Insertion sort:

import java.util.Arrays;

public class InsertionSort {

public static void insertionSort(int[] arr) {

int n = arr.length;

// Traverse through the array starting from the second element

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

int key = arr[i]; // Current element to be inserted into the sorted part

int j = i - 1;

// Move elements of arr[0..i-1], that are greater than key,

// to one position ahead of their current position

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

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

j--;

}

// Insert the current element into the correct position

arr[j + 1] = key;

// Print the array after each pass

System.out.println("After pass " + i + ": " + Arrays.toString(arr));

}

}

public static void main(String[] args) {

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

System.out.println("Initial Array: " + Arrays.toString(arr));

// Perform Insertion Sort and print array after each pass

insertionSort(arr);

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

}

}

selection sort:

import java.util.Arrays;

public class SelectionSort {

public static void selectionSort(int[] arr) {

int n = arr.length;

// Traverse through the array

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

int minIndex = i;

// Find the index of the minimum element in the unsorted part

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

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Swap the minimum element with the current element (at index i)

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

// Print the array after each pass

System.out.println("After pass " + (i + 1) + ": " + Arrays.toString(arr));

}

}

public static void main(String[] args) {

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

System.out.println("Initial Array: " + Arrays.toString(arr));

// Perform Selection Sort and print array after each pass

selectionSort(arr);

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

}

}

Quick sort

import java.util.Arrays;

public class QuickSort {

private static int quickSortCalls = 0;

private static void quickSort(int[] arr, int low, int high) {

if (low < high) {

quickSortCalls++; // Increment the call counter for each recursive call

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

// Recursively sort elements before and after the partition

quickSort(arr, low, partitionIndex - 1);

quickSort(arr, partitionIndex + 1, high);

}

}

private static int partition(int[] arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

// Swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

i++;

// Swap arr[i+1] and arr[high] (pivot)

int temp = arr[i];

arr[i] = arr[high];

arr[high] = temp;

return i;

}

public static void main(String[] args) {

int[] arr = {64, 25, 12, 22, 11, 7};

System.out.println("Initial Array: " + Arrays.toString(arr));

// Reset the call counter

quickSortCalls = 0;

// Perform Quick Sort and count the number of calls

quickSort(arr, 0, arr.length - 1);

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

System.out.println("Number of calls to Quick Sort: " + quickSortCalls);

}

}

Merge Sort  
  
import java.util.Arrays;

public class MergeSort {

private static int mergeSortCalls = 0;

private static void mergeSort(int[] arr, int low, int high) {

if (low < high) {

mergeSortCalls++; // Increment the call counter for each recursive call

int mid = low + (high - low) / 2;

// Recursively sort the two halves

mergeSort(arr, low, mid);

mergeSort(arr, mid + 1, high);

// Merge the sorted halves

merge(arr, low, mid, high);

}

}

private static void merge(int[] arr, int low, int mid, int high) {

// Create temporary arrays to hold the two halves

int[] left = Arrays.copyOfRange(arr, low, mid + 1);

int[] right = Arrays.copyOfRange(arr, mid + 1, high + 1);

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

// Merge the two halves back into the original array

while (i < left.length && j < right.length) {

if (left[i] <= right[j]) {

arr[k++] = left[i++];

} else {

arr[k++] = right[j++];

}

}

// Copy any remaining elements from the left half

while (i < left.length) {

arr[k++] = left[i++];

}

// Copy any remaining elements from the right half

while (j < right.length) {

arr[k++] = right[j++];

}

}

public static void main(String[] args) {

int[] arr = {64, 25, 12, 22, 11, 7};

System.out.println("Initial Array: " + Arrays.toString(arr));

// Reset the call counter

mergeSortCalls = 0;

// Perform Merge Sort and count the number of calls

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

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

System.out.println("Number of calls to Merge Sort: " + mergeSortCalls);

}

}

Prims MST :

// Prim's Algorithm in Java

import java.util.\*;

class PGraph {

public void Prim(int G[][], int V) {

int INF = 9999999;

int no\_edge; // number of edge

boolean[] selected = new boolean[V];

Arrays.fill(selected, false);

no\_edge = 0;

selected[0] = true;

System.out.println("Edge : Weight");

while (no\_edge < V - 1) {

int min = INF;

int x = 0; // row number

int y = 0; // col number

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

if (selected[i] == true) {

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

// not in selected and there is an edge

if (!selected[j] && G[i][j] != 0) {

if (min > G[i][j]) {

min = G[i][j];

x = i;

y = j;

}

}

}

}

}

System.out.println(x + " - " + y + " : " + G[x][y]);

selected[y] = true;

no\_edge++;

}

}

public static void main(String[] args) {

PGraph g = new PGraph();

Scanner sc = new Scanner(System.in);

// number of vertices in grapj

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

int V = sc.nextInt();

int[][] G = new int[V][V];

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

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

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

G[i][j] = sc.nextInt();

}

}

g.Prim(G, V);

}

}

kruskal’s algo:

import java.util.\*;

public class KruskalsMST {

static class Edge {

int src, dest, weight;

public Edge(int src, int dest, int weight)

{

this.src = src;

this.dest = dest;

this.weight = weight;

}

}

static class Subset {

int parent, rank;

public Subset(int parent, int rank)

{

this.parent = parent;

this.rank = rank;

}

}

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

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

int V = scanner.nextInt();

System.out.print("Enter the number of edges: ");

int E = scanner.nextInt();

List<Edge> graphEdges = new ArrayList<Edge>();

System.out.println("Enter the edges (src, dest, weight)");

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

int src = scanner.nextInt();

int dest = scanner.nextInt();

int weight = scanner.nextInt();

graphEdges.add(new Edge(src, dest, weight));

}

scanner.close();

graphEdges.sort(new Comparator<Edge>() {

@Override public int compare(Edge o1, Edge o2)

{

return o1.weight - o2.weight;

}

});

kruskals(V, graphEdges);

}

private static void kruskals(int V, List<Edge> edges)

{

int j = 0;

int noOfEdges = 0;

// Allocate memory for creating V subsets

Subset subsets[] = new Subset[V];

// Allocate memory for results

Edge results[] = new Edge[V];

// Create V subsets with single elements

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

subsets[i] = new Subset(i, 0);

}

while (noOfEdges < V - 1) {

Edge nextEdge = edges.get(j);

int x = findRoot(subsets, nextEdge.src);

int y = findRoot(subsets, nextEdge.dest);

if (x != y) {

results[noOfEdges] = nextEdge;

union(subsets, x, y);

noOfEdges++;

}

j++;

}

System.out.println("Following are the edges of the constructed MST:");

int minCost = 0;

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

System.out.println(results[i].src + " -- "+ results[i].dest + "=="+ results[i].weight);

minCost += results[i].weight;

}

System.out.println("Total cost of MST: " + minCost);

}

private static void union(Subset[] subsets, int x, int y)

{

int rootX = findRoot(subsets, x);

int rootY = findRoot(subsets, y);

if (subsets[rootY].rank < subsets[rootX].rank) {

subsets[rootY].parent = rootX;

}

else if (subsets[rootX].rank

< subsets[rootY].rank) {

subsets[rootX].parent = rootY;

}

else {

subsets[rootY].parent = rootX;

subsets[rootX].rank++;

}

}

private static int findRoot(Subset[] subsets, int i)

{

if (subsets[i].parent == i)

return subsets[i].parent;

subsets[i].parent

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

return subsets[i].parent;

}

}

fractional knapsack:

import java.util.\*;

// Class to represent an item

class Item {

int value;

int weight;

public Item(int value, int weight) {

this.value = value;

this.weight = weight;

}

}

public class FractionalKnapsack {

// Function to solve Fractional Knapsack problem

public static void fractionalKnapsack(Item[] items, int capacity) {

// Sort items based on value-to-weight ratio (descending order)

Arrays.sort(items, (a, b) -> Double.compare((double)b.value / b.weight, (double)a.value / a.weight));

double totalProfit = 0.0;

List<Double> selectedWeights = new ArrayList<>();

for (Item item : items) {

if (capacity <= 0) {

break;

}

int currentWeight = Math.min(item.weight, capacity);

double currentProfit = (double)currentWeight \* ((double)item.value / item.weight);

selectedWeights.add((double)currentWeight);

totalProfit += currentProfit;

capacity -= currentWeight;

}

// Print the solution vector (selected weights)

System.out.println("Selected weights: " + selectedWeights);

// Print total profit earned

System.out.println("Total profit earned: " + totalProfit);

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of items: ");

int n = scanner.nextInt();

System.out.print("Enter the capacity of the knapsack: ");

int capacity = scanner.nextInt();

// Input the items (value, weight)

Item[] items = new Item[n];

System.out.println("Enter the value and weight of each item:");

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

int value = scanner.nextInt();

int weight = scanner.nextInt();

items[i] = new Item(value, weight);

}

// Solve the Fractional Knapsack problem and print the solution

fractionalKnapsack(items, capacity);

scanner.close();

}

}

0/1 knapsack:  
  
import java.util.\*;

// Class to represent an item

class Item {

int value;

int weight;

public Item(int value, int weight) {

this.value = value;

this.weight = weight;

}

}

public class ZeroOneKnapsack {

// Function to solve 0/1 knapsack problem

public static void zeroOneKnapsack(Item[] items, int n, int capacity) {

// Initialize the table to store the maximum value that can be obtained

// for each combination of items and capacities

int[][] table = new int[n + 1][capacity + 1];

// Populate the table using dynamic programming

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

for (int j = 1; j <= capacity; j++) {

// If the current item can be included in the knapsack

if (items[i - 1].weight <= j) {

// Choose the maximum value between including and excluding the current item

table[i][j] = Math.max(items[i - 1].value + table[i - 1][j - items[i - 1].weight], table[i - 1][j]);

} else {

// If the current item cannot be included, the value remains the same as the previous row

table[i][j] = table[i - 1][j];

}

}

}

// Backtrack to find which items were included in the knapsack

int[] selectedItems = new int[n];

int remainingCapacity = capacity;

for (int i = n; i > 0 && remainingCapacity > 0; i--) {

if (table[i][remainingCapacity] != table[i - 1][remainingCapacity]) {

selectedItems[i - 1] = 1;

remainingCapacity -= items[i - 1].weight;

} else {

selectedItems[i - 1] = 0;

}

}

// Print the solution vector

System.out.println("Solution vector:");

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

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

}

System.out.println();

// Print total profit earned

System.out.println("Total profit earned: " + table[n][capacity]);

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of items: ");

int n = scanner.nextInt();

System.out.print("Enter the capacity of the knapsack: ");

int capacity = scanner.nextInt();

// Input the items (value, weight)

Item[] items = new Item[n];

System.out.println("Enter the value and weight of each item:");

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

int value = scanner.nextInt();

int weight = scanner.nextInt();

items[i] = new Item(value, weight);

}

zeroOneKnapsack(items, n, capacity);

scanner.close();

}

}

LCS:  
  
import java.io.\*;

import java.util.\*;

public class LongestCommonSubsequence {

int lcs(String X, String Y, int m, int n, StringBuilder lcsString) {

if (m == 0 || n == 0) {

return 0;

}

if (X.charAt(m - 1) == Y.charAt(n - 1)) {

// Characters match, include this character in the LCS

lcsString.append(X.charAt(m - 1));

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

} else {

// Characters do not match, recursively compute LCS length

int lcs1 = lcs(X, Y, m, n - 1, lcsString);

int lcs2 = lcs(X, Y, m - 1, n, lcsString);

// Return the maximum length of the two recursive calls

return Math.max(lcs1, lcs2);

}

}

public static void main(String[] args) {

LongestCommonSubsequence lcs = new LongestCommonSubsequence();

String S1 = "AGGTAB";

String S2 = "GXTXAYB";

int m = S1.length();

int n = S2.length();

StringBuilder lcsString = new StringBuilder();

int length = lcs(S1, S2, m, n, lcsString);

System.out.println("Length of LCS is: " + length);

System.out.println("Longest Common Subsequence is: " + lcsString.reverse().toString());

}

}

Dijkstra’s Algorithm:

import java.util.\*;

public class ShortestPath {

static final int V = 9; // Number of vertices in the graph

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

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

int minDistance(int dist[], boolean sptSet[]) {

int min = Integer.MAX\_VALUE;

int minIndex = -1;

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

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

min = dist[v];

minIndex = v;

}

}

return minIndex;

}

// A utility function to print the final distance (D) and predecessor (Pi) matrices

void printSolution(int dist[], int pred[], int source) {

System.out.println("Final Distance (D) Matrix:");

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

System.out.println("From " + source + " to " + i + ": " + dist[i]);

}

System.out.println("\nPredecessor (Pi) Matrix:");

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

System.out.println("Predecessor of " + i + ": " + pred[i]);

}

// Print paths from source vertex to all other vertices

System.out.println("\nShortest Paths from Source Vertex " + source + ":");

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

if (i != source) {

System.out.print("Path from " + source + " to " + i + ": ");

printPath(pred, source, i);

System.out.println(" (Cost: " + dist[i] + ")");

}

}

}

// A utility function to print the path using the predecessor matrix (Pi)

void printPath(int pred[], int source, int destination) {

LinkedList<Integer> path = new LinkedList<>();

int step = destination;

path.add(step);

while (pred[step] != -1) {

path.addFirst(pred[step]);

step = pred[step];

}

path.addFirst(source);

Iterator<Integer> iterator = path.iterator();

while (iterator.hasNext()) {

System.out.print(iterator.next());

if (iterator.hasNext()) {

System.out.print(" -> ");

}

}

}

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

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

int dist[] = new int[V]; // The output array to hold the shortest distance from src to i

int pred[] = new int[V]; // The array to hold the predecessors

// sptSet[i] will be true if vertex i is included in the shortest path tree

boolean sptSet[] = new boolean[V];

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

Arrays.fill(dist, Integer.MAX\_VALUE);

Arrays.fill(pred, -1);

// 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

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++) {

if (!sptSet[v] && graph[u][v] != 0 && dist[u] != Integer.MAX\_VALUE

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

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

pred[v] = u;

}

}

}

// Print the final D and Pi matrices and paths

printSolution(dist, pred, src);

}

// Driver's code

public static void main(String[] args) {

// Create the adjacency matrix of the graph

int graph[][] = new int[][] {

{ 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 }

};

int source = 0; // Source vertex

ShortestPath sp = new ShortestPath();

sp.dijkstra(graph, source);

}

}

Floyd warshall:

import java.util.Arrays;

public class FloydWarshall {

static final int INF = Integer.MAX\_VALUE; // Represents infinity (no direct path)

public static void main(String[] args) {

int[][] graph = {

{0, 3, INF, 7},

{8, 0, 2, INF},

{5, INF, 0, 1},

{2, INF, INF, 0}

};

int n = graph.length;

// Call the Floyd-Warshall algorithm

floydWarshall(graph, n);

}

// Function to implement Floyd-Warshall algorithm and print results

public static void floydWarshall(int[][] graph, int n) {

int[][] D = new int[n][n]; // Distance matrix

int[][] Pi = new int[n][n]; // Predecessor matrix

// Initialize D and Pi matrices based on the graph

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

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

D[i][j] = graph[i][j];

if (i != j && graph[i][j] != INF) {

Pi[i][j] = i;

} else {

Pi[i][j] = -1; // No predecessor

}

}

}

// Floyd-Warshall algorithm to find all pairs shortest paths

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

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

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

if (D[i][k] != INF && D[k][j] != INF && D[i][k] + D[k][j] < D[i][j]) {

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

Pi[i][j] = Pi[k][j];

}

}

}

}

// Print the final distance matrix D

System.out.println("Final Distance Matrix (D):");

printMatrix(D);

// Print the predecessor matrix Pi

System.out.println("\nPredecessor Matrix (Pi):");

printMatrix(Pi);

// Print paths between every pair of vertices

System.out.println("\nShortest Paths between Every Pair of Vertices:");

printAllPairShortestPaths(D, Pi);

}

// Function to print a 2D matrix

public static void printMatrix(int[][] matrix) {

for (int[] row : matrix) {

System.out.println(Arrays.toString(row));

}

}

// Function to print shortest paths between every pair of vertices

public static void printAllPairShortestPaths(int[][] D, int[][] Pi) {

int n = D.length;

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

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

if (i != j && D[i][j] != INF) {

System.out.print("Shortest path from " + i + " to " + j + ": ");

printPath(i, j, Pi);

System.out.println(" (Distance = " + D[i][j] + ")");

}

}

}

}

// Function to print the path from vertex u to v using the predecessor matrix Pi

public static void printPath(int u, int v, int[][] Pi) {

if (u == v) {

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

} else if (Pi[u][v] == -1) {

System.out.println("No path from " + u + " to " + v + " exists.");

} else {

printPath(u, Pi[u][v], Pi);

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

}

}

}

sum of subsets:

public class SubsetSumCount {

public static int countSubsetSumCalls(int[] arr, int n, int sum) {

// Base cases

if (sum == 0) {

return 1; // One way to achieve sum 0 (empty subset)

}

if (n == 0 || sum < 0) {

return 0; // No possible subsets with negative sum or no elements

}

// Count of subsets excluding the current element

int exclude = countSubsetSumCalls(arr, n - 1, sum);

// Count of subsets including the current element

int include = countSubsetSumCalls(arr, n - 1, sum - arr[n - 1]);

return exclude + include;

}

public static void main(String[] args) {

int[] arr = {3, 2, 7, 1, 4, 5};

int sum = 6;

int totalCalls = countSubsetSumCalls(arr, arr.length, sum);

System.out.println("Total number of recursive calls: " + totalCalls);

}

}

KMP string matching

class KMP\_String\_Matching {

void KMPSearch(String pat, String txt)

{

int M = pat.length();

int N = txt.length();

// create lps[] that will hold the longest

// prefix suffix values for pattern

int lps[] = new int[M];

int j = 0; // index for pat[]

// Preprocess the pattern (calculate lps[]

// array)

computeLPSArray(pat, M, lps);

int i = 0; // index for txt[]

while ((N - i) >= (M - j)) {

if (pat.charAt(j) == txt.charAt(i)) {

j++;

i++;

}

if (j == M) {

System.out.println("Found pattern "

+ "at index " + (i - j));

j = lps[j - 1];

}

// mismatch after j matches

else if (i < N

&& pat.charAt(j) != txt.charAt(i)) {

// Do not match lps[0..lps[j-1]] characters,

// they will match anyway

if (j != 0)

j = lps[j - 1];

else

i = i + 1;

}

}

}

void computeLPSArray(String pat, int M, int lps[])

{

// length of the previous longest prefix suffix

int len = 0;

int i = 1;

lps[0] = 0; // lps[0] is always 0

// the loop calculates lps[i] for i = 1 to M-1

while (i < M) {

if (pat.charAt(i) == pat.charAt(len)) {

len++;

lps[i] = len;

i++;

}

else // (pat[i] != pat[len])

{

// This is tricky. Consider the example.

// AAACAAAA and i = 7. The idea is similar

// to search step.

if (len != 0) {

len = lps[len - 1];

}

else // if (len == 0)

{

lps[i] = len;

i++;

}

}

}

}

public static void main(String args[])

{

String txt = "ABABDABACDABABCABAB";

String pat = "ABABCABAB";

new KMP\_String\_Matching().KMPSearch(pat, txt);

}

}

note:

public static void printPath(int[] pred, int source, int destination) {

// Base case: If the destination is the source vertex

if (destination == source) {

System.out.print(source);

} else if (pred[destination] == -1) {

// If there's no predecessor (unreachable)

System.out.print("No path from " + source + " to " + destination);

} else {

// Recursively print the path from source to predecessor of destination

printPath(pred, source, pred[destination]);

System.out.print(" -> " + destination);

}

}