodes are reachable");
    return 0;
}
```

KNAPSACK PROBLEM USING GREEDY APPROACH

```
//C Program to Simulate KnapSack Problem
// Code by Nived Kannada
#include<stdio.h>

void main ()
{
  int n, m, w[100], p[100], ratio[100] , i, j, u, temp;
  float xr, x[100], total_profit=0, total_weight=0;

  //Reading number of items
  printf ("Enter the number of items(n): ");
  scanf ("%d", &n);

  //Reading the capacity of the knapsack
  printf ("Enter the capacity of the Knapsack(m): ");
  scanf ("%d", &m);

  //Initializing remaining capacity of Knapsack (u)
  u = m;

  //Initializing Solution Array x[]
  for(i=0;i<n;i++)
  {
      x[i]=0;
```

```
}

//Reading the Weights
printf ("Enter the Weights of items: ");
for (i = 0; i < n; i++)
  {
    printf ("\n\tWeight of item %d = ", i + 1);
    scanf ("%d", &w[i]);
  }

//Reading the Profit values
printf ("\nEnter the Profit Values of items: ");
for (i = 0; i < n; i++)
  {
    printf ("\n\tProfit of item %d = ", i + 1);
    scanf ("%d", &p[i]);
  }

//Calculating Pi/Wi ratio of each item and storing in array ratio[]
for (i = 0; i < n; i++)
  {
    ratio[i] = p[i] / w[i];
  }

//Sorting all the arrays based on the ratio in descending order
for (i = 0; i < n; i++)
  {
    for (j = 0; j < n - 1; j++)
     {
       if (ratio[j] < ratio[i])
         {
           temp = ratio[i];
           ratio[i] = ratio[j];
           ratio[j] = temp;

           temp = w[i];
           w[i] = w[j];
           w[j] = temp;

           temp = p[i];
           p[i] = p[j];
           p[j] = temp;
         }
     }
  }

  //PRINTING THE SORTED TABLE
  printf("\n The Table After Sorting based on the Ratio: \n");

  //Printing Item numbers
  printf("\nItem:\t\t");
  for(i=0;i<n;i++)
  {
```

```
        printf("%d\t",i+1);
    }

    //Printing Profit Array
    printf("\nProfit:\t\t");
    for(i=0;i<n;i++)
    {
        printf("%d\t",p[i]);
    }

    //Printing Weight Array
    printf("\nWeights:\t");
    for(i=0;i<n;i++)
    {
        printf("%d\t",w[i]);
    }


    //Printing RATIO Array
    printf ("\nRATIO:\t\t");
    for (i = 0; i < n; i++)
    {
      printf ("%d\t", ratio[i]);
    }

    //Calculating Solution Array x
    for(i=0;i<n;i++)
    {
        if(w[i]<=u)
        {
            x[i]=1;      //Setting solution index as 1
            u=u-w[i];    //updating remaining knapsack capacity
        }
        else if(w[i]>u)
        {
            break;
        }
    }

    if(i<=n)
    {
        xr = (float)u/w[i];     //Calculating what fraction of that item
will fit into the knapsack
        x[i] = xr;       //Setting this fraction to solution array
    }

    //Printing Solution Array x
    printf("\n X = [");
    for(i=0;i<n;i++)
    {
        printf("%.3f , ",x[i]);
    }
    printf("]");
```

```
    //Calculating Total Profit & Total Weight
    for(i=0;i<n;i++)
    {
        total_profit += x[i]*p[i];
        total_weight += x[i]*w[i];
    }

    //Displaying Total Profit and Total Weight
    printf("\nTotal   Profit   =   %.2f   \n   Total   Weight   =   %.2f
",total_profit,total_weight);


}
```

KRUSKAL'S ALGORITHM

```
#include<stdio.h>

#define MAX 30

typedef struct edge
{
int u,v,w;
}edge;

typedef struct edgelist
{
edge data[MAX];
int n;
}edgelist;

edgelist elist;

int G[MAX][MAX],n;
edgelist spanlist;

void kruskal();
int find(int belongs[],int vertexno);
void union1(int belongs[],int c1,int c2);
void sort();
void print();

void main()
{
int i,j,total_cost;
printf("\nEnter number of vertices:");
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&G[i][j]);
kruskal();
print();
}
```

```
void kruskal()
{
int belongs[MAX],i,j,cno1,cno2;
elist.n=0;

for(i=1;i<n;i++)
for(j=0;j<i;j++)
{
if(G[i][j]!=0)
{
elist.data[elist.n].u=i;
elist.data[elist.n].v=j;
elist.data[elist.n].w=G[i][j];
elist.n++;
}
}

sort();
for(i=0;i<n;i++)
belongs[i]=i;
spanlist.n=0;
for(i=0;i<elist.n;i++)
{
cno1=find(belongs,elist.data[i].u);
cno2=find(belongs,elist.data[i].v);
if(cno1!=cno2)
{
spanlist.data[spanlist.n]=elist.data[i];
spanlist.n=spanlist.n+1;
union1(belongs,cno1,cno2);
}
}
}

int find(int belongs[],int vertexno)
{
return(belongs[vertexno]);
}

void union1(int belongs[],int c1,int c2)
{
int i;
for(i=0;i<n;i++)
if(belongs[i]==c2)
belongs[i]=c1;
}

void sort()
{
int i,j;
edge temp;
for(i=1;i<elist.n;i++)
for(j=0;j<elist.n-1;j++)
if(elist.data[j].w>elist.data[j+1].w)
{
temp=elist.data[j];
elist.data[j]=elist.data[j+1];
elist.data[j+1]=temp;
}
}

void print()
{
int i,cost=0;
for(i=0;i<spanlist.n;i++)
{
```

```
printf("\n%d\t%d\t%d",spanlist.data[i].u,spanlist.data[i].v,spanlist.data[i].w);
cost=cost+spanlist.data[i].w;
}

printf("\n\nCost of the spanning tree=%d",cost);
}
```

PRIM'S ALGORITHM

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

#define infinity 9999
#define MAX 20

int G[MAX][MAX],spanning[MAX][MAX],n;

int prims();

int main()
{
int i,j,total_cost;
printf("Enter no. of vertices:");
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&G[i][j]);
total_cost=prims();
printf("\nspanning tree matrix:\n");
for(i=0;i<n;i++)
{
printf("\n");
for(j=0;j<n;j++)
printf("%d\t",spanning[i][j]);
}
printf("\n\nTotal cost of spanning tree=%d",total_cost);
return 0;
}

int prims()
{
int cost[MAX][MAX];
int u,v,min_distance,distance[MAX],from[MAX];
int visited[MAX],no_of_edges,i,min_cost,j;
//create cost[][] matrix,spanning[][]
for(i=0;i<n;i++)
for(j=0;j<n;j++)
{
if(G[i][j]==0)
cost[i][j]=infinity;
else
cost[i][j]=G[i][j];
spanning[i][j]=0;
}
//initialise visited[],distance[] and from[]
distance[0]=0;
visited[0]=1;
for(i=1;i<n;i++)
{
distance[i]=cost[0][i];
from[i]=0;
visited[i]=0;
```

```
}
min_cost=0; //cost of spanning tree
no_of_edges=n-1; //no. of edges to be added
while(no_of_edges>0)
{
//find the vertex at minimum distance from the tree
min_distance=infinity;
for(i=1;i<n;i++)
if(visited[i]==0&&distance[i]<min_distance)
{
v=i;
min_distance=distance[i];
}
u=from[v];
//insert the edge in spanning tree
spanning[u][v]=distance[v];
spanning[v][u]=distance[v];
no_of_edges--;
visited[v]=1;
//updated the distance[] array
for(i=1;i<n;i++)
if(visited[i]==0&&cost[i][v]<distance[i])
{
distance[i]=cost[i][v];
from[i]=v;
}
min_cost=min_cost+cost[u][v];
}
return(min_cost);
}
```

MATRIX CHAIN MULTIPLICATION

```
#include <stdio.h>
#include<limits.h>
#define INFY 999999999
long int m[20][20];
int s[20][20];
int p[20],i,j,n;

void print_optimal(int i,int j)
{
if (i == j)
printf(" A%d ",i);
else
   {
      printf("( ");
      print_optimal(i, s[i][j]);
      print_optimal(s[i][j] + 1, j);
      printf(" )");
   }
}

void matmultiply(void)
{
long int q;
int k;
for(i=n;i>0;i--)
 {
   for(j=i;j<=n;j++)
    {
```

```
     if(i==j)
       m[i][j]=0;
     else
       {
        for(k=i;k<j;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;
            s[i][j]=k;
          }
         }
        }
      }
 }
}

int MatrixChainOrder(int p[], int i, int j)
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    for (k = i; k <j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

void main()
{
int k;
printf("Enter the no. of elements: ");
scanf("%d",&n);
for(i=1;i<=n;i++)
for(j=i+1;j<=n;j++)
{
 m[i][i]=0;
 m[i][j]=INFY;
 s[i][j]=0;
}
printf("\nEnter the dimensions: \n");
for(k=0;k<=n;k++)
{
 printf("P%d: ",k);
 scanf("%d",&p[k]);
}
matmultiply();
printf("\nCost Matrix M:\n");
for(i=1;i<=n;i++)
```

```
 for(j=i;j<=n;j++)
  printf("m[%d][%d]: %ld\n",i,j,m[i][j]);


i=1,j=n;
printf("\nMultiplication Sequence : ");
print_optimal(i,j);
printf("\nMinimum number of multiplications is : %d ",
                          MatrixChainOrder(p, 1, n));

}
```

DIJKSTRA'S ALGORITHM

```
 #include<stdio.h>
                    #include<conio.h>
                    #define INFINITY 9999
                    #define MAX 10

                    void dijkstra(int G[MAX][MAX],int n,int startnode);

                    int main()
                    {
                           int G[MAX][MAX],i,j,n,u;
                           printf("Enter no. of vertices:");
                           scanf("%d",&n);
                           printf("\nEnter the adjacency matrix:\n");

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

                           printf("\nEnter the starting node:");
                           scanf("%d",&u);
                           dijkstra(G,n,u);

                           return 0;
                    }

                    void dijkstra(int G[MAX][MAX],int n,int startnode)
                    {

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

                           //pred[] stores the predecessor of each node
                           //count gives the number of nodes seen so far
```

```
	//create the cost matrix
	for(i=0;i<n;i++)
		for(j=0;j<n;j++)
			if(G[i][j]==0)
				cost[i][j]=INFINITY;
			else
				cost[i][j]=G[i][j];

	//initialize pred[],distance[] and visited[]
	for(i=0;i<n;i++)
	{
		distance[i]=cost[startnode][i];
		pred[i]=startnode;
		visited[i]=0;
	}

	distance[startnode]=0;
	visited[startnode]=1;
	count=1;

	while(count<n-1)
	{
		mindistance=INFINITY;

		//nextnode gives the node at minimum distance
		for(i=0;i<n;i++)
			if(distance[i]<mindistance&&!visited[i])
			{
				mindistance=distance[i];
				nextnode=i;
			}

			//check if a better path exists through
nextnode
			visited[nextnode]=1;
			for(i=0;i<n;i++)
				if(!visited[i])

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

	distance[i]=mindistance+cost[nextnode][i];
						pred[i]=nextnode;
					}
		count++;
	}
```

```
                //print the path and distance of each node
                for(i=0;i<n;i++)
                        if(i!=startnode)
                        {
                                printf("\nDistance of
        node%d=%d",i,distance[i]);
                                printf("\nPath=%d",i);

                                j=i;
                                do
                                {
                                        j=pred[j];
                                        printf("<-%d",j);
                                }while(j!=startnode);
                        }
                }
```

N QUEENSS

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

            bool isSafe(int n, int board[][n], int row, int col)
            {
                int i, j;

                for (i = 0; i < col; i++)
                    if (board[row][i])
                        return false;

                for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
                    if (board[i][j])
                        return false;

                for (i = row, j = col; j >= 0 && i < n; i++, j--)
                    if (board[i][j])
                        return false;

                return true;
            }

            void printSolution(int n, int board[][n])
```

```
{
    static int solCount = 0;
    printf("Solution %d:\n", ++solCount);
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            printf("%d ", board[i][j]);
        printf("\n");
    }
    printf("\n");
}

void solveNQueensUtil(int n, int board[][n], int col)
{
    if (col >= n)
    {
        printSolution(n, board);
        return;
    }

    for (int i = 0; i < n; i++)
    {
        if (isSafe(n, board, i, col))
        {
            board[i][col] = 1;
            solveNQueensUtil(n, board, col + 1);
            board[i][col] = 0;
        }
    }
}

void solveNQueens(int n)
{
    int board[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            board[i][j] = 0;

    solveNQueensUtil(n, board, 0);
}

int main()
{
    int n;
    printf("N-Queen Problem in C\n");
    printf("Enter the value of N: ");
```

```
    scanf("%d", &n);
    solveNQueens(n);
    return 0;
}
```

Implementing knapsack problem using branch and bound technique

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

typedef enum { NO, YES } BOOL;

int N;
int vals[100];
int wts[100];

int cap = 0;
int mval = 0;

void getWeightAndValue (BOOL incl[N], int *weight, int *value) {
	int i, w = 0, v = 0;
	for (i = 0; i < N; ++i) {
		if (incl[i]) {
			w += wts[i];
			v += vals[i];
		}
	}
	*weight = w;
	*value = v;
}

void printSubset (BOOL incl[N]) {
	int i;
	int val = 0;
	printf("Included = { ");
	for (i = 0; i < N; ++i) {
		if (incl[i]) {
			printf("%d ", wts[i]);
			val += vals[i];
		}
	}
```

```
        printf("}; Total value = %d\n", val);
}

void findKnapsack (BOOL incl[N], int i) {
        int cwt, cval;
        getWeightAndValue(incl, &cwt, &cval);
        if (cwt <= cap) {
                if (cval > mval) {
                        printSubset(incl);
                        mval = cval;
                }
        }
        if (i == N || cwt >= cap) {
                return;
        }
        int x = wts[i];
        BOOL use[N], nouse[N];
        memcpy(use, incl, sizeof(use));
        memcpy(nouse, incl, sizeof(nouse));
        use[i] = YES;
        nouse[i] = NO;
        findKnapsack(use, i+1);
        findKnapsack(nouse, i+1);
}

int main(int argc, char const * argv[]) {
        printf("Enter the number of elements: ");
        scanf(" %d", &N);
        BOOL incl[N];
        int i;
        for (i = 0; i < N; ++i) {
                printf("Enter weight and value for element %d: ", i+1);
                scanf(" %d %d", &wts[i], &vals[i]);
                incl[i] = NO;
        }
        printf("Enter knapsack capacity: ");
        scanf(" %d", &cap);
        findKnapsack(incl, 0);
        return 0;
}
```

Floyd warshall

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

```
#include <limits.h>

#define V 5
#define INF INT_MAX

void floydWarshall(int graph[V][V])
{
    int dist[V][V];
    int i, j, k;

    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    for (k = 0; k < V; k++)
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] +
dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];

    printf("Shortest path matrix:\n");
    for (i = 0; i < V; i++)
    {
        for (j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("INF\t");
            else
                printf("%d\t", dist[i][j]);
        }
        printf("\n");
    }
}

int main()
{
    int graph[V][V] = {{0, 3, 8, INF, -4},
                       {INF, 0, INF, 1, 7},
                       {INF, 4, 0, INF, INF},
                       {2, INF, -5, 0, INF},
                       {INF, INF, INF, 6, 0}};

    printf("Floyd Warshall algorithm in C\n");
    floydWarshall(graph);
    return 0;
```

```
}
```

Strassen n x n

```
#include <stdio.h>

#include <stdlib.h>


void strassen(int n, int **A, int **B, int **C);

void add(int n, int **A, int **B, int **C);

void subtract(int n, int **A, int **B, int **C);

void allocate_matrix(int n, int ***A);

void free_matrix(int n, int ***A);

void print_matrix(int n, int **A);


int main() {

   int n = 4;

   int **A, **B, **C;

   allocate_matrix(n, &A);

   allocate_matrix(n, &B);

   allocate_matrix(n, &C);


   A[0][0] = 1; A[0][1] = 2; A[0][2] = 3; A[0][3] = 4;

   A[1][0] = 5; A[1][1] = 6; A[1][2] = 7; A[1][3] = 8;

   A[2][0] = 9; A[2][1] = 10; A[2][2] = 11; A[2][3] = 12;

   A[3][0] = 13; A[3][1] = 14; A[3][2] = 15; A[3][3] = 16;


   B[0][0] = 17; B[0][1] = 18; B[0][2] = 19; B[0][3] = 20;

   B[1][0] = 21; B[1][1] = 22; B[1][2] = 23; B[1][3] = 24;
```

```
    B[2][0] = 25; B[2][1] = 26; B[2][2] = 27; B[2][3] = 28;
    B[3][0] = 29; B[3][1] = 30; B[3][2] = 31; B[3][3] = 32;

    strassen(n, A, B, C);
    print_matrix(n, C);

    free_matrix(n, &A);
    free_matrix(n, &B);
    free_matrix(n, &C);
    return 0;
}

void strassen(int n, int **A, int **B, int **C) {
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
        return;
    }

    int i, j;
    int **a11, **a12, **a21, **a22;
    int **b11, **b12, **b21, **b22;
    int **c11, **c12, **c21, **c22;
    int **p1, **p2, **p3, **p4, **p5, **p6, **p7;

    // Allocate memory for submatrices
    allocate_matrix(n/2, &a11);
    allocate_matrix(n/2, &a12);
```

```
    allocate_matrix(n/2, &a21);

    allocate_matrix(n/2, &a22);



    allocate_matrix(n/2, &b11);

    allocate_matrix(n/2, &b12);

  allocate_matrix(n/2, &b21);
    allocate_matrix(n/2, &b22);

    allocate_matrix(n/2, &c11);
    allocate_matrix(n/2, &c12);
    allocate_matrix(n/2, &c21);
    allocate_matrix(n/2, &c22);

    allocate_matrix(n/2, &p1);

      allocate_matrix(n/2, &p2);

allocate_matrix(n/2, &p3);
    allocate_matrix(n/2, &p4);
    allocate_matrix(n/2, &p5);
    allocate_matrix(n/2, &p6);
    allocate_matrix(n/2, &p7);

    // Divide A and B into 4 submatrices
    for (i = 0; i < n/2; i++) {
        for (j = 0; j < n/2; j++) {
            a11[i][j] = A[i][j];
            a12[i][j] = A[i][j + n/2];
            a21[i][j] = A[i + n/2][j];
            a22[i][j] = A[i + n/2][j + n/2];
            b11[i][j] = B[i][j];

            b12[i][j] = B[i][j + n/2];
            b21[i][j] = B[i + n/2][j];
            b22[i][j] = B[i + n/2][j + n/2];
        }
    }

    // Compute the 7 products (p1 to p7) using Strassen's algorithm
    subtract(n/2, b12, b22, c11);
    strassen(n/2, a11, c11, p1);

    add(n/2, a11, a12, c11);

      strassen(n/2, c11, b22, p2);

subtract(n/2, a21, a11, c11);
    add(n/2, b11, b12, c12);
    strassen(n/2, c11, c12, p3);
subtract(n/2, a12, a22, c11);
```

```
    add(n/2, b21, b22, c12);
    strassen(n/2, c11, c12, p4);

    add(n/2, a11, a22, c11);

    subtract(n/2, a12, a22, c11);
    add(n/2, b21, b22, c12);
    strassen(n/2, c11, c12, p4);

    add(n/2, a11, a22, c11);
    add(n/2, b21, b22, c12);

strassen(n/2, c11, c12, p6);

subtract(n/2, a11, a21, c11);
add(n/2, b11, b12, c12);
strassen(n/2, c11, c12, p7);

// Compute submatrices of C using Strassen's algorithm
add(n/2, p1, p4, c11);
subtract(n/2, c11, p5, c11);
add(n/2, c11, p7, c11);

c12 = p3 + p5;
c21 = p2 + p4;

add(n/2, p1, p3, c11);
subtract(n/2, c11, p2, c11);
add(n/2, c11, p6, c22);

// Free memory for submatrices
free_matrix(n/2, &a11);
free_matrix(n/2, &a12);
free_matrix(n/2, &a21);
free_matrix(n/2, &a22);

free_matrix(n/2, &b11);
free_matrix(n/2, &b12);
free_matrix(n/2, &b21);
free_matrix(n/2, &b22);

free_matrix(n/2, &p1);
free_matrix(n/2, &p2);
free_matrix(n/2, &p3);
free_matrix(n/2, &p4);
free_matrix(n/2, &p5);
free_matrix(n/2, &p6);
free_matrix(n/2, &p7);
```

```
}

void add(int n, int **A, int **B, int **C) { int i, j; for (i = 0; i < n; ++i) { for (j = 0; j < n;
++j) { C[i][j] = A[i][j] + B[i][j]; } } }
```

```
void subtract(int n, int **A, int **B, int **C) { int i, j; for (i = 0; i < n; ++i) { for (j = 0; j < n; ++j) {

C[i][j] = A[i][j] - B[i][j]; } } }

void allocate_matrix(int n, int ***A) { int i; *A = (int **)malloc(n * sizeof(int *)); for (i = 0; i < n; ++i) { (*A)[i] = (int *)malloc(n * sizeof(int)); } }

void free_matrix(int n, int ***A) {

int i; for (i = 0; i < n; ++i) { free((*A)[i]); } free(*A); }

void print_matrix(int n, int **A) { int i, j; for (i = 0; i < n; ++i) { for (j = 0; j < n; ++j) { printf("%d ", A[i][j]); } printf("\n");
} }
```

```
// Write a C
program to
implement
Strassen’s
matrix
multiplication.
4 X 4
```

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

void strassen(int n, int A[][n], int B[][n], int C[][n]);

void add(int n, int A[][n], int B[][n], int C[][n]);
void subtract(int n, int A[][n], int B[][n], int C[][n]);

int main()
{
    int n = 4, i, j;
    int A[n][n], B[n][n], C[n][n];

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            A[i][j] = i + j;
            B[i][j] = 2 * A[i][j];
        }
    }

    printf("Strassen's Matrix Multiplication in C\n", n, n);
```

```
    printf("Order of both matrices: %d * %d\n", n, n);
    strassen(n, A, B, C);

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

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

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

    return 0;
}

void strassen(int n, int A[][n], int B[][n], int C[][n])
{
    if (n == 1)
    {
        C[0][0] = A[0][0] * B[0][0];
        return;
    }

    int i, j;

    int A11[n / 2][n / 2], A12[n / 2][n / 2], A21[n / 2][n / 2], A22[n
/ 2][n / 2];
    int B11[n / 2][n / 2], B12[n / 2][n / 2], B21[n / 2][n / 2], B22[n
/ 2][n / 2];
    int C11[n / 2][n / 2], C12[n / 2][n / 2], C21[n / 2][n / 2], C22[n
/ 2][n / 2];
```

```
    int P1[n / 2][n / 2], P2[n / 2][n / 2], P3[n / 2][n / 2], P4[n /
2][n / 2], P5[n / 2][n / 2], P6[n / 2][n / 2], P7[n / 2][n / 2];
    int temp1[n / 2][n / 2], temp2[n / 2][n / 2];

    // Divide A & B into 4 submatrices
    for (i = 0; i < n / 2; i++)
    {
        for (j = 0; j < n / 2; j++)
        {
            A11[i][j] = A[i][j];
            B11[i][j] = B[i][j];
        }
    }
    for (i = 0; i < n / 2; i++)
    {
        for (j = n / 2; j < n; j++)
        {
            A12[i][j - n / 2] = A[i][j];
            B12[i][j - n / 2] = B[i][j];
        }
    }
    for (i = n / 2; i < n; i++)
    {
        for (j = 0; j < n / 2; j++)
        {
            A21[i - n / 2][j] = A[i][j];
            B21[i - n / 2][j] = B[i][j];
        }
    }
    for (i = n / 2; i < n; i++)
    {
        for (j = n / 2; j < n; j++)
        {
            A22[i - n / 2][j - n / 2] = A[i][j];
            B22[i - n / 2][j - n / 2] = B[i][j];
        }
    }

    // Calculate the 7 products
    add(n / 2, A11, A22, temp1);
    add(n / 2, B11, B22, temp2);
    strassen(n / 2, temp1, temp2, P1);

    add(n / 2, A21, A22, temp1);
    strassen(n / 2, temp1, B11, P2);
```

```
    subtract(n / 2, B12, B22, temp1);
    strassen(n / 2, A11, temp1, P3);

    subtract(n / 2, B21, B11, temp1);
    strassen(n / 2, A22, temp1, P4);

    add(n / 2, A11, A12, temp1);
    strassen(n / 2, temp1, B22, P5);

    subtract(n / 2, A21, A11, temp1);
    add(n / 2, B11, B12, temp2);
    strassen(n / 2, temp1, temp2, P6);

    subtract(n / 2, A12, A22, temp1);
    add(n / 2, B21, B22, temp2);
    strassen(n / 2, temp1, temp2, P7);

    // Calculate the 4 quadrants of C using the products
    add(n / 2, P1, P4, temp1);
    subtract(n / 2, temp1, P5, temp2);
    add(n / 2, temp2, P7, C11);

    add(n / 2, P3, P5, C12);

    add(n / 2, P2, P4, C21);

    add(n / 2, P1, P3, temp1);
    subtract(n / 2, temp1, P2, temp2);
    add(n / 2, temp2, P6, C22);

    // Combine the 4 quadrants of C into one matrix
    for (i = 0; i < n / 2; i++)
        for (j = 0; j < n / 2; j++)
            C[i][j] = C11[i][j];
    for (i = 0; i < n / 2; i++)
        for (j = n / 2; j < n; j++)
            C[i][j] = C12[i][j - n / 2];
    for (i = n / 2; i < n; i++)
        for (j = 0; j < n / 2; j++)
            C[i][j] = C21[i - n / 2][j];
    for (i = n / 2; i < n; i++)
        for (j = n / 2; j < n; j++)
            C[i][j] = C22[i - n / 2][j - n / 2];
}

void add(int n, int A[][n], int B[][n], int C[][n])
```

```
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            C[i][j] = A[i][j] + B[i][j];
}

void subtract(int n, int A[][n], int B[][n], int C[][n])
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
}
```