**ADA Manual**

1. **Binary and Linear Search**

import java.util.Arrays;

import java.util.Random;

import java.util.Scanner;

public class SearchProgram {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

Random random = new Random();

System.out.print("Enter the size of the array: ");

int size = scanner.nextInt();

// Generate an array of random numbers

int[] array = generateRandomArray(size, 1, 100, random);

System.out.println("Generated Array: " + Arrays.toString(array));

System.out.println("Choose search type:");

System.out.println("1. Linear Search");

System.out.println("2. Binary Search");

int choice = scanner.nextInt();

long startTime = System.nanoTime(); // Record start time

switch (choice) {

case 1:

System.out.print("Enter the number to search: ");

int linearSearchKey = scanner.nextInt();

int linearSearchResult = linearSearch(array, linearSearchKey);

if (linearSearchResult != -1)

System.out.println(linearSearchKey + " found at index " + linearSearchResult);

else

System.out.println(linearSearchKey + " not found in the array.");

break;

case 2:

// Binary search requires a sorted array

Arrays.sort(array);

System.out.print("Enter the number to search: ");

int binarySearchKey = scanner.nextInt();

int binarySearchResult = binarySearch(array, binarySearchKey);

if (binarySearchResult != -1)

System.out.println(binarySearchKey + " found at index " + binarySearchResult);

else

System.out.println(binarySearchKey + " not found in the array.");

break;

default:

System.out.println("Invalid choice.");

}

long endTime = System.nanoTime(); // Record end time

long duration = endTime - startTime;

System.out.println("Time taken: " + duration + " nanoseconds");

scanner.close();

}

// Linear Search Time Complexity: O(n)

private static int linearSearch(int[] array, int key) {

for (int i = 0; i < array.length; i++) {

if (array[i] == key) {

return i;

}

}

return -1; // Key not found

}

// Binary Search Time Complexity: O(log n)

private static int binarySearch(int[] array, int key) {

int low = 0;

int high = array.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (array[mid] == key)

return mid; // Key found

else if (array[mid] < key)

low = mid + 1;

else

high = mid - 1;

}

return -1; // Key not found

}

private static int[] generateRandomArray(int size, int min, int max, Random random) {

int[] array = new int[size];

for (int i = 0; i < size; i++) {

array[i] = random.nextInt(max - min + 1) + min;

}

return array;

}

}

1. **Sort a given set of elements using the Merge sort method and determine the time required to search an element. Repeat the experiment for different values of n, thenumber of elements in the list to be searched and plot a graph of the time taken versus n.**

import java.util.Arrays;

import java.util.Random;

import java.util.Scanner;

public class MergeSortProgram {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Take input for the size of the array

System.out.print("Enter the size of the array (n): ");

int n = scanner.nextInt();

Random random = new Random();

// Generate an array of random numbers

int[] array = generateRandomArray(n, 1, 100, random);

System.out.println("Generated Array: " + Arrays.toString(array));

long startTime = System.nanoTime(); // Record start time

mergeSort(array, 0, array.length - 1);

long endTime = System.nanoTime(); // Record end time

long duration = endTime - startTime;

System.out.println("Sorted Array: " + Arrays.toString(array));

System.out.println("Time taken for Merge Sort: " + duration + " nanoseconds");

// Close the scanner to prevent resource leak

scanner.close();

}

// Merge Sort Time Complexity: O(n log n)

private static void mergeSort(int[] array, int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

// Recursively sort the two halves

mergeSort(array, left, mid);

mergeSort(array, mid + 1, right);

// Merge the sorted halves

merge(array, left, mid, right);

}

}

private static void merge(int[] array, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

// Create temporary arrays

int[] leftArray = new int[n1];

int[] rightArray = new int[n2];

// Copy data to temporary arrays

System.arraycopy(array, left, leftArray, 0, n1);

System.arraycopy(array, mid + 1, rightArray, 0, n2);

// Merge the temporary arrays

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (leftArray[i] <= rightArray[j]) {

array[k++] = leftArray[i++];

} else {

array[k++] = rightArray[j++];

}

}

// Copy remaining elements if any

while (i < n1) {

array[k++] = leftArray[i++];

}

while (j < n2) {

array[k++] = rightArray[j++];

}

}

private static int[] generateRandomArray(int size, int min, int max, Random random) {

int[] array = new int[size];

for (int i = 0; i < size; i++) {

array[i] = random.nextInt(max - min + 1) + min;

}

return array;

}

}

1. **Obtain the Topological ordering of vertices in a given digraph.**

import java.util.ArrayList;

import java.util.Stack;

class TopologicalSort {

private int V; // Number of vertices

private ArrayList<ArrayList<Integer>> adjList;

TopologicalSort(int v) {

V = v;

adjList = new ArrayList<>(v);

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

adjList.add(new ArrayList<>());

}

void addEdge(int v, int w) {

adjList.get(v).add(w);

}

void topologicalSortUtil(int v, boolean[] visited, Stack<Integer> stack) {

visited[v] = true;

for (Integer neighbor : adjList.get(v)) {

if (!visited[neighbor]) {

topologicalSortUtil(neighbor, visited, stack);

}

}

stack.push(v);

}

void topologicalSort() {

Stack<Integer> stack = new Stack<>();

boolean[] visited = new boolean[V];

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

if (!visited[i]) {

topologicalSortUtil(i, visited, stack);

}

}

System.out.println("Topological Ordering:");

while (!stack.isEmpty()) {

System.out.print(stack.pop() + " ");

}

}

public static void main(String[] args) {

TopologicalSort g = new TopologicalSort(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);

g.topologicalSort();

}

}

1. **Insertion Sort**

import java.util.Arrays;

import java.util.Random;

import java.util.Scanner;

public class InsertionSortExample {

public static void insertionSort(int[] arr) {

int n = arr.length;

for (int i = 1; i < n; ++i) {

int key = arr[i];

int j = i - 1;

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

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the value of n: ");

int n = scanner.nextInt();

Random rand = new Random();

// Generate a random array of size n

int[] elements = new int[n];

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

elements[i] = rand.nextInt(10000);

}

System.out.println("Original array: " + Arrays.toString(elements));

// Measure the time taken to sort the array

long startTime = System.nanoTime();

insertionSort(elements);

long endTime = System.nanoTime();

double sortTime = (endTime - startTime) / 1e9; // Convert to seconds

System.out.printf("n=%d, Sort Time: %.6f seconds%n", n, sortTime);

System.out.println("Sorted array: " + Arrays.toString(elements));

}

}

1. **Horspool algorithm for String matching.**

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class HorspoolAlgorithm {

public static int horspoolSearch(String text, String pattern) {

int textLength = text.length();

int patternLength = pattern.length();

if (patternLength > textLength) {

return -1; // Pattern cannot be longer than the text

}

Map<Character, Integer> shiftTable = preprocessShiftTable(pattern);

int i = patternLength - 1;

while (i < textLength) {

int k = 0;

while (k < patternLength && pattern.charAt(patternLength - 1 - k) == text.charAt(i - k)) {

k++;

}

if (k == patternLength) {

return i - patternLength + 1; // Match found

} else {

char mismatchChar = text.charAt(i);

int shift = shiftTable.containsKey(mismatchChar) ? shiftTable.get(mismatchChar) : patternLength;

i += shift;

}

}

return -1; // No match found

}

private static Map<Character, Integer> preprocessShiftTable(String pattern) {

Map<Character, Integer> shiftTable = new HashMap<>();

int patternLength = pattern.length();

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

char currentChar = pattern.charAt(i);

int shift = patternLength - i - 1;

shiftTable.put(currentChar, shift);

}

return shiftTable;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the text: ");

String text = scanner.nextLine();

System.out.print("Enter the pattern: ");

String pattern = scanner.nextLine();

int result = horspoolSearch(text, pattern);

if (result != -1) {

System.out.println("Pattern found at index " + result);

} else {

System.out.println("Pattern not found in the text");

}

}

}

1. **0/1 Knapsack Problem**

import java.util.Scanner;

class Knapsack

{

public static void main(String[] args)

{

Scanner sc=new Scanner(System.in);

int object,m;

System.out.println("Enter the Total Objects");

object=sc.nextInt();

int weight[]=new int[object];

int profit[]=new int[object];

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

{

System.out.println("Enter the Profit");

profit[i]=sc.nextInt();

System.out.println("Enter the weight");

weight[i]=sc.nextInt();

}

System.out.println("Enter the Knapsack capacity");

m=sc.nextInt();

double p\_w[]=new double[object];

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

{

p\_w[i]=(double)profit[i]/(double)weight[i];

}

System.out.println("");

System.out.println(" ");

System.out.println("-----Data-Set ");

System.out.print(" ");

System.out.println("");

System.out.print("Objects");

for(int i=1;i<=object;i++)

{

System.out.print(i+" ");

}

System.out.println();

System.out.print("Profit ");

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

{

System.out.print(profit[i]+" ");

}

System.out.println();

System.out.print("Weight ");

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

{

System.out.print(weight[i]+" ");

}

System.out.println();

System.out.print("P/W ");

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

{

System.out.print(p\_w[i]+" ");

}

for(int i=0;i<object-1;i++)

{

for(int j=i+1;j<object;j++)

{

if(p\_w[i]<p\_w[j])

{

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

double temp=p\_w[j];

p\_w[j]=p\_w[i];

p\_w[i]=temp;

int temp1=profit[j];

profit[j]=profit[i];

profit[i]=temp1;

int temp2=weight[j];

weight[j]=weight[i];

weight[i]=temp2;

}

}

}

System.out.println("");

System.out.println(" ");

System.out.println("--After Arranging--");

System.out.print(" ");

");

System.out.println("");

System.out.print("Objects");

for(int i=1;i<=object;i++)

{

System.out.print(i+" ");

}

System.out.println();

System.out.print("Profit ");

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

{

System.out.print(profit[i]+" ");

}

System.out.println();

System.out.print("Weight ");

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

{

System.out.print(weight[i]+" ");

}

System.out.println();

System.out.print("P/W ");

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

{

System.out.print(p\_w[i]+" ");

}

int k=0; double sum=0;

System.out.println();

while(m>0)

{

if(weight[k]<m)

{

sum+=1\*profit[k];

m=m-weight[k];

}

else

{

double x4=m\*profit[k];

double x5=weight[k];

double x6=x4/x5;

sum=sum+x6;

m=0;

} k++;

}

System.out.println("Final Profit is="+sum);

}

}

1. **Kruskal’s algorithm**

import java.util.Scanner;

public class KRUSKAL

{

public static void main(String[] args)

{

int cost[][]=new int[10][10];

int i, j,mincost=0;

Scanner in = new Scanner(System.in);

System.out.println("\*\*\*\*\*\*\*\*\* KRUSKAL'S ALGORITHM \*\*\*\*\*\*\*");

System.out.println("Enter the number of nodes: ");

int n = in.nextInt();

System.out.println("Enter the cost matrix");

for(i=1;i<=n;i++){

for(j=1;j<=n;j++){

cost[i][j] = in.nextInt();

}

}

System.out.println("The entered cost matrix is");

for(i=1;i<=n;i++){

for(j=1;j<=n;j++){

System.out.print(cost[i][j]+"\t");

}

System.out.println();

}

mincost=kruskals(n,mincost,cost);

System.out.println("The minimum spanning tree cost is:");

System.out.println(mincost);

}

static int kruskals(int n,int mincost,int cost[][] )

{

int ne = 1,a=0,u=0,b=0,v=0,min;

int parent[]=new int[10];

while(ne < n){

min=999;

for(int i=1; i<=n; i++)

{

for(int j=1; j<=n; j++)

{

if(cost[i][j] < min){

min = cost[i][j];

a=u=i;

b=v=j;

}

}

}

while(parent[u]>0) u = parent[u];

while(parent[v]>0) v = parent[v];

if(u != v)

{

System.out.print((ne++)+">minimum edge is :");

System.out.println("("+a+","+b+") and its cost is:"+min);

mincost += min;

parent[v] = u;

}

cost[a][b] = cost[b][a] = 999;

}

return mincost; } }

1. **Floyd’s algorithm**

import java.util.Scanner;

class Floyd {

public static void main(String[] args)

{

int a[][]=new int[10][10];

int i, j;

Scanner in = new Scanner(System.in); System.out.println("\*\*\*\*\*\*\*\*\*\*\*FLOYD'SALGORITHM\*\*\*\*\*\*\*\*\*\*");

System.out.println("Enter the number of vertices: ");

int n = in.nextInt();

System.out.println("Enter the adjacency matrix");

for (i=1;i<=n;i++)

for (j=1;j<=n;j++)

a[i][j] = in.nextInt();

System.out.println("Entered adjacency matrix is: ");

for(i=1;i<=n;i++)

{

for(j=1; j<=n; j++)

{

System.out.print(a[i][j]+"\t");

}

System.out.println();

}

floyd(a,n);

System.out.println("All pair shortest path matrix:");

for (i=1; i<=n; i++)

{

for (j=1; j<=n; j++)

System.out.print(a[i][j]+"\t");

System.out.println();

}

}

static void floyd(int a[][],int n)

{

for (int k=1; k<=n; k++)

{

for (int i=1; i<=n; i++)

for (int j=1; j<=n; j++)

a[i][j] = min(a[i][j], a[i][k] + a[k][j]);

}

}

static int min(int a,int b)

{

if(a>b) return b;

else return a;

} }

1. **BFS Method**

import java.util.\*;

public class BFSExample {

public static void bfs(int startNode, Map<Integer, List<Integer>> graph) {

Set<Integer> visited = new HashSet<>();

Queue<Integer> queue = new LinkedList<>();

visited.add(startNode);

queue.add(startNode);

System.out.println("Nodes reachable from node " + startNode + " using BFS:");

while (!queue.isEmpty()) {

int current = queue.poll();

System.out.print(current + " ");

List<Integer> neighbors = graph.getOrDefault(current, Collections.emptyList());

for (int neighbor : neighbors) {

if (!visited.contains(neighbor)) {

visited.add(neighbor);

queue.add(neighbor);

}

}

}

System.out.println();

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of nodes in the graph: ");

int numNodes = scanner.nextInt();

Map<Integer, List<Integer>> graph = new HashMap<>();

System.out.println("Enter the directed edges (node1 -> node2): ");

for (int i = 0; i < numNodes; i++) {

System.out.print("Enter the number of edges for node " + i + ": ");

int numEdges = scanner.nextInt();

List<Integer> neighbors = new ArrayList<>();

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

System.out.print("Enter neighbor for node " + i + ": ");

int neighbor = scanner.nextInt();

neighbors.add(neighbor);

}

graph.put(i, neighbors);

}

System.out.print("Enter the starting node for BFS: ");

int startNode = scanner.nextInt();

bfs(startNode, graph);

}

}

1. **DFS method**

import java.util.\*;

public class DFSExample {

public static void dfs(int current, Map<Integer, List<Integer>> graph, Set<Integer> visited) {

visited.add(current);

System.out.print(current + " ");

List<Integer> neighbors = graph.getOrDefault(current, Collections.emptyList());

for (int neighbor : neighbors) {

if (!visited.contains(neighbor)) {

dfs(neighbor, graph, visited);

}

}

}

public static void performDFS(Map<Integer, List<Integer>> graph) {

Set<Integer> visited = new HashSet<>();

for (int node : graph.keySet()) {

if (!visited.contains(node)) {

dfs(node, graph, visited);

}

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of nodes in the undirected graph: ");

int numNodes = scanner.nextInt();

Map<Integer, List<Integer>> graph = new HashMap<>();

System.out.println("Enter the edges (node1 - node2): ");

for (int i = 0; i < numNodes; i++) {

System.out.print("Enter the number of neighbors for node " + i + ": ");

int numNeighbors = scanner.nextInt();

List<Integer> neighbors = new ArrayList<>();

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

System.out.print("Enter neighbor for node " + i + ": ");

int neighbor = scanner.nextInt();

neighbors.add(neighbor);

}

graph.put(i, neighbors);

}

System.out.println("DFS traversal starting from each node:");

performDFS(graph);

}

}

1. **Prim’s algorithm**

import java.util.Scanner;

public class PRIM

{

public static void main(String[] args)

{

int cost[][]=new int[10][10];

int i, j, mincost = 0;

Scanner in = new Scanner(System.in);

System.out.println("\*\*\*\*\*\*\*\*\* PRIMS ALGORITHM \*\*\*\*\*\*\*\*\*");

System.out.println("Enter the number of nodes");

int n = in.nextInt();

System.out.println("Enter the cost matrix");

for(i=1; i<=n; i++)

{

for(j=1; j<=n; j++){

cost[i][j] = in.nextInt();

}

}

System.out.println("The entered cost matrix is");

for(i=1; i<=n; i++)

{

for(j=1; j<=n; j++)

{

System.out.print(cost[i][j]+"\t");

}

System.out.println();

}

System.out.println("Minimum Spanning Tree Edges and costs are"); mincost=prims(cost,n,mincost);

System.out.print("The minimum spanning tree cost is:");

System.out.print(+mincost);

}

static int prims(int cost[][],int n,int mincost)

{

int nearV[]=new int[10],t[][]=new int[10][3],u = 0,i,j,k;

for(i=2; i<=n; i++)

nearV[i]=1;

nearV[1]=0;

for(i=1; i<n; i++)

{

int min=999;

for(j=1;j<=n;j++)

{

if(nearV[j]!=0 && cost[j][nearV[j]]<min)

{

min=cost[j][nearV[j]];

u=j;

}

}

t[i][1] = u;

t[i][2] = nearV[u];

mincost += min;

nearV[u] = 0;

for(k=1; k<=n; k++){

if(nearV[k] != 0 && cost[k][nearV[k]] > cost[k][u]) nearV[k] = u;

}

System.out.print(i+") Minimum edge is ("+t[i][1]);

System.out.println(","+t[i][2]+") and its cost is :"+min);

}

return mincost;

}

}

1. **N Queens’s Problem**

public class NQueenProblem {

final int N = 4;

void printSolution(int board[][]) {

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

System.out.print(" " + board[i][j] + " ");

System.out.println();

}

}

boolean isSafe(int board[][], int row, int col) {

int i, j;

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

if (board[row][i] == 1)

return false;

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

if (board[i][j] == 1)

return false;

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

if (board[i][j] == 1)

return false;

return true;

}

boolean solveNQUtil(int board[][], int col) {

if (col >= N)

return true;

for (int i = 0; i < N; i++) {

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

boolean solveNQ() {

int board[][] = { { 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 } };

if (!solveNQUtil(board, 0)) {

System.out.print("Solution does not exist");

return false;

}

printSolution(board);

return true;

}

public static void main(String args[]) {

NQueenProblem Queen = new NQueenProblem();

Queen.solveNQ();

}

}