**NAME: AKSHAT SRIVASTAV**

**REGISTER NUMBER: 19BCE0811**

**DSA PROJECT ASSIGNMENT**

[1] Heap Sort Algorithm (the algorithm must build a Max-Heap from the input data and then perform the sorting operation)

// C++ program for implementation of Heap Sort

#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is

// an index in arr[]. n is size of heap

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

{

    int largest = i; // Initialize largest as root

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

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

    // If left child is larger than root

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

        largest = l;

    // If right child is larger than largest so far

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

        largest = r;

    // If largest is not root

    if (largest != i)

    {

        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree

        heapify(arr, n, largest);

    }

}

// main function to do heap sort

void heapSort(int arr[], int n)

{

    // Build heap (rearrange array)

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

        heapify(arr, n, i);

    // One by one extract an element from heap

    for (int i=n-1; i>0; i--)

    {

        // Move current root to end

        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap

        heapify(arr, i, 0);

    }

}

/\* A utility function to print array of size n \*/

void printArray(int arr[], int n)

{

    for (int i=0; i<n; ++i)

        cout << arr[i] << " ";

    cout << "\n";

}

// Driver program

int main()

{

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

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

    heapSort(arr, n);

    cout << "Sorted array is \n";

    printArray(arr, n);

}

output:

Sorted array is

5 6 7 11 12 13

[2] Prim’s Algorithm

// A C++ program for Prim's Minimum

// Spanning Tree (MST) algorithm. The program is

// for adjacency matrix representation of the graph

#include <bits/stdc++.h>

using namespace std;

// 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[]

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

{

    cout<<"Edge \tWeight\n";

    for (int i = 1; i < V; i++)

        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";

}

// Function to construct and print MST for

// a graph represented using adjacency

// matrix representation

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

{

    // Array to store constructed MST

    int parent[V];

    // Key values used to pick minimum weight edge in cut

    int key[V];

    // To represent set of vertices included in MST

    bool mstSet[V];

    // 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.

    // Make key 0 so that this vertex is picked as first vertex.

    key[0] = 0;

    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 the 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, graph);

}

// Driver code

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;

}

output:

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

[3] Dijkstra’s Algorithm

// A C++ program for Dijkstra's single source shortest path algorithm.

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

#include <limits.h>

#include <stdio.h>

// Number of vertices in the graph

#define V 9

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

// the set of vertices not yet included in shortest path tree

int minDistance(int dist[], bool sptSet[])

{

    // Initialize min value

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < V; v++)

        if (sptSet[v] == false && dist[v] <= min)

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

    return min\_index;

}

// A utility function to print the constructed distance array

void printSolution(int dist[])

{

    printf("Vertex \t\t Distance from Source\n");

    for (int i = 0; i < V; i++)

        printf("%d \t\t %d\n", i, dist[i]);

}

// Function that implements Dijkstra's single source shortest path algorithm

// for a graph represented using adjacency matrix representation

void dijkstra(int graph[V][V], int src)

{

    int dist[V]; // The output array.  dist[i] will hold the shortest

    // distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest

    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false

    for (int i = 0; i < V; i++)

        dist[i] = INT\_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0

    dist[src] = 0;

    // Find shortest path for all vertices

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

        // Pick the minimum distance vertex from the set of vertices not

        // yet processed. u is always equal to src in the first iteration.

        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed

        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the picked vertex.

        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an edge from

            // u to v, and total weight of path from src to  v through u is

            // smaller than current value of dist[v]

            if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

                && dist[u] + graph[u][v] < dist[v])

                dist[v] = dist[u] + graph[u][v];

    }

    // print the constructed distance array

    printSolution(dist);

}

// driver program to test above function

int main()

{

    /\* Let us create the example graph discussed above \*/

    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },

                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },

                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },

                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },

                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },

                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },

                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },

                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },

                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;

}

output:

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

5 11

6 9

7 8

8 14

[4] Quick Sort Algorithm (Divide and Conquer Approach)

/\* C++ implementation of QuickSort \*/

#include <bits/stdc++.h>

using namespace std;

// 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 the 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++)

        cout << arr[i] << " ";

    cout << endl;

}

// Driver Code

int main()

{

    int arr[] = {10, 7, 8, 9, 1, 5};

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

    quickSort(arr, 0, n - 1);

    cout << "Sorted array: \n";

    printArray(arr, n);

    return 0;

}

output:

Sorted array:

1 5 7 8 9 10

[5] Brute Force & Greedy Algorithm for Solving 0-1 Knapsack Problem

#include <bits/stdc++.h>

using namespace std;

// 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 code

int main()

{

int val[] = { 60, 100, 120 };

int wt[] = { 10, 20, 30 };

int W = 50;

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

cout << knapSack(W, wt, val, n);

return 0;

}

output:

220

[6] 8-Queens Problem using Backtracking Technique

#include<stdio.h>

#include<stdlib.h>

int t[8] = {-1};

int sol = 1;

void printsol()

{

int i,j;

char crossboard[8][8];

for(i=0;i<8;i++)

{

for(j=0;j<8;j++)

{

crossboard[i][j]='\_';

}

}

for(i=0;i<8;i++)

{

crossboard[i][t[i]]='q';

}

for(i=0;i<8;i++)

{

for(j=0;j<8;j++)

{

printf("%c ",crossboard[i][j]);

}

printf("\n");

}

}

int empty(int i)

{

int j=0;

while((t[i]!=t[j])&&(abs(t[i]-t[j])!=(i-j))&&j<8)j++;

return i==j?1:0;

}

void queens(int i)

{

for(t[i] = 0;t[i]<8;t[i]++)

{

if(empty(i))

{

if(i==7){

printsol();

printf("\n solution %d\n",sol++);

}

else

queens(i+1);

}

}

}

int main()

{

queens(0);

return 0;

}

[7] Unbounded Knapsack Problem using Dynamic Programming

#include <bits/stdc++.h>

using namespace std;

long int UnboundedKnapsack(long int Capacity,long int n, long int weight[],long int val[]){

long int dp[Capacity+1];

for(int i=0;i < W+1;i++){

dp[i]=0;

}

for(int i=0;i < W+1;i++){

for(int j=0;j < n;j++){

if(weight[j] < i){

dp[i] = max(dp[i], dp[i-weight[j]] + val[j]);

}

}

}

return dp[Capacity];

}

int main(){

// The no. of items :

long int n = 4;

// Weights of all the items :

long int weight[4] = {5 , 10, 8, 15};

// Enter values of all the items :

long int val[4] = {40, 30, 50, 25};

// Enter the knapsack capacity :

long int Capacity = 120;

cout << "The maximum value you can achieve in Unbounded Knapsack is: " << UnboundedKnapsack(W,n,wt,val);

return 0;

}

/\* C++ program for solution of Hamiltonian

Cycle problem using backtracking \*/

#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 5

void printSolution(int path[]);

/\* A utility function to check if

the vertex v can be added at index 'pos'

in the Hamiltonian Cycle constructed

so far (stored in 'path[]') \*/

bool isSafe(int v, bool graph[V][V],

int path[], int pos)

{

/\* Check if this vertex is an adjacent

vertex of the previously added vertex. \*/

if (graph [path[pos - 1]][ v ] == 0)

return false;

/\* Check if the vertex has already been included.

This step can be optimized by creating

an array of size V \*/

for (int i = 0; i < pos; i++)

if (path[i] == v)

return false;

return true;

}

/\* A recursive utility function

to solve hamiltonian cycle problem \*/

bool hamCycleUtil(bool graph[V][V],

int path[], int pos)

{

/\* base case: If all vertices are

included in Hamiltonian Cycle \*/

if (pos == V)

{

// And if there is an edge from the

// last included vertex to the first vertex

if (graph[path[pos - 1]][path[0]] == 1)

return true;

else

return false;

}

// Try different vertices as a next candidate

// in Hamiltonian Cycle. We don't try for 0 as

// we included 0 as starting point in hamCycle()

for (int v = 1; v < V; v++)

{

/\* Check if this vertex can be added

// to Hamiltonian Cycle \*/

if (isSafe(v, graph, path, pos))

{

path[pos] = v;

/\* recur to construct rest of the path \*/

if (hamCycleUtil (graph, path, pos + 1) == true)

return true;

/\* If adding vertex v doesn't lead to a solution,

then remove it \*/

path[pos] = -1;

}

}

/\* If no vertex can be added to

Hamiltonian Cycle constructed so far,

then return false \*/

return false;

}

/\* This function solves the Hamiltonian Cycle problem

using Backtracking. It mainly uses hamCycleUtil() to

solve the problem. It returns false if there is no

Hamiltonian Cycle possible, otherwise return true

and prints the path. Please note that there may be

more than one solutions, this function prints one

of the feasible solutions. \*/

bool hamCycle(bool graph[V][V])

{

int \*path = new int[V];

for (int i = 0; i < V; i++)

path[i] = -1;

/\* Let us put vertex 0 as the first vertex in the path.

If there is a Hamiltonian Cycle, then the path can be

started from any point of the cycle as the graph is undirected \*/

path[0] = 0;

if (hamCycleUtil(graph, path, 1) == false )

{

cout << "\nSolution does not exist";

return false;

}

printSolution(path);

return true;

}

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

void printSolution(int path[])

{

cout << "Solution Exists:"

" Following is one Hamiltonian Cycle \n";

for (int i = 0; i < V; i++)

cout << path[i] << " ";

// Let us print the first vertex again

// to show the complete cycle

cout << path[0] << " ";

cout << endl;

}

[8] Hamilton Cycle using Backtracking Algorithm

// Driver Code

int main()

{

/\* Let us create the following graph

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3)-------(4) \*/

bool graph1[V][V] = {{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 1},

{0, 1, 1, 1, 0}};

// Print the solution

hamCycle(graph1);

/\* Let us create the following graph

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3) (4) \*/

bool graph2[V][V] = {{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 0},

{0, 1, 1, 0, 0}};

// Print the solution

hamCycle(graph2);

return 0;

}

output:

Solution Exists: Following is one Hamiltonian Cycle

0 1 2 4 3 0

Solution does not exist