## Binary search :

* 1. **import** java.util.Scanner; 2.

3. **class** BinarySearch 4. {

5. **public static void** main(String args[]) 6. {

#### 7. **int** c, first, last, middle, n, search, array[]; 8.

1. Scanner in = **new** Scanner(System.in);
2. System.out.println("Enter number of elements");
3. n = in.nextInt();
4. array = **new int**[n]; 13.

#### 14. System.out.println("Enter " + n + " integers"); 15.

16.

17. **for** (c = 0; c < n; c++)

18. array[c] = in.nextInt(); 19.

1. System.out.println("Enter value to find");
2. search = in.nextInt(); 22.
3. first = 0;
4. last = n - 1;
5. middle = (first + last)/2; 26.

27. **while**( first <= last )

28. {

1. **if** ( array[middle] < search )
2. first = middle + 1;
3. **else if** ( array[middle] == search )

#### 32. {

1. System.out.println(search + " found at location " + (middle + 1) + ".");
2. **break**;

35. }

1. **else**
2. last = middle - 1; 38.

39. middle = (first + last)/2;

40. }

41. **if** (first > last)

|  |  |  |
| --- | --- | --- |
| 42.  the | System.out.println(search + " isn't present  list.**\n**"); | in |
| 43. | } |  |
| 44. | } |  |

**Linear search :**

* **import** java.util.Scanner;

#### 

* **class** LinearSearch

#### {

* **public static void** main(String args[])

#### {

* **int** c, n, search, array[];



* Scanner in = **new** Scanner(System.in);
* System.out.println("Enter number of elements");
* n = in.nextInt();
* array = **new int**[n];

#### 

* System.out.println("Enter " + n + " integers");



* **for** (c = 0; c < n; c++)
* array[c] = in.nextInt();



* System.out.println("Enter value to find");
* search = in.nextInt();



* **for** (c = 0; c < n; c++)
* {
* **if** (array[c] == search) */\* Searching element is present \*/*

#### {

* System.out.println(search + " is present at location "

+ (c + 1) + ".");

* **break**;
* }
* }
* **if** (c == n) */\* Element to search isn't present \*/*

#### System.out.println(search + " isn't present in array.");

* }

**2.**

public class HeapSort

{

public void sort(int arr[])

{

int n = arr.length;

// 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 int temp = arr[0];

arr[0] = arr[i]; arr[i] = temp;

// call max heapify on the reduced heap heapify(arr, i, 0);

}

}

// 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)

{

int swap = arr[i]; arr[i] = arr[largest]; arr[largest] = swap;

// Recursively heapify the affected sub-tree heapify(arr, n, largest);

}

}

/\* A utility function to print array of size n \*/ static void printArray(int arr[])

{

int n = arr.length; for (int i=0; i<n; ++i)

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

}

// Driver program

public static void main(String args[])

{

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

int n = arr.length;

HeapSort ob = new HeapSort(); ob.sort(arr);

System.out.println("Sorted array is"); printArray(arr);

}

}

# 3.

class MergeSort

{

// 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)

{

// Find sizes of two subarrays to be merged int n1 = m - l + 1;

int n2 = r - m;

/\* Create temp arrays \*/ int L[] = new int [n1]; int R[] = new int [n2];

/\*Copy data to temp arrays\*/ for (int i=0; i<n1; ++i)

L[i] = arr[l + i]; for (int j=0; j<n2; ++j)

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

/\* Merge the temp arrays \*/

// Initial indexes of first and second subarrays int i = 0, j = 0;

// Initial index of merged subarry array int k = l;

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

}

else

{

} k++;

}

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

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

/\* Copy remaining elements of L[] if any \*/ while (i < n1)

{

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

k++;

}

/\* Copy remaining elements of R[] if any \*/ while (j < n2)

{

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

k++;

}

}

// Main function that sorts arr[l..r] using

// merge()

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

{

if (l < r)

{

// Find the middle point int m = (l+r)/2;

// Sort first and second halves sort(arr, l, m);

sort(arr , m+1, r);

// Merge the sorted halves merge(arr, l, m, r);

}

}

# 4.

class SelectionSort

{

void sort(int arr[])

{

int n = arr.length;

// One by one move boundary of unsorted subarray for (int i = 0; i < n-1; i++)

{

// Find the minimum element in unsorted array int min\_idx = i;

for (int j = i+1; j < n; j++) if (arr[j] < arr[min\_idx])

min\_idx = j;

// Swap the found minimum element with the first

// element

int temp = arr[min\_idx]; arr[min\_idx] = arr[i]; arr[i] = temp;

}

}

// Prints the array

void printArray(int arr[])

{

int n = arr.length; for (int i=0; i<n; ++i)

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

}

// Driver code to test above

public static void main(String args[])

{

SelectionSort ob = new SelectionSort(); int arr[] = {64,25,12,22,11};

ob.sort(arr); System.out.println("Sorted array"); ob.printArray(arr);

}

}

# 5.

class Graph

{

private int V; // No. of vertices

private LinkedList<Integer> adj[]; // Adjacency List

//Constructor Graph(int v)

{

V = v;

adj = new LinkedList[v]; for (int i=0; i<v; ++i)

adj[i] = new LinkedList();

}

// Function to add an edge into the graph void addEdge(int v,int w) { adj[v].add(w); }

// A recursive function used by topologicalSort void topologicalSortUtil(int v, boolean visited[],

Stack stack)

{

// Mark the current node as visited. visited[v] = true;

Integer i;

// Recur for all the vertices adjacent to this

// vertex

Iterator<Integer> it = adj[v].iterator(); while (it.hasNext())

{

i = it.next();

if (!visited[i])

topologicalSortUtil(i, visited, stack);

}

// Push current vertex to stack which stores result stack.push(new Integer(v));

}

// The function to do Topological Sort. It uses

// recursive topologicalSortUtil() void topologicalSort()

{

Stack stack = new Stack();

// Mark all the vertices as not visited boolean visited[] = new boolean[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)

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

}

// Driver method

public static void main(String args[])

{

// Create a graph given in the above diagram Graph g = new Graph(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);

System.out.println("Following is a Topological " +

"sort of the given graph"); g.topologicalSort();

}

}

**Output :**

Following is a Topological Sort of the given graph 5 4 2 3 1 0

# 6.

class AllPairShortestPath

{

final static int INF = 99999, V = 4;

void floydWarshall(int graph[][])

{

int dist[][] = new int[V][V]; int i, j, k;

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

for (j = 0; j < V; j++) dist[i][j] = graph[i][j];

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];

}

}

}

printSolution(dist);

}

void printSolution(int dist[][])

{

System.out.println("The following matrix shows the shortest "+ "distances between every pair of vertices");

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

{

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

{

if (dist[i][j]==INF) System.out.print("INF ");

else

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

}

System.out.println();

}

}

// Driver program to test above function public static void main (String[] args)

{

/\* Let us create the following weighted graph 10

(0)------->(3)

| /|\

5 | |

| | 1

\|/ |

(1)------->(2)

3 \*/

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

{INF, 0, 3, INF},

{INF, INF, 0, 1},

{INF, INF, INF, 0}

};

AllPairShortestPath a = new AllPairShortestPath();

// Print the solution a.floydWarshall(graph);

}

}

### Output:

Following matrix shows the shortest distances between every pair of vertices 0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0

# 7.

/\* A Naive recursive implementation of 0-1 Knapsack problem \*/ class Knapsack

{

// A utility function that returns maximum of two integers static 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 static 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 public static void main(String args[])

{

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

int wt[] = new int[]{10, 20, 30}; int W = 50;

int n = val.length; System.out.println(knapSack(W, wt, val, n));

}

}

#### Output:

220

# 8.

import java.util.Scanner; //Scanner Function to take in the Input Values public class Dijkstra

{

static Scanner scan; // scan is a Scanner Object

public static void main(String[] args)

{

int[] preD = new int[5];

int min = 999, nextNode = 0; // min holds the minimum value, nextNode holds the value for the next node.

scan = new Scanner(System.in);

int[] distance = new int[5]; // the distance matrix int[][] matrix = new int[5][5]; // the actual matrix int[] visited = new int[5]; // the visited array

System.out.println("Enter the cost matrix"); for (int i = 0; i < distance.length; i++)

{

visited[i] = 0; //initialize visited array to zeros preD[i] = 0;

for (int j = 0; j < distance.length; j++)

{

matrix[i][j] = scan.nextInt(); //fill the matrix if (matrix[i][j]==0)

matrix[i][j] = 999; // make the zeros as 999

}

}

distance = matrix[0]; //initialize the distance array visited[0] = 1; //set the source node as visited

distance[0] = 0; //set the distance from source to source to zero which is the starting point

for (int counter = 0; counter < 5; counter++)

{

min = 999;

for (int i = 0; i < 5; i++)

{

if (min > distance[i] && visited[i]!=1)

{

min = distance[i]; nextNode = i;

}

}

visited[nextNode] = 1;

for (int i = 0; i < 5; i++)

{

if (visited[i]!=1)

{

if (min+matrix[nextNode][i] < distance[i])

{

distance[i] = min+matrix[nextNode][i]; preD[i] = nextNode;

}

}

}

}

for(int i = 0; i < 5; i++) System.out.print("|" + distance[i]);

System.out.println("|"); int j;

for (int i = 0; i < 5; i++)

{

if (i!=0)

{

System.out.print("Path = " + i); j = i;

do

{

j = preD[j]; System.out.print(" <- " + j);

}

while(j != 0);

}

System.out.println();

}

}

}

# 9.

class QuickSort

{

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

{

int pivot = arr[high];

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

}

}

// swap arr[i+1] and arr[high] (or pivot) int temp = arr[i+1];

arr[i+1] = arr[high]; arr[high] = temp;

return i+1;

}

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

{

if (low < high)

{

/\* pi is partitioning index, arr[pi] is now at right place \*/

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

// Recursively sort elements before

// partition and after partition sort(arr, low, pi-1);

sort(arr, pi+1, high);

}

}

/\* A utility function to print array of size n \*/ static void printArray(int arr[])

{

int n = arr.length; for (int i=0; i<n; ++i)

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

}

// Driver program

public static void main(String args[])

{

int arr[] = {10, 7, 8, 9, 1, 5};

int n = arr.length;

QuickSort ob = new QuickSort(); ob.sort(arr, 0, n-1);

System.out.println("sorted array"); printArray(arr);

}

}

# 10.

**11.**

import java.io.\*; import java.util.\*;

class Graph

{

private int V; // No. of vertices

private LinkedList<Integer> adj[]; //Adjacency Lists

// Constructor Graph(int v)

{

V = v;

adj = new LinkedList[v]; for (int i=0; i<v; ++i)

adj[i] = new LinkedList();

}

// Function to add an edge into the graph void addEdge(int v,int w)

{

adj[v].add(w);

}

// prints BFS traversal from a given source s void BFS(int s)

{

boolean visited[] = new boolean[V];

// Create a queue for BFS

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

// Mark the current node as visited and enqueue it visited[s]=true;

queue.add(s);

while (queue.size() != 0)

{

// Dequeue a vertex from queue and print it s = queue.poll();

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

Iterator<Integer> i = adj[s].listIterator(); while (i.hasNext())

{

int n = i.next(); if (!visited[n])

{

visited[n] = true; queue.add(n);

}

}

}

}

// Driver method to

public static void main(String args[])

{

Graph g = new Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

System.out.println("Following is Breadth First Traversal "+"(starting from vertex 2)");

g.BFS(2);

}

}

#### Output:

Following is Breadth First Traversal (starting from vertex 2) 2 0 3 1

# 12.