Table of Contents

[Aim: Binary Search Recursion 2](#_Toc126025755)

[Aim: Dijkstra algorithm 3](#_Toc126025756)

[Aim: 0/1 knapsack Dynamic programming 6](#_Toc126025757)

[Aim: All pairs shortest path using floyd warshall algorithm 8](#_Toc126025758)

[Aim: knapsack using Greedy method 11](#_Toc126025759)

[Aim: Heap Sort 13](#_Toc126025760)

[Aim: Insertion Sort 15](#_Toc126025761)

[Aim: MST using Kruskal algorithm 17](#_Toc126025762)

[Aim: Linear Search Recursion 22](#_Toc126025763)

[Aim: Merge sort 23](#_Toc126025764)

[Aim: N Queens backtracking 26](#_Toc126025765)

[Aim: MST using Prim’s algorithm 29](#_Toc126025766)

[Aim: Quick sort 32](#_Toc126025767)

[Aim: Selection sort 34](#_Toc126025768)

[Aim: Travelling Salesman Problem 36](#_Toc126025769)

# Aim: write a program in c to perform binary search with recursive approach

Algorithm:

1. Take an array arr, left index l, right index r, and element to be searched x as input.
2. If r >= l, find the mid index mid = l + (r - l) / 2.
3. If arr[mid] is equal to x, return mid as element is found.
4. If arr[mid] > x, recursively call the function with l and mid - 1.
5. If arr[mid] < x, recursively call the function with mid + 1 and r.
6. Return -1 if element is not found.

Source Code:

#include <stdio.h>

*int* binary\_search\_recursive(*int* arr[], *int* l, *int* r, *int* x) {

    if (r >= l) {

*int* mid = l + (r - l) / 2;

        if (arr[mid] == x)

            return mid;

        if (arr[mid] > x)

            return binary\_search\_recursive(arr, l, mid - 1, x);

        return binary\_search\_recursive(arr, mid + 1, r, x);

    }

    return -1;

}

*int* main() {

*int* arr[] = {2, 3, 4, 10, 40};

*int* x = 10;

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

*int* result = binary\_search\_recursive(arr, 0, n - 1, x);

    if (result == -1)

        printf("Element is not present in array");

    else

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

    return 0;

}

Output:

Element is present at index 3

# Aim: write a program in c to implement Dijkstra algorithm

Algorithm:

1. Initialize the distances from the source node to all other nodes as infinity and keep track of which nodes have been processed.
2. Choose the node with the shortest distance from the source that has not been processed.
3. Mark that node as processed.
4. Update the distances of the neighboring nodes of the processed node if the new distance is shorter.
5. Repeat steps 2-4 until all nodes have been processed.
6. The final distances from the source node will be the solution to the problem.

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_NODES 100

#define MAX\_DISTANCE 1000000

*int* n;

*int* adjacency\_matrix[MAX\_NODES][MAX\_NODES];

*int* distances[MAX\_NODES];

*bool* processed[MAX\_NODES];

*int* find\_min\_distance\_node() {

*int* min\_distance = MAX\_DISTANCE;

*int* min\_distance\_node = -1;

  for (*int* i = 0; i < n; i++) {

    if (!processed[i] && distances[i] < min\_distance) {

      min\_distance = distances[i];

      min\_distance\_node = i;

    }

  }

  return min\_distance\_node;

}

*void* dijkstra(*int* source) {

  for (*int* i = 0; i < n; i++) {

    distances[i] = MAX\_DISTANCE;

    processed[i] = *false*;

  }

  distances[source] = 0;

  for (*int* i = 0; i < n; i++) {

*int* min\_distance\_node = find\_min\_distance\_node();

    processed[min\_distance\_node] = *true*;

    for (*int* j = 0; j < n; j++) {

      if (!processed[j] && adjacency\_matrix[min\_distance\_node][j] != 0) {

*int* distance = distances[min\_distance\_node] + adjacency\_matrix[min\_distance\_node][j];

        if (distance < distances[j]) {

          distances[j] = distance;

        }

      }

    }

  }

}

*int* main() {

  printf("Enter the number of nodes: ");

  scanf("%d", &n);

  printf("Enter the adjacency matrix representation of the graph:\n");

  for (*int* i = 0; i < n; i++) {

    for (*int* j = 0; j < n; j++) {

      scanf("%d", &adjacency\_matrix[i][j]);

    }

  }

*int* source;

  printf("Enter the source node: ");

  scanf("%d", &source);

  dijkstra(source);

  printf("Distances from the source node:\n");

  for (*int* i = 0; i < n; i++) {

    printf("%d ", distances[i]);

  }

  printf("\n");

  return 0;

}

Output:

Enter the number of nodes: 6

Enter the adjacency matrix representation of the graph:

0 4 5 0 0 0

4 0 11 9 7 0

5 11 0 0 3 0

0 9 0 0 13 2

0 7 3 13 0 6

0 0 0 2 6 0

Enter the source node: 0

Distances from the source node:

0 4 5 13 8 14

# Aim: write a program in c to implement 0/1 knapsack problem using dynamic programming

Algorithm:

1. Initialize a 2D array K[n+1][W+1] where n is the number of items and W is the capacity of the knapsack.
2. Loop through the items i and the weight w of the knapsack.
3. If the weight of the current item wt[i-1] is less than or equal to w, then store the maximum value of either including the current item or not including it. K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
4. If the weight of the current item wt[i-1] is more than w, then the current item can't be included and the value remains the same as the previous value. K[i][w] = K[i-1][w]
5. Return K[n][W] as the maximum value that can be put in the knapsack.

Source Code:

#include <stdio.h>

#include <stdlib.h>

*int* max(*int* a, *int* b)

{

    return (a > b) ? a : b;

}

*int* knapSack(*int* W, *int* wt[], *int* val[], *int* n)

{

*int* i, w;

*int* K[n + 1][W + 1];

    for (i = 0; i <= n; i++)

    {

        for (w = 0; w <= W; w++)

        {

            if (i == 0 || w == 0)

                K[i][w] = 0;

            else if (wt[i - 1] <= w)

                K[i][w] = max(val[i - 1] +

                    K[i - 1][w - wt[i - 1]], K[i - 1][w]);

            else

                K[i][w] = K[i - 1][w];

        }

    }

    return K[n][W];

}

*int* main()

{

*int* n, W;

    printf("Enter number of items: ");

    scanf("%d", &n);

*int* val[n], wt[n];

    for (*int* i = 0; i < n; i++) {

        printf("Enter value and weight for item %d: ", i + 1);

        scanf("%d%d", &val[i], &wt[i]);

    }

    printf("Enter knapsack capacity: ");

    scanf("%d", &W);

    printf("The maximum value that can be put in knapsack is %d\n", knapSack(W, wt, val, n));

    return 0;

}

Output:

Enter number of items: 3

Enter value and weight for item 1: 60 10

Enter value and weight for item 2: 100 20

Enter value and weight for item 3: 120 30

Enter knapsack capacity: 50

The maximum value that can be put in knapsack is 220

# Aim: write a program in c to implement all pairs shortest path using floyed's algorithm

Algorithm:

1. Initialize the distances between all pairs of nodes as infinity or zero (if the nodes are the same).
2. Get the distances between all pairs of nodes from user input.
3. Find all pairs shortest path by repeatedly checking if there is a shorter path through a intermediate node.
4. Repeat step 3 for all possible intermediate nodes.
5. The final distances between all pairs of nodes will be the solution to the problem.

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX\_NODES 100

#define MAX\_DISTANCE INT\_MAX

*int* n;

*int* distances[MAX\_NODES][MAX\_NODES];

*void* floyd() {

  for (*int* i = 0; i < n; i++) {

    for (*int* j = 0; j < n; j++) {

      if (i == j) {

        distances[i][j] = 0;

      } else {

        distances[i][j] = MAX\_DISTANCE;

      }

    }

  }

  for (*int* i = 0; i < n; i++) {

    for (*int* j = 0; j < n; j++) {

*int* distance;

      scanf("%d", &distance);

      distances[i][j] = distance;

    }

  }

  for (*int* k = 0; k < n; k++) {

    for (*int* i = 0; i < n; i++) {

      for (*int* j = 0; j < n; j++) {

        if (distances[i][j] > distances[i][k] + distances[k][j]) {

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

        }

      }

    }

  }

}

*int* main() {

  printf("Enter the number of nodes: ");

  scanf("%d", &n);

  floyd();

  printf("All pairs shortest path:\n");

  for (*int* i = 0; i < n; i++) {

    for (*int* j = 0; j < n; j++) {

      printf("%d ", distances[i][j]);

    }

    printf("\n");

  }

  return 0;

}

Output:

Enter the number of nodes: 4

0 3 9999 5

2 0 9999 4

9999 1 0 9999

9999 9999 2 0

All pairs shortest path:

0 3 7 5

2 0 6 4

3 1 0 5

5 3 2 0

# Aim: write a program in c to perform knapsack problem using greedy method

Algorithm for Knapsack problem using Greedy method:

1. Sort the items in descending order of their value-to-weight ratio.
2. Pick items in the sorted order, adding them to the knapsack until it is full.
3. Calculate the final value of the knapsack by adding up the values of the items picked.

Source Code:

#include <stdio.h>

#include <stdlib.h>

typedef *struct* {

*int* value;

*int* weight;

*float* ratio;

} Item;

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

{

    Item \*ia = (Item \*)a;

    Item \*ib = (Item \*)b;

    return (*int*)(100.0f \* ia->ratio - 100.0f \* ib->ratio);

}

*void* knapsack(*int* n, *int* W, Item arr[])

{

*int* curWeight = 0;

*int* finalValue = 0;

    for (*int* i = 0; i < n; i++) {

        if (curWeight + arr[i].weight <= W) {

            finalValue += arr[i].value;

            curWeight += arr[i].weight;

        }

        else {

*int* remain = W - curWeight;

            finalValue += arr[i].value \* ((*float*) remain / arr[i].weight);

            break;

        }

    }

    printf("Maximum value we can obtain = %d\n", finalValue);

}

*int* main()

{

*int* W, n;

    printf("Enter the maximum weight capacity of the knapsack: ");

    scanf("%d", &W);

    printf("Enter the number of items: ");

    scanf("%d", &n);

    Item arr[n];

    for (*int* i = 0; i < n; i++) {

        printf("Enter the value and weight of item %d: ", i + 1);

        scanf("%d%d", &arr[i].value, &arr[i].weight);

        arr[i].ratio = (*float*) arr[i].value / arr[i].weight;

    }

    qsort(arr, n, sizeof(arr[0]), compare);

    knapsack(n, W, arr);

    return 0;

}

Output:

Enter the maximum weight capacity of the knapsack: 50

Enter the number of items: 3

Enter the value and weight of item 1: 60

10

Enter the value and weight of item 2: 100

20

Enter the value and weight of item 3: 120

30

Maximum value we can obtain = 220

# Aim: write a program in c to perform heap sort

Algorithm:

1. Build a max heap from the input data.
2. At the root of the heap, exchange the first element with the last element.
3. Reduce the size of the heap by 1 and heapify the root element.
4. Repeat steps 2 and 3 until all elements are sorted.

Source Code:

#include <stdio.h>

#include <stdlib.h>

*void* heapify(*int* arr[], *int* n, *int* i)

{

*int* largest = i;

*int* l = 2\*i + 1;

*int* r = 2\*i + 2;

    if (l < n && arr[l] > arr[largest])

        largest = l;

    if (r < n && arr[r] > arr[largest])

        largest = r;

    if (largest != i)

    {

*int* temp = arr[i];

        arr[i] = arr[largest];

        arr[largest] = temp;

        heapify(arr, n, largest);

    }

}

*void* heapSort(*int* arr[], *int* n)

{

    for (*int* i = n / 2 - 1; i >= 0; i--)

        heapify(arr, n, i);

    for (*int* i=n-1; i>=0; i--)

    {

*int* temp = arr[0];

        arr[0] = arr[i];

        arr[i] = temp;

        heapify(arr, i, 0);

    }

}

*void* printArray(*int* arr[], *int* n)

{

    for (*int* i=0; i < n; ++i)

        printf("%d ", arr[i]);

    printf("\n");

}

*int* main()

{

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

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

    heapSort(arr, n);

    printf("Sorted array is \n");

    printArray(arr, n);

}

Output:

Sorted array is

5 6 7 11 12 13

# Aim: write a program in c to perform insertion sort

Algorithm for Insertion Sort:

1. Start by picking the second element in the array.
2. Compare the second element with the one before it and swap if necessary.
3. Continue to the next element and if it is in the incorrect order, iterate through the sorted portion (i.e. the left side) to place the element in the correct place.
4. Repeat until the array is sorted.

Source Code:

#include <stdio.h>

*void* insertionSort(*int* arr[], *int* n)

{

*int* i, key, j;

    for (i = 1; i < n; i++)

    {

        key = arr[i];

        j = i - 1;

        while (j >= 0 && arr[j] > key)

        {

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = key;

    }

}

*void* printArray(*int* arr[], *int* n)

{

*int* i;

    for (i = 0; i < n; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

*int* main()

{

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

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

    insertionSort(arr, n);

    printArray(arr, n);

    return 0;

}

Output:

5 6 11 12 13

# Aim: write a program in c to find MST using Kruskal algorithm

Algorithm:

1. Sort the edges of the graph in ascending order of their weights.

2. Initialize an empty result set and an empty subset (disjoint sets).

3. Iterate through the sorted edges:

a. If adding the edge to the result set doesn't form a cycle, add it to the result set and update the disjoint sets.

b. If adding the edge forms a cycle, skip it.

4. Repeat the steps 3a and 3b until there are V-1 edges in the result set or all edges have been considered.

5. The result set is the minimum spanning tree.

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 100

typedef *struct* edge {

*int* source, dest, weight;

} Edge;

typedef *struct* graph {

*int* V, E;

    Edge\* edge;

} Graph;

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

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

    graph->V = V;

    graph->E = E;

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

    return graph;

}

*int* find(*int* parent[], *int* i) {

    if (parent[i] == -1)

        return i;

    return find(parent, parent[i]);

}

*void* union1(*int* parent[], *int* x, *int* y) {

*int* xset = find(parent, x);

*int* yset = find(parent, y);

    parent[xset] = yset;

}

*int* isCycle(Graph\* graph) {

*int* parent[MAX];

    memset(parent, -1, sizeof(parent));

    for (*int* i = 0; i < graph->E; i++) {

*int* x = find(parent, graph->edge[i].source);

*int* y = find(parent, graph->edge[i].dest);

        if (x == y)

            return 1;

        union1(parent, x, y);

    }

    return 0;

}

*int* cmp(const *void*\* a, const *void*\* b) {

    Edge\* a1 = (Edge\*)a;

    Edge\* b1 = (Edge\*)b;

    return a1->weight > b1->weight;

}

*void* KruskalMST(Graph\* graph) {

*int* V = graph->V;

    Edge result[V];

*int* e = 0;

*int* i = 0;

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

*int* subset[MAX];

    memset(subset, -1, sizeof(subset));

    while (e < V - 1) {

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

*int* x = find(subset, next\_edge.source);

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

        if (x != y) {

            result[e++] = next\_edge;

            union1(subset, x, y);

        }

    }

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

    for (i = 0; i < e; ++i)

        printf("%d - %d: %d\n", result[i].source, result[i].dest, result[i].weight);

    return;

}

*int* main() {

*int* V, E, i, s, d, w;

    printf("Enter number of vertices: ");

    scanf("%d", &V);

    printf("Enter number of edges: ");

    scanf("%d", &E);

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

    printf("Enter source, destinationand weight for each edge:\n");

    for (i = 0; i < E; i++) {

        scanf("%d %d %d", &s, &d, &w);

        graph->edge[i].source = s;

        graph->edge[i].dest = d;

        graph->edge[i].weight = w;

    }

    KruskalMST(graph);

    return 0;

}

Output:

Enter number of vertices: 4

Enter number of edges: 5

Enter source, destinationand weight for each edge:

0 1 10

0 2 6

0 3 5

1 3 15

2 3 4

Following are the edges in the constructed MST

2 - 3: 4

0 - 3: 5

0 - 1: 1

# Aim: write a program in c to perform linear search with recursive approach

Algorithm:

1. Take an array arr, element to be searched x, size of array n and current index index as input.
2. If index is equal to n, return -1 as element is not found.
3. If arr[index] is equal to x, return index as element is found.
4. Recursively call the function with index + 1.
5. Return the index if element is found, else -1 if not found.

Source Code:

#include <stdio.h>

*int* linear\_search\_recursive(*int* arr[], *int* x, *int* n, *int* index) {

    if (index == n)

        return -1;

    if (arr[index] == x)

        return index;

    return linear\_search\_recursive(arr, x, n, index + 1);

}

*int* main() {

*int* arr[] = {10, 20, 80, 30, 60, 50,

                 110, 100, 130, 170};

*int* x = 110;

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

*int* result = linear\_search\_recursive(arr, x, n, 0);

    if (result == -1)

        printf("Element is not present in array");

    else

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

    return 0;

}

Output:

Element is present at index 6

# Aim: write a program in c to perform merge sort

Algorithm:

1. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Source Code:

#include <stdio.h>

#include <stdlib.h>

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

{

*int* i, j, k;

*int* n1 = m - l + 1;

*int* n2 =  r - m;

*int* L[n1], R[n2];

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1+ j];

    i = 0;

    j = 0;

    k = l;

    while (i < n1 && j < n2)

    {

        if (L[i] <= R[j])

        {

            arr[k] = L[i];

            i++;

        }

        else

        {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    while (i < n1)

    {

        arr[k] = L[i];

        i++;

        k++;

    }

    while (j < n2)

    {

        arr[k] = R[j];

        j++;

        k++;

    }

}

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

{

    if (l < r)

    {

*int* m = l+(r-l)/2;

        mergeSort(arr, l, m);

        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);

    }

}

*void* printArray(*int* A[], *int* size)

{

*int* i;

    for (i=0; i < size; i++)

        printf("%d ", A[i]);

    printf("\n");

}

*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;

}

Output:

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

# Aim: write a program to implement n queens problem using backtracking

Algorithm:

1. Initialize a 2D array chess\_board[N][N] to represent the chess board, where N is the size of the chess board.
2. Define a function isSafe(chess\_board, row, col) to check if it is safe to place a queen in a given cell (row, col).
3. Define a function solveNQUtil(chess\_board, col) which uses backtracking to place queens in each column.
4. If all queens are placed successfully, the function returns true.
5. If a queen cannot be placed in the current column, the function returns false.
6. In the main function, take the input n from the user and call solveNQUtil(chess\_board, 0).
7. If solveNQUtil returns false, the solution does not exist.
8. If solveNQUtil returns true, the solution exists and print the chess board.

Note: The chess\_board is initialized to all zeros and the cells with 1 represent the placement of a queen.

Source Code:

#include<stdio.h>

#include<stdbool.h>

*int* N;

*int* chess\_board[100][100];

*int* count = 0;

*void* printSolution(*int* chess\_board[100][100])

{

    printf("Solution %d:\n", ++count);

    for (*int* i = 0; i < N; i++)

    {

        for (*int* j = 0; j < N; j++)

            printf(" %d ", chess\_board[i][j]);

        printf("\n");

    }

}

*bool* isSafe(*int* chess\_board[100][100], *int* row, *int* col)

{

*int* i, j;

    for (i = 0; i < col; i++)

        if (chess\_board[row][i])

            return *false*;

    for (i=row, j=col; i>=0 && j>=0; i--, j--)

        if (chess\_board[i][j])

            return *false*;

    for (i=row, j=col; j>=0 && i<N; i++, j--)

        if (chess\_board[i][j])

            return *false*;

    return *true*;

}

*bool* solveNQUtil(*int* chess\_board[100][100], *int* col)

{

    if (col == N)

    {

        printSolution(chess\_board);

        return *false*;

    }

    for (*int* i = 0; i < N; i++)

    {

        if ( isSafe(chess\_board, i, col) )

        {

            chess\_board[i][col] = 1;

            if ( solveNQUtil(chess\_board, col + 1) == *true* )

                return *true*;

            chess\_board[i][col] = 0;

        }

    }

    return *false*;

}

*int* main()

{

    printf("Enter the value of n for n-queen problem: ");

    scanf("%d", &N);

    solveNQUtil(chess\_board, 0);

    return 0;

}

Output:

Enter the value of n for n-queen problem: 4

Solution 1:

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

Solution 2:

0 1 0 0

0 0 0 1

1 0 0 0

0 0 1 0

# Aim: write a program in c to find MST using Prim’s algorithm

The algorithm for Prim's algorithm is as follows:

1. Create a distance array to store the minimum distance from a node to the source node and initialize it with a large value except for the source node which should be 0.
2. Create a selected array to keep track of the nodes that have been selected for the minimum spanning tree and initialize it with false.
3. For n-1 times, do the following: a. Select a node u that has not been selected and has the minimum distance from the source node. b. Mark u as selected. c. For each node v that is not selected, if the distance from u to v is smaller than the current distance stored in the distance array, update the distance array with the new distance and store the node u as the previous node in the from array.
4. After completing the above steps, print the edges and their costs in the minimum spanning tree.

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX 100

*int* n, cost[MAX][MAX];

*void* prim(*int* source) {

*int* i, j, u, v, min\_distance, distance[MAX], from[MAX];

*bool* selected[MAX];

    for (i = 1; i <= n; i++) {

        distance[i] = INT\_MAX;

        selected[i] = *false*;

    }

    distance[source] = 0;

    from[source] = -1;

    for (i = 1; i < n; i++) {

        min\_distance = INT\_MAX;

        for (j = 1; j <= n; j++) {

            if (!selected[j] && distance[j] < min\_distance) {

                min\_distance = distance[j];

                u = j;

            }

        }

        selected[u] = *true*;

        for (v = 1; v <= n; v++) {

            if (!selected[v] && cost[u][v] && distance[v] > cost[u][v]) {

                distance[v] = cost[u][v];

                from[v] = u;

            }

        }

    }

    printf("Edge\tCost\n");

    for (i = 2; i <= n; i++)

        printf("%d - %d\t%d\n", from[i], i, distance[i]);

}

*int* main() {

*int* i, j, source;

    printf("Enter number of nodes: ");

    scanf("%d", &n);

    printf("Enter the cost matrix:\n");

    for (i = 1; i <= n; i++) {

        for (j = 1; j <= n; j++) {

            scanf("%d", &cost[i][j]);

        }

    }

    printf("Enter the source node: ");

    scanf("%d", &source);

    prim(source);

    return 0;

}

Output:

Enter number of nodes: 5

Enter the cost matrix:

0 9 75 0 0

9 0 95 19 42

75 95 0 51 66

0 19 51 0 31

0 42 66 31 0

Enter the source node: 1

Edge Cost

1 - 2 9

4 - 3 51

2 - 4 19

4 - 5 31

# Aim: write a program in c to perform quick sort

Algorithm for Quick Sort:

1. Choose a pivot element from the array.
2. Partition the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
3. Recursively sort the sub-arrays.
4. Combine the elements back into a single sorted array.

Source Code:

#include <stdio.h>

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

{

*int* temp = \*a;

    \*a = \*b;

    \*b = temp;

}

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

{

*int* pivot = arr[high];

*int* i = (low - 1);

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

    {

        if (arr[j] < pivot)

        {

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

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

{

    if (low < high)

    {

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

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

*void* printArray(*int* arr[], *int* size)

{

*int* i;

    for (i = 0; i < size; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

*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;

}

Output:

Sorted array:

1 5 7 8 9 10

# Aim: write a program in c to perform selection sort

Algorithm:

1. Find the minimum element in the unsorted portion of the list.
2. Swap the minimum element with the first element of the unsorted portion.
3. Move the boundary of the unsorted portion one element to the right.
4. Repeat the steps above until the unsorted portion is empty.

Source Code:

#include <stdio.h>

*void* swap(*int* \*x, *int* \*y)

{

*int* temp = \*x;

    \*x = \*y;

    \*y = temp;

}

*void* selectionSort(*int* arr[], *int* n)

{

*int* i, j, min\_idx;

    for (i = 0; i < n-1; i++)

    {

        min\_idx = i;

        for (j = i+1; j < n; j++)

        if (arr[j] < arr[min\_idx])

            min\_idx = j;

        swap(&arr[min\_idx], &arr[i]);

    }

}

*void* printArray(*int* arr[], *int* size)

{

*int* i;

    for (i=0; i < size; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

*int* main()

{

*int* arr[] = {64, 25, 12, 22, 11};

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

    selectionSort(arr, n);

    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}

Output:

Sorted array:

11 12 22 25 64

# Aim: write a program in c to perform travelling salesman problem

Algorithm:

1. Start with the first city as the current city.
2. Try every possible city as the next city and recursively continue the search.
3. If all cities have been visited, calculate the distance of the path and compare it to the current minimum distance. If it is shorter, update the minimum distance and store the path.
4. Backtrack by marking the current city as unvisited and trying the next city.
5. Repeat steps 2-4 until all possible paths have been tried.
6. The minimum distance and path found in step 5 will be the solution to the traveling salesman problem.

Source Code: #include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_CITIES 100

#define MAX\_DISTANCE 10000

*int* n;

*int* dist[MAX\_CITIES][MAX\_CITIES];

*int* path[MAX\_CITIES];

*bool* visited[MAX\_CITIES];

*int* min\_distance = MAX\_DISTANCE;

*int* calculate\_distance(*int* path[]) {

*int* distance = 0;

  for (*int* i = 0; i < n - 1; i++) {

    distance += dist[path[i]][path[i + 1]];

  }

  distance += dist[path[n - 1]][path[0]];

  return distance;

}

*void* search(*int* current\_city, *int* current\_distance, *int* current\_index) {

  if (current\_index == n) {

    if (current\_distance < min\_distance) {

      min\_distance = current\_distance;

      for (*int* i = 0; i < n; i++) {

        path[i] = path[i];

      }

    }

    return;

  }

  for (*int* i = 0; i < n; i++) {

    if (!visited[i]) {

      visited[i] = *true*;

      path[current\_index] = i;

      search(i, current\_distance + dist[current\_city][i], current\_index + 1);

      visited[i] = *false*;

    }

  }

}

*int* main() {

  printf("Enter the number of cities: ");

  scanf("%d", &n);

  printf("Enter the distances between the cities (use -1 for infinity):\n");

  for (*int* i = 0; i < n; i++) {

    for (*int* j = 0; j < n; j++) {

      scanf("%d", &dist[i][j]);

    }

  }

  path[0] = 0;

  visited[0] = *true*;

  search(0, 0, 1);

  printf("Shortest path: ");

  for (*int* i = 0; i < n; i++) {

    printf("%d ", path[i]);

  }

  printf("\n");

  printf("Shortest distance: %d\n", min\_distance);

  printf("Path Cost: %d",calculate\_distance(path));

  return 0;

}

Output:

Enter the number of cities: 6

Enter the distances between the cities (use -1 for infinity):

-1 10 15 20 -1 8

5 -1 9 10 8 -1

6 13 -1 12 -1 5

8 8 9 -1 6 -1

-1 10 -1 6 -1 -1

10 -1 5 -1 -1 -1

Shortest path: 0 5 4 3 2 1

Shortest distance: 8

Path Cost: 40