The [Floyd Warshall Algorithm](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm) is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

#include<stdio.h>

// Number of vertices in the graph

#define V 4

/\* Define Infinite as a large enough value. This value will be used

  for vertices not connected to each other \*/

#define INF 99999

// A function to print the solution matrix

void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm

void floydWarshall (int graph[][V])

{

    /\* dist[][] will be the output matrix that will finally have the shortest

      distances between every pair of vertices \*/

    int dist[V][V], i, j, k;

    /\* Initialize the solution matrix same as input graph matrix. Or

       we can say the initial values of shortest distances are based

       on shortest paths considering no intermediate vertex. \*/

    for (i = 0; i < V; i++)

        for (j = 0; j < V; j++)

            dist[i][j] = graph[i][j];

    /\* Add all vertices one by one to the set of intermediate vertices.

      ---> Before start of a iteration, we have shortest distances between all

      pairs of vertices such that the shortest distances consider only the

      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.

      ----> After the end of a iteration, vertex no. k is added to the set of

      intermediate vertices and the set becomes {0, 1, 2, .. k} \*/

    for (k = 0; k < V; k++)

    {

        // Pick all vertices as source one by one

        for (i = 0; i < V; i++)

        {

            // Pick all vertices as destination for the

            // above picked source

            for (j = 0; j < V; j++)

            {

                // If vertex k is on the shortest path from

                // i to j, then update the value of dist[i][j]

                if (dist[i][k] + dist[k][j] < dist[i][j])

                    dist[i][j] = dist[i][k] + dist[k][j];

            }

        }

    }

    // Print the shortest distance matrix

    printSolution(dist);

}

/\* A utility function to print solution \*/

void printSolution(int dist[][V])

{

    printf ("Following matrix shows the shortest distances"

            " between every pair of vertices \n");

    for (int i = 0; i < V; i++)

    {

        for (int j = 0; j < V; j++)

        {

            if (dist[i][j] == INF)

                printf("%7s", "INF");

            else

                printf ("%7d", dist[i][j]);

        }

        printf("\n");

    }

}

// driver program to test above function

int main()

{

    /\* Let us create the following weighted graph

            10

       (0)------->(3)

        |         /|\

      5 |          |

        |          | 1

       \|/         |

       (1)------->(2)

            3           \*/

    int graph[V][V] = { {0,   5,  INF, 10},

                        {INF, 0,   3, INF},

                        {INF, INF, 0,   1},

                        {INF, INF, INF, 0}

                      };

    // Print the solution

    floydWarshall(graph);

    return 0;

}

**PRIM:**

/ A C / C++ program for Prim's Minimum Spanning Tree (MST) algorithm.

// The program is for adjacency matrix representation of the graph

#include <stdio.h>

#include <limits.h>

// Number of vertices in the graph

#define V 5

// A utility function to find the vertex with minimum key value, from

// the set of vertices not yet included in MST

int minKey(int key[], bool mstSet[])

{

   // Initialize min value

   int min = INT\_MAX, min\_index;

   for (int v = 0; v < V; v++)

     if (mstSet[v] == false && key[v] < min)

         min = key[v], min\_index = v;

   return min\_index;

}

// A utility function to print the constructed MST stored in parent[]

int printMST(int parent[], int n, int graph[V][V])

{

   printf("Edge   Weight\n");

   for (int i = 1; i < V; i++)

      printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);

}

// Function to construct and print MST for a graph represented using adjacency

// matrix representation

void primMST(int graph[V][V])

{

     int parent[V]; // Array to store constructed MST

     int key[V];   // Key values used to pick minimum weight edge in cut

     bool mstSet[V];  // To represent set of vertices not yet included in MST

     // Initialize all keys as INFINITE

     for (int i = 0; i < V; i++)

        key[i] = INT\_MAX, mstSet[i] = false;

     // Always include first 1st vertex in MST.

     key[0] = 0;     // Make key 0 so that this vertex is picked as first vertex

     parent[0] = -1; // First node is always root of MST

     // The MST will have V vertices

     for (int count = 0; count < V-1; count++)

     {

        // Pick thd minimum key vertex from the set of vertices

        // not yet included in MST

        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set

        mstSet[u] = true;

        // Update key value and parent index of the adjacent vertices of

        // the picked vertex. Consider only those vertices which are not yet

        // included in MST

        for (int v = 0; v < V; v++)

           // graph[u][v] is non zero only for adjacent vertices of m

           // mstSet[v] is false for vertices not yet included in MST

           // Update the key only if graph[u][v] is smaller than key[v]

          if (graph[u][v] && mstSet[v] == false && graph[u][v] <  key[v])

             parent[v]  = u, key[v] = graph[u][v];

     }

     // print the constructed MST

     printMST(parent, V, graph);

}

// driver program to test above function

int main()

{

   /\* Let us create the following graph

          2    3

      (0)--(1)--(2)

       |   / \   |

      6| 8/   \5 |7

       | /     \ |

      (3)-------(4)

            9          \*/

   int graph[V][V] = {{0, 2, 0, 6, 0},

                      {2, 0, 3, 8, 5},

                      {0, 3, 0, 0, 7},

                      {6, 8, 0, 0, 9},

                      {0, 5, 7, 9, 0},

                     };

    // Print the solution

    primMST(graph);

    return 0;

}

**KRUSCARL:**

// C++ program for Kruskal's algorithm to find Minimum Spanning Tree

// of a given connected, undirected and weighted graph

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// a structure to represent a weighted edge in graph

struct Edge

{

    int src, dest, weight;

};

// a structure to represent a connected, undirected and weighted graph

struct Graph

{

    // V-> Number of vertices, E-> Number of edges

    int V, E;

    // graph is represented as an array of edges. Since the graph is

    // undirected, the edge from src to dest is also edge from dest

    // to src. Both are counted as 1 edge here.

    struct Edge\* edge;

};

// Creates a graph with V vertices and E edges

struct Graph\* createGraph(int V, int E)

{

    struct Graph\* graph = (struct Graph\*) malloc( sizeof(struct Graph) );

    graph->V = V;

    graph->E = E;

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

    return graph;

}

// A structure to represent a subset for union-find

struct subset

{

    int parent;

    int rank;

};

// A utility function to find set of an element i

// (uses path compression technique)

int find(struct subset subsets[], int i)

{

    // find root and make root as parent of i (path compression)

    if (subsets[i].parent != i)

        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;

}

// A function that does union of two sets of x and y

// (uses union by rank)

void Union(struct subset subsets[], int x, int y)

{

    int xroot = find(subsets, x);

    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree

    // (Union by Rank)

    if (subsets[xroot].rank < subsets[yroot].rank)

        subsets[xroot].parent = yroot;

    else if (subsets[xroot].rank > subsets[yroot].rank)

        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment

    // its rank by one

    else

    {

        subsets[yroot].parent = xroot;

        subsets[xroot].rank++;

    }

}

// Compare two edges according to their weights.

// Used in qsort() for sorting an array of edges

int myComp(const void\* a, const void\* b)

{

    struct Edge\* a1 = (struct Edge\*)a;

    struct Edge\* b1 = (struct Edge\*)b;

    return a1->weight > b1->weight;

}

// The main function to construct MST using Kruskal's algorithm

void KruskalMST(struct Graph\* graph)

{

    int V = graph->V;

    struct Edge result[V];  // Tnis will store the resultant MST

    int e = 0;  // An index variable, used for result[]

    int i = 0;  // An index variable, used for sorted edges

    // Step 1:  Sort all the edges in non-decreasing order of their weight

    // If we are not allowed to change the given graph, we can create a copy of

    // array of edges

    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V ssubsets

    struct subset \*subsets =

        (struct subset\*) malloc( V \* sizeof(struct subset) );

    // Create V subsets with single elements

    for (int v = 0; v < V; ++v)

    {

        subsets[v].parent = v;

        subsets[v].rank = 0;

    }

    // Number of edges to be taken is equal to V-1

    while (e < V - 1)

    {

        // Step 2: Pick the smallest edge. And increment the index

        // for next iteration

        struct Edge next\_edge = graph->edge[i++];

        int x = find(subsets, next\_edge.src);

        int y = find(subsets, next\_edge.dest);

        // If including this edge does't cause cycle, include it

        // in result and increment the index of result for next edge

        if (x != y)

        {

            result[e++] = next\_edge;

            Union(subsets, x, y);

        }

        // Else discard the next\_edge

    }

    // print the contents of result[] to display the built MST

    printf("Following are the edges in the constructed MST\n");

    for (i = 0; i < e; ++i)

        printf("%d -- %d == %d\n", result[i].src, result[i].dest,

                                                   result[i].weight);

    return;

}

// Driver program to test above functions

int main()

{

    /\* Let us create following weighted graph

             10

        0--------1

        |  \     |

       6|   5\   |15

        |      \ |

        2--------3

            4       \*/

    int V = 4;  // Number of vertices in graph

    int E = 5;  // Number of edges in graph

    struct Graph\* graph = createGraph(V, E);

    // add edge 0-1

    graph->edge[0].src = 0;

    graph->edge[0].dest = 1;

    graph->edge[0].weight = 10;

    // add edge 0-2

    graph->edge[1].src = 0;

    graph->edge[1].dest = 2;

    graph->edge[1].weight = 6;

    // add edge 0-3

    graph->edge[2].src = 0;

    graph->edge[2].dest = 3;

    graph->edge[2].weight = 5;

    // add edge 1-3

    graph->edge[3].src = 1;

    graph->edge[3].dest = 3;

    graph->edge[3].weight = 15;

    // add edge 2-3

    graph->edge[4].src = 2;

    graph->edge[4].dest = 3;

    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;

}

**TOPOLOGICAL SORT:**

// A C++ program to print topological sorting of a DAG

#include<iostream>

#include <list>

#include <stack>

using namespace std;

// Class to represent a graph

class Graph

{

    int V;    // No. of vertices'

    // Pointer to an array containing adjacency listsList

    list<int> \*adj;

    // A function used by topologicalSort

    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);

public:

    Graph(int V);   // Constructor

     // function to add an edge to graph

    void addEdge(int v, int w);

    // prints a Topological Sort of the complete graph

    void topologicalSort();

};

Graph::Graph(int V)

{

    this->V = V;

    adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

    adj[v].push\_back(w); // Add w to v’s list.

}

// A recursive function used by topologicalSort

void Graph::topologicalSortUtil(int v, bool visited[],

                                stack<int> &Stack)

{

    // Mark the current node as visited.

    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex

    list<int>::iterator i;

    for (i = adj[v].begin(); i != adj[v].end(); ++i)

        if (!visited[\*i])

            topologicalSortUtil(\*i, visited, Stack);

    // Push current vertex to stack which stores result

    Stack.push(v);

}

// The function to do Topological Sort. It uses recursive

// topologicalSortUtil()

void Graph::topologicalSort()

{

    stack<int> Stack;

    // Mark all the vertices as not visited

    bool \*visited = new bool[V];

    for (int i = 0; i < V; i++)

        visited[i] = false;

    // Call the recursive helper function to store Topological

    // Sort starting from all vertices one by one

    for (int i = 0; i < V; i++)

      if (visited[i] == false)

        topologicalSortUtil(i, visited, Stack);

    // Print contents of stack

    while (Stack.empty() == false)

    {

        cout << Stack.top() << " ";

        Stack.pop();

    }

}

// Driver program to test above functions

int main()

{

    // Create a graph given in the above diagram

    Graph g(6);

    g.addEdge(5, 2);

    g.addEdge(5, 0);

    g.addEdge(4, 0);

    g.addEdge(4, 1);

    g.addEdge(2, 3);

    g.addEdge(3, 1);

    cout << "Following is a Topological Sort of the given graph \n";

    g.topologicalSort();

    return 0;

}.

**LCS:**

/\* A Naive recursive implementation of LCS problem \*/

#include<bits/stdc++.h>

int max(int a, int b);

/\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/

int lcs( char \*X, char \*Y, int m, int n )

{

   if (m == 0 || n == 0)

     return 0;

   if (X[m-1] == Y[n-1])

     return 1 + lcs(X, Y, m-1, n-1);

   else

     return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));

}

/\* Utility function to get max of 2 integers \*/

int max(int a, int b)

{

    return (a > b)? a : b;

}

/\* Driver program to test above function \*/

int main()

{

  char X[] = "AGGTAB";

  char Y[] = "GXTXAYB";

  int m = strlen(X);

  int n = strlen(Y);

  printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

  return 0;

}

**LIS:**

/\* A Naive C/C++ recursive implementation of LIS problem \*/

#include<stdio.h>

#include<stdlib.h>

/\* To make use of recursive calls, this function must return

   two things:

   1) Length of LIS ending with element arr[n-1]. We use

      max\_ending\_here for this purpose

   2) Overall maximum as the LIS may end with an element

      before arr[n-1] max\_ref is used this purpose.

   The value of LIS of full array of size n is stored in

   \*max\_ref which is our final result \*/

int \_lis( int arr[], int n, int \*max\_ref)

{

    /\* Base case \*/

    if (n == 1)

        return 1;

    // 'max\_ending\_here' is length of LIS ending with arr[n-1]

    int res, max\_ending\_here = 1;

    /\* Recursively get all LIS ending with arr[0], arr[1] ...

       arr[n-2]. If   arr[i-1] is smaller than arr[n-1], and

       max ending with arr[n-1] needs to be updated, then

       update it \*/

    for (int i = 1; i < n; i++)

    {

        res = \_lis(arr, i, max\_ref);

        if (arr[i-1] < arr[n-1] && res + 1 > max\_ending\_here)

            max\_ending\_here = res + 1;

    }

    // Compare max\_ending\_here with the overall max. And

    // update the overall max if needed

    if (\*max\_ref < max\_ending\_here)

       \*max\_ref = max\_ending\_here;

    // Return length of LIS ending with arr[n-1]

    return max\_ending\_here;

}

// The wrapper function for \_lis()

int lis(int arr[], int n)

{

    // The max variable holds the result

    int max = 1;

    // The function \_lis() stores its result in max

    \_lis( arr, n, &max );

    // returns max

    return max;

}

/\* Driver program to test above function \*/

int main()

{

    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };

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

    printf("Length of lis is %d\n",

           lis( arr, n ));

    return 0;

}

**##Longest Path in a Matrix**

|  |
| --- |
| #include<bits/stdc++.h>  #define n 3  using namespace std;    // Returns length of the longest path beginning with mat[i][j].  // This function mainly uses lookup table dp[n][n]  int findLongestFromACell(int i, int j, int mat[n][n], int dp[n][n])  {      // Base case      if (i<0 || i>=n || j<0 || j>=n)          return 0;        // If this subproblem is already solved      if (dp[i][j] != -1)          return dp[i][j];        // Since all numbers are unique and in range from 1 to n\*n,      // there is atmost one possible direction from any cell      if (j<n-1 && ((mat[i][j] +1) == mat[i][j+1]))         return dp[i][j] = 1 + findLongestFromACell(i,j+1,mat,dp);        if (j>0 && (mat[i][j] +1 == mat[i][j-1]))         return dp[i][j] = 1 + findLongestFromACell(i,j-1,mat,dp);        if (i>0 && (mat[i][j] +1 == mat[i-1][j]))         return dp[i][j] = 1 + findLongestFromACell(i-1,j,mat,dp);        if (i<n-1 && (mat[i][j] +1 == mat[i+1][j]))         return dp[i][j] = 1 + findLongestFromACell(i+1,j,mat,dp);        // If none of the adjacent fours is one greater      return dp[i][j] = 1;  }    // Returns length of the longest path beginning with any cell  int finLongestOverAll(int mat[n][n])  {      int result = 1;  // Initialize result        // Create a lookup table and fill all entries in it as -1      int dp[n][n];      memset(dp, -1, sizeof dp);        // Compute longest path beginning from all cells      for (int i=0; i<n; i++)      {        for (int j=0; j<n; j++)         {            if (dp[i][j] == -1)               findLongestFromACell(i, j, mat, dp);              //  Update result if needed            result = max(result, dp[i][j]);         }       }         return result;  }    // Driver program  int main()  {     int  mat[n][n] = {{1, 2, 9},                      {5, 3, 8},                      {4, 6, 7}};     cout << "Length of the longest path is "          << finLongestOverAll(mat);     return 0;  } |

**OPTIMAL STRATIGY FOR A GAME:**

|  |
| --- |
| // C program to find out maximum value from a given sequence of coins  #include <stdio.h>  #include <limits.h>    // Utility functions to get maximum and minimum of two intgers  int max(int a, int b)  {    return a > b ? a : b;  }  int min(int a, int b)  {    return a < b ? a : b;  }    // Returns optimal value possible that a player can collect from  // an array of coins of size n. Note than n must be even  int optimalStrategyOfGame(int\* arr, int n)  {      // Create a table to store solutions of subproblems      int table[n][n], gap, i, j, x, y, z;        // Fill table using above recursive formula. Note that the table      // is filled in diagonal fashion (similar to <http://goo.gl/PQqoS>),      // from diagonal elements to table[0][n-1] which is the result.      for (gap = 0; gap < n; ++gap)      {          for (i = 0, j = gap; j < n; ++i, ++j)          {              // Here x is value of F(i+2, j), y is F(i+1, j-1) and              // z is F(i, j-2) in above recursive formula              x = ((i+2) <= j) ? table[i+2][j] : 0;              y = ((i+1) <= (j-1)) ? table[i+1][j-1] : 0;              z = (i <= (j-2))? table[i][j-2]: 0;                table[i][j] = max(arr[i] + min(x, y), arr[j] + min(y, z));          }      }        return table[0][n-1];  }    // Driver program to test above function  int main()  {      int arr1[] = {8, 15, 3, 7};      int n = sizeof(arr1)/sizeof(arr1[0]);      printf("%d\n", optimalStrategyOfGame(arr1, n));        int arr2[] = {2, 2, 2, 2};      n = sizeof(arr2)/sizeof(arr2[0]);      printf("%d\n", optimalStrategyOfGame(arr2, n));        int arr3[] = {20, 30, 2, 2, 2, 10};      n = sizeof(arr3)/sizeof(arr3[0]);      printf("%d\n", optimalStrategyOfGame(arr3, n));        return 0;  } |

Output:

22

4

42

**O/1 KNAPSAK:**

|  |
| --- |
| /\* A Naive recursive implementation of 0-1 Knapsack problem \*/  #include<stdio.h>  // A utility function that returns maximum of two integers  int max(int a, int b) { return (a > b)? a : b; }  // Returns the maximum value that can be put in a knapsack of capacity W  int knapSack(int W, int wt[], int val[], int n)  {     // Base Case     if (n == 0 || W == 0)         return 0;     // If weight of the nth item is more than Knapsack capacity W, then     // this item cannot be included in the optimal solution     if (wt[n-1] > W)         return knapSack(W, wt, val, n-1);     // Return the maximum of two cases:     // (1) nth item included     // (2) not included     else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),                      knapSack(W, wt, val, n-1)                    );  }  // Driver program to test above function  int main()  {      int val[] = {60, 100, 120};      int wt[] = {10, 20, 30};      int  W = 50;      int n = sizeof(val)/sizeof(val[0]);      printf("%d", knapSack(W, wt, val, n));      return 0;  } |

**BINARY SEARCH:**

#include <stdio.h>

// A recursive binary search function. It returns location of x in

// given array arr[l..r] is present, otherwise -1

int binarySearch(int arr[], int l, int r, int x)

{

   if (r >= l)

   {

        int mid = l + (r - l)/2;

        // If the element is present at the middle itself

        if (arr[mid] == x)  return mid;

        // If element is smaller than mid, then it can only be present

        // in left subarray

        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray

        return binarySearch(arr, mid+1, r, x);

   }

   // We reach here when element is not present in array

   return -1;

}

int main(void)

{

   int arr[] = {2, 3, 4, 10, 40};

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

   int x = 10;

   int result = binarySearch(arr, 0, n-1, x);

   (result == -1)? printf("Element is not present in array")

                 : printf("Element is present at index %d", result);

   return 0;

}

**QUICK SORT:**

\* C implementation QuickSort \*/

#include<stdio.h>

// A utility function to swap two elements

void swap(int\* a, int\* b)

{

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

/\* This function takes last element as pivot, places

   the pivot element at its correct position in sorted

    array, and places all smaller (smaller than pivot)

   to left of pivot and all greater elements to right

   of pivot \*/

int partition (int arr[], int low, int high)

{

    int pivot = arr[high];    // pivot

    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)

    {

        // If current element is smaller than or

        // equal to pivot

        if (arr[j] <= pivot)

        {

            i++;    // increment index of smaller element

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

/\* The main function that implements QuickSort

 arr[] --> Array to be sorted,

  low  --> Starting index,

  high  --> Ending index \*/

void quickSort(int arr[], int low, int high)

{

    if (low < high)

    {

        /\* pi is partitioning index, arr[p] is now

           at right place \*/

        int pi = partition(arr, low, high);

        // Separately sort elements before

        // partition and after partition

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

    int i;

    for (i=0; i < size; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

// Driver program to test above functions

int main()

{

    int arr[] = {10, 7, 8, 9, 1, 5};

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

    quickSort(arr, 0, n-1);

    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}

**#MARGE SORT:**

/\* C program for Merge Sort \*/

#include<stdlib.h>

#include<stdio.h>

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

    int i, j, k;

    int n1 = m - l + 1;

    int n2 =  r - m;

    /\* create temp arrays \*/

    int L[n1], R[n2];

    /\* Copy data to temp arrays L[] and R[] \*/

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1+ j];

    /\* Merge the temp arrays back into arr[l..r]\*/

    i = 0; // Initial index of first subarray

    j = 0; // Initial index of second subarray

    k = l; // Initial index of merged subarray

    while (i < n1 && j < n2)

    {

        if (L[i] <= R[j])

        {

            arr[k] = L[i];

            i++;

        }

        else

        {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    /\* Copy the remaining elements of L[], if there

       are any \*/

    while (i < n1)

    {

        arr[k] = L[i];

        i++;

        k++;

    }

    /\* Copy the remaining elements of R[], if there

       are any \*/

    while (j < n2)

    {

        arr[k] = R[j];

        j++;

        k++;

    }

}

/\* l is for left index and r is right index of the

   sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

    if (l < r)

    {

        // Same as (l+r)/2, but avoids overflow for

        // large l and h

        int m = l+(r-l)/2;

        // Sort first and second halves

        mergeSort(arr, l, m);

        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);

    }

}

/\* UTILITY FUNCTIONS \*/

/\* Function to print an array \*/

void printArray(int A[], int size)

{

    int i;

    for (i=0; i < size; i++)

        printf("%d ", A[i]);

    printf("\n");

}

/\* Driver program to test above functions \*/

int main()

{

    int arr[] = {12, 11, 13, 5, 6, 7};

    int arr\_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");

    printArray(arr, arr\_size);

    mergeSort(arr, 0, arr\_size - 1);

    printf("\nSorted array is \n");

    printArray(arr, arr\_size);

    return 0;

}

**#define \_CRT\_SECURE\_NO\_WARNINGS**

**#include<iostream>**

**#include<sstream>**

**#include<cstdio>**

**#include<cmath>**

**#include<cstdlib>**

**#include<cctype>**

**#include<cstring>**

**#include<iomanip>**

**#include<string>**

**#include<vector>**

**#include<stack>**

**#include<queue>**

**#include<deque>**

**#include<set>**

**#include<map>**

**#include<list>**

**#include<algorithm>**

**#include<utility>**

**#include<functional>**

**using namespace std;**

**#define psb(x) push\_back(x)**

**#define psf(x) push\_front(x)**

**#define ppb pop\_back**

**#define ppf pop\_front**

**#define pop pop()**

**#define front front()**

**#define bgn begin()**

**#define end end()**

**#define emp empty()**

**#define clr clear()**

**#define sz size()**

**#define sp setprecision**

**#define fx fixed**

**#define ff first**

**#define ss second**

**#define reset(a) memset(a,0,sizeof(a))**

**#define assign(a,b) memset(a,b,sizeof(a))**

**#define assignmx(a) memset(a,127,sizeof(a))**

**#define assignmn(a) memset(a,128,sizeof(a))**

**#define max3(x, y, z) max(max(x, y), max(y, z))**

**#define range(r,c) ((r >=0 && r <row) && (c >=0 && c <column))**

**#define sfi3(a,b,c) scanf("%d%d%d",&a,&b,&c)**

**#define sfll3(a,b,c) scanf("%lld%lld%lld",&a,&b,&c)**

**#define sfd3(a,b,c) scanf("%lf%lf%lf",&a,&b,&c)**

**#define sfi2(a,b) scanf("%d%d",&a,&b)**

**#define sfll2(a,b) scanf("%lld%lld",&a,&b)**

**#define sfd2(a,b) scanf("%lf%lf",&a,&b)**

**#define sfi1(a) scanf("%d",&a)**

**#define sfll1(a) scanf("%lld",&a)**

**#define sfd1(a) scanf("%lf",&a)**

**#define sfch(a) scanf("%c",&a)**

**#define pf printf**

**#define sf scanf**

**#define getch ch=getchar()**

**#define for1(i,n) for(long long i=1;i<=n;i++)**

**#define for0(i,n) for(long long i=0;i<n;i++)**

**#define fort(t) for(int o=1 ;o <=t ;o++)**

**int X4[] = { 0, -1, 0, 1 };**

**int Y4[] = { -1, 0, 1, 0 };**

**int X8[] = { -1, -1, -1, 0, 0, 1, 1, 1 };**

**int Y8[] = { -1, 0, 1, -1, 1, -1, 0, 1 };**

**int X3D6[] = { 0, 0, -1, 1, 0, 0 };**

**int Y3D6[] = { -1, 1, 0, 0, 0, 0 };**

**int Z3D6[] = { 0, 0, 0, 0, -1, 1 };**

**#define INFMX 2139062143**

**#define INFMN -2139062144**

**#define pi acos(-1.0)**

**#define N 100001**

**#define size 32001**

**bool flag, flag1, flag2, flag3, flag4;**