Design & Analysis of Algorithms (322COMP-3)

LABORATORY MANUAL

DEPARTMENT OF COMPUTER SCIENCE
College of Computer Science & Information Technology
Jazan University
2023-2024

Course Coordinator: Mrs.P.Jayasuriya

TABLE OF CONTENTS

PROCEDURE TO INSTALL/EXECUTE THE SOFTWARE IN LAB

1.1 How to Install NetBeans on Windows

Step 0: Install JDK

To use Net Beans for Java programming, you need to first install Java Development Kit (JDK). See "JDK - How to Install".

Step 1: Download

Download "NetBeans IDE" installer from http://netbeans.org/downloads/index.html. There are many "bundles" available. For beginners, choose the 1st entry "Java SE" (e.g., "netbeans-8.2-javase-windows.exe" 95MB).

Step 2: Run the Installer

Run the downloaded installer.

1.2 How to Install NetBeans on Mac OS X

To use NetBeans for Java programming, you need to first install JDK. Read "How to install JDK on Mac".

To install NetBeans:

1. Download NetBeans from http://netbeans.org/downloads/. Set "Platform" to "Mac OS X". There are many "bundles" available. For beginners, choose "Java SE" (e.g., "netbeans-8.2-javase-macosx.dmg" 116MB).
2. Double-click the download Disk Image (DMG) file.
3. Double-click the "NetBeans 8.x.mpkg", and follow the instructions to install NetBeans. NetBeans will be installed under "/Applications/NetBeans".
4. Eject the Disk Image (".dmg").

You can launch NetBeans from the "Applications".

Notes: To uninstall NetBeans, drag the "/Applications/NetBeans" folder to trash.

Creating a Java Application Project

1. Choose File > New Project. Under Categories, select Java. Under Projects, select Java Application. Click Next.

2. Under Project Name, type MyApp. Make sure the Project Location is set to NetBeans Projects.

3. (Optional) Check the Use Dedicated Folder for Storing Libraries checkbox.

4. Enter acrostic .Main as the main class.

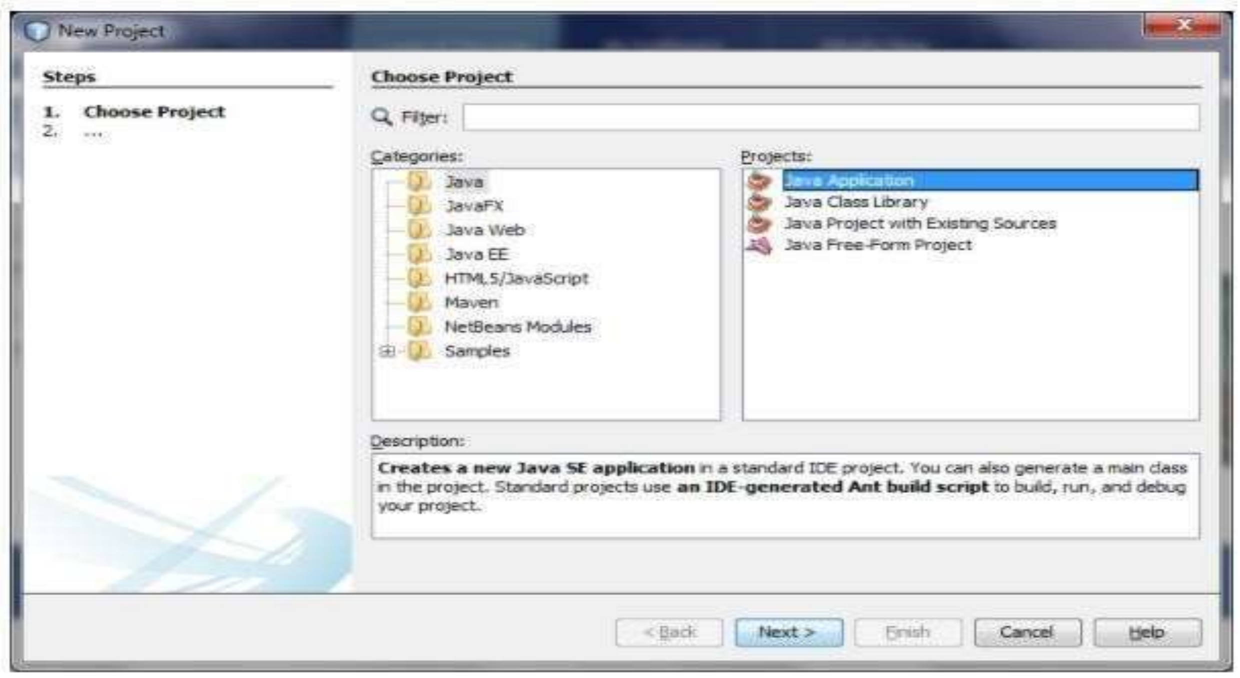
5. Ensure that the Create Main Class checkbox is checked.

6. Click Finish. The MyApp project is displayed in the Project window and Main.java opens in the Source Editor.

1. Create Your First Java Project

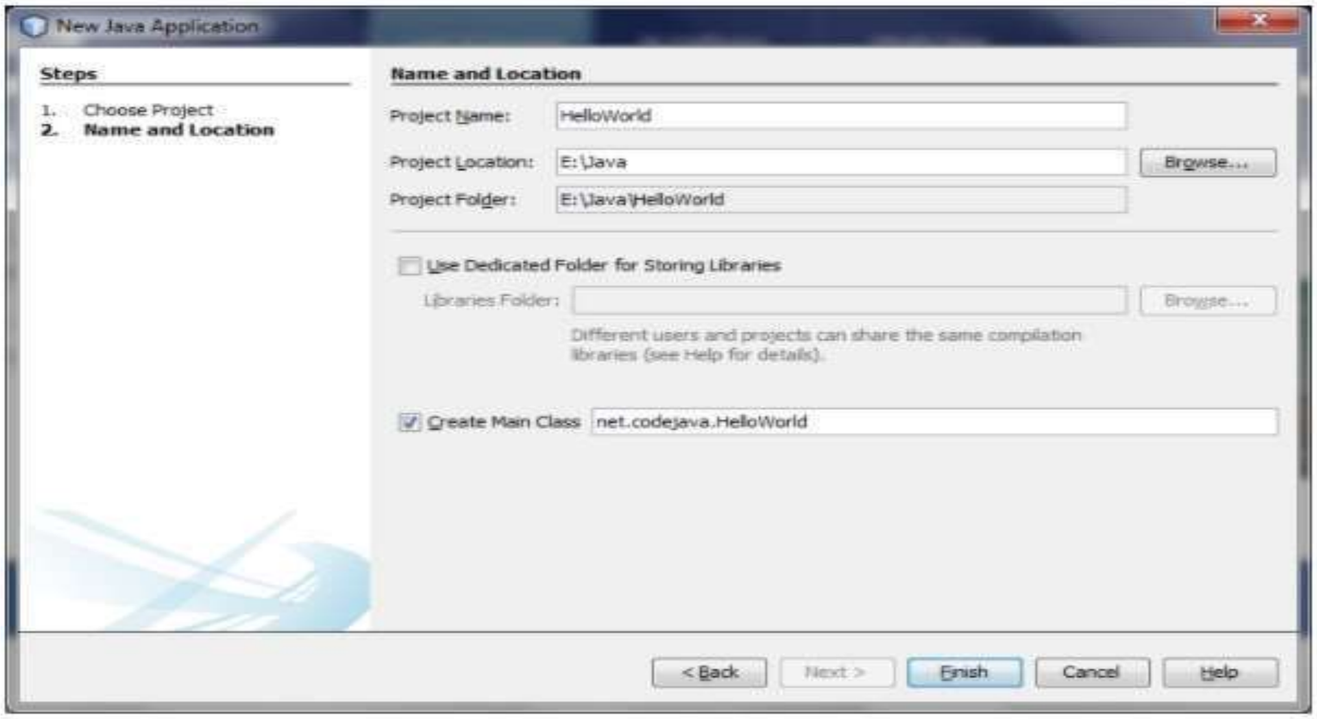
Now, let's create a Java project using NetBeans IDE. Go to menu **File > New Project…**

Under the *New Project* dialog, choose Java application as shown in the following screenshot:



Creates a new Java SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.

Click Next to advance to the next step. In the New Java Application screen, type Project Name, specify
Project Location and the main class:

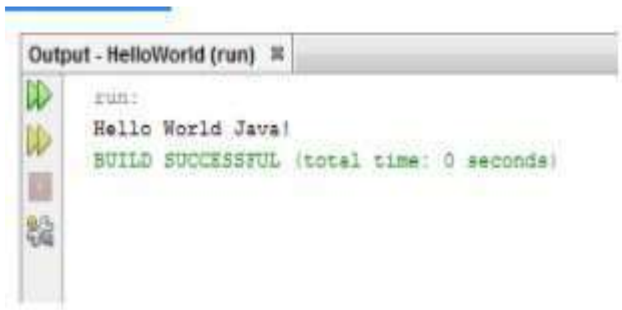


Project Name: HelloWorld
Project Location: E:\Java
Project Folder: E:\Java\HelloWorld
Use Dedicated Folder for Storing Libraries
Libraries Folder:
Different users and projects can share the same compilation libraries (see Help for details).
Create Main Class: net.codejava.HelloWorld

Run Your First Java Program

To run the HelloWorld program above, there are several ways:

- Go to menu **Run > Run Project <ProjectName>**
- Click **Run Project** icon in the toolbar.
- Press **F6** key.
- Right click in the code editor, and select **Run File** (or press **Shift + F6**). You should



```
run:
Hello World Java!
BUILD SUCCESSFUL (total time: 0 seconds)
```

PROGRAM: 1

Write a Program in Java to find the GCD (Greatest Common Divisor) of any two non-negative numbers.

The greatest common divisor of two nonnegative, not-both-zero integers m and n, denoted gcd(m, n), is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero. Euclid of Alexandria (third century BC.) outlined an algorithm for solving this problem in one of the volumes of his Elements most famous for its systematic exposition of geometry. In modern terms, Euclid's algorithm is based on applying repeatedly the equality gcd(m, n) = gcd(n, m mod n),where m mod n is the remainder of the division of m by n, until m mod n is equal to 0. Since gcd(m,0) = m (why?), the last value of m is also the greatest common divisor of the initial m and n.

For example, gcd(60, 24) can be computed as follows:

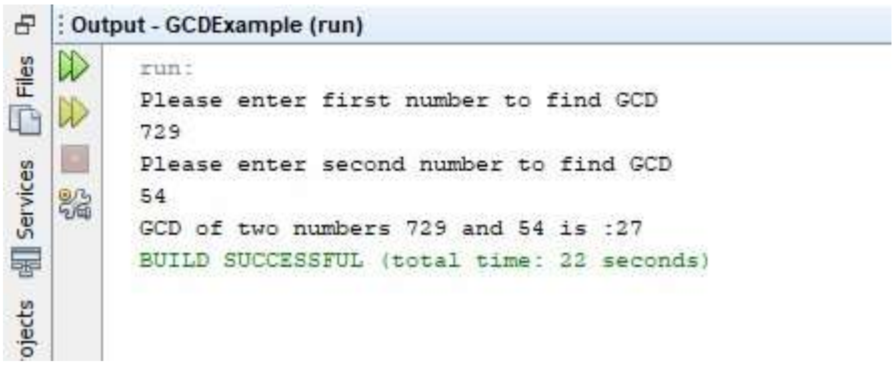
gcd(60, 24) = gcd(24, 12) = gcd(12, 0) = 12.

Euclid's algorithm for computing gcd(m, n)

Step 1 If n = 0, return the value of m as the answer and stop; otherwise, proceed to Step 2.
Step 2 Divide m by n and assign the value of the remainder to r.
Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.
Alternatively, we can express the same algorithm in pseudocode:

```
ALGORITHM     Euclid(m, n)
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers' m and n
//Output: Greatest common divisor of m and n
 while n ≠ 0 do
r ← m mod n
m ← n
 n ← r return m
```

```
package gcdexample;
import java.util.Scanner;
public class GCDExample
{
public static void main(String args[])
{
//Enter two number whose GCD needs to be calculated.
Scanner scanner = new Scanner(System.in);
System.out.println("Please enter first number to find GCD");
int number1 = scanner.nextInt();
System.out.println("Please enter second number to find GCD");
int number2 = scanner.nextInt();
System.out.println("GCD of two numbers " + number1 +" and "+ number2 +" is :" + findGCD(number1,number2));
}
private static int findGCD(int number1, int number2)
{
if(number2 == 0)
{
return number1;
}
return findGCD(number2, number1%number2);
} }
```

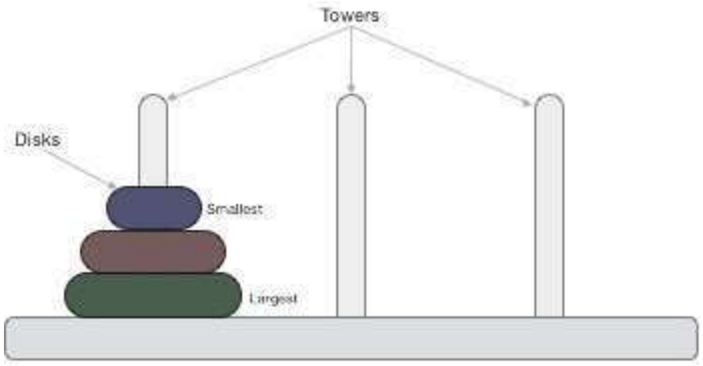


```
run:
Please enter first number to find GCD
729
Please enter second number to find GCD
54
GCD of two numbers 729 and 54 is :27
BUILD SUCCESSFUL (total time: 22 seconds)
```

PROGRAM: 2

Write a Program in Java for Tower of Hanoi Problem for Five disks.

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted



These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Tower of Hanoi puzzle with n disks can be solved in minimum **2^n-1** steps. This presentation shows that a puzzle with 3 disks has taken **$2^3 - 1 = 7$** steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.

So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.

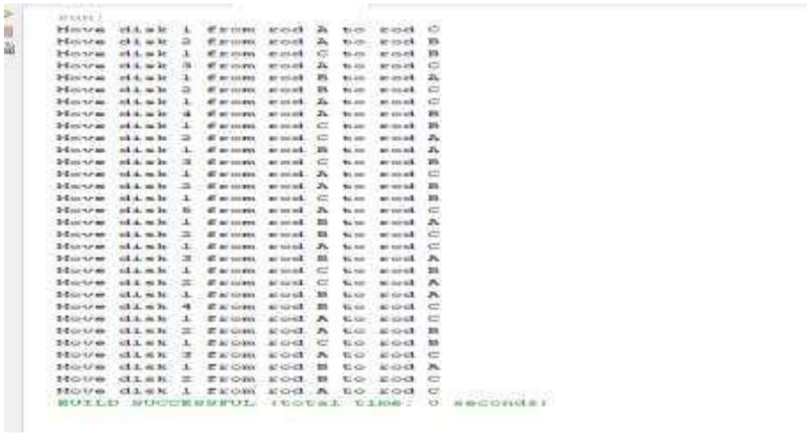
Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are −

Step 1 − Move n-1 disks from source to aux
Step 2 − Move nth disk from source to dest
Step 3 − Move n-1 disks from aux to dest
A recursive algorithm for Tower of Hanoi can be driven as follows –

```
START
Procedure Hanoi(disk, source, dest, aux)

  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)     // Step 1
    move disk from source to dest          // Step 2
    Hanoi(disk - 1, aux, dest, source)     // Step 3
  END IF
  END Procedure
STOP
```

```
package toh;
public class TOH {
// Java recursive function to solve tower of hanoi puzzle
static void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
if (n == 1)
{
System.out.println("Move disk 1 from rod " +  from_rod + " to rod " + to_rod);
return;
}
towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
System.out.println("Move disk " + n + " from rod " +  from_rod + " to rod " + to_rod);
towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}
//  Driver method
public static void main(String args[])
{
int n = 5; // Number of disks
towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
}
}
```



```
run:
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
Move disk 4 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 2 from rod C to rod A
Move disk 1 from rod B to rod A
Move disk 3 from rod C to rod B
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 5 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
Move disk 3 from rod B to rod A
Move disk 1 from rod C to rod B
Move disk 2 from rod C to rod A
Move disk 1 from rod B to rod A
Move disk 4 from rod B to rod C
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
BUILD SUCCESSFUL (total time: 0 seconds)
```

PROGRAM: 3

Write a Program in Java to implement Bubble Sort algorithm for sorting a list of integers in ascending order.

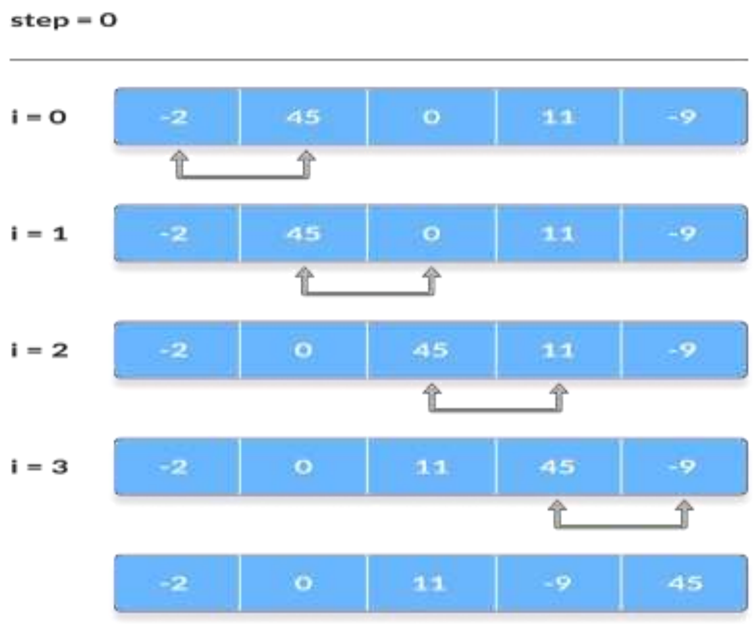
Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are not in the intended order.
Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

Working of Bubble Sort

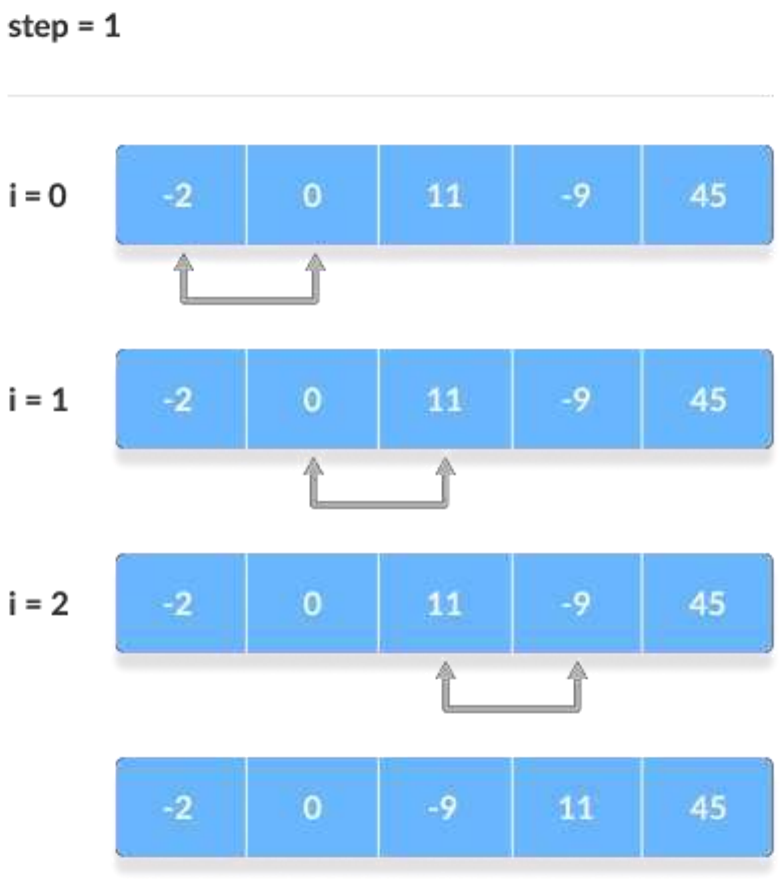
Suppose we are trying to sort the elements in **ascending order**.

1. First Iteration (Compare and Swap)

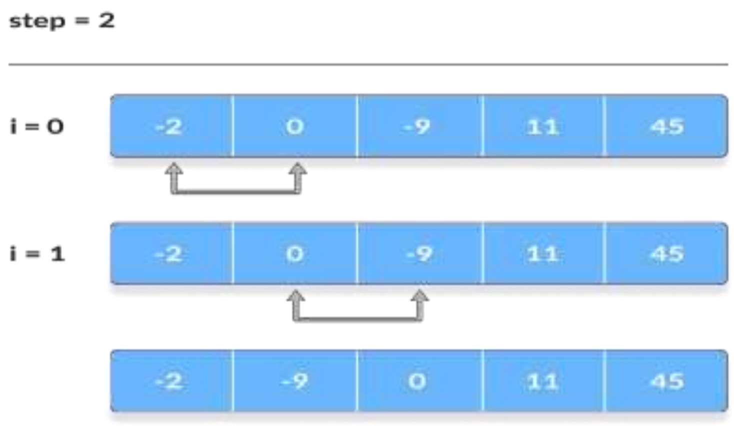
1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.



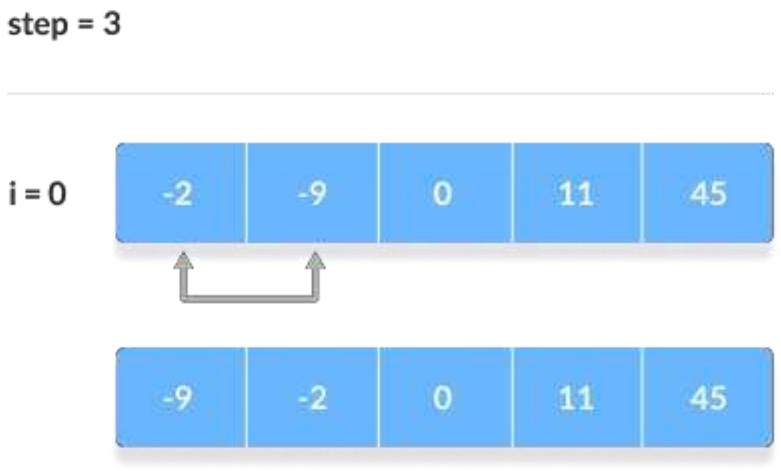
2. Remaining Iteration
The same process goes on for the remaining iterations.
After each iteration, the largest element among the unsorted elements is placed at the end.



In each iteration, the comparison takes place up to the last unsorted element

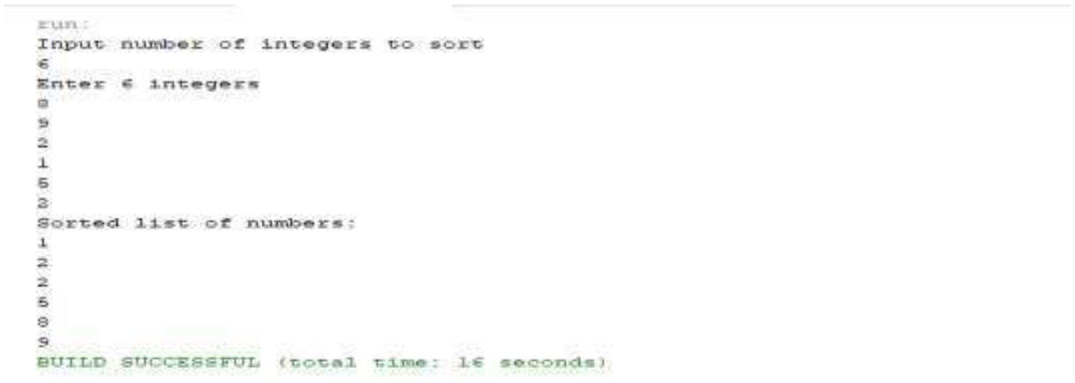


The array is sorted when all the unsorted elements are placed at their correct positions.



The array is sorted if all elements are kept in the right order

```
package bubblesort;
import java.util.Scanner;
public class BubbleSort {
public static void main(String []args)
{
int n, c, d, swap;
Scanner in = new Scanner(System.in);
System.out.println("Input number of integers to sort");
n = in.nextInt();
int array[] = new int[n];
System.out.println("Enter " + n + " integers");
for (c = 0; c < n; c++)
array[c] = in.nextInt();
for(c =0;c<(n - 1);c++)
{
for (d = 0; d < n - c - 1;d++)
{
if (array[d] > array[d+1])/*For descending order use*/
{
swap = array[d];
array[d]= array[d+1];
array[d+1]=swap;
}
}
}
System.out.println("Sorted list of numbers:");
for (c = 0; c < n; c++)
System.out.println(array[c]);
}
}
```



```
run:
Input number of integers to sort
6
Enter 6 integers
8
9
2
1
5
2
Sorted list of numbers:
1
2
2
5
8
9
BUILD SUCCESSFUL (total time: 16 seconds)
```

PROGRAM: 4

Write a Java program to implement Selection sort algorithm for sorting a list of integers in ascending order.

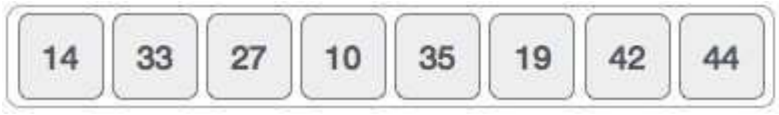
Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

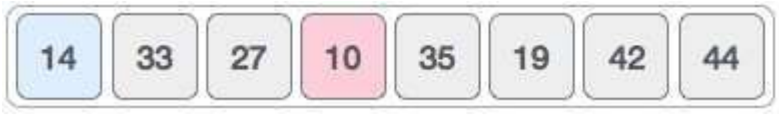
This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n^2), where **n** is the number of items.

How Selection Sort Works?

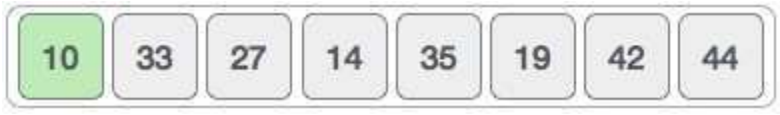
Consider the following depicted array as an example.



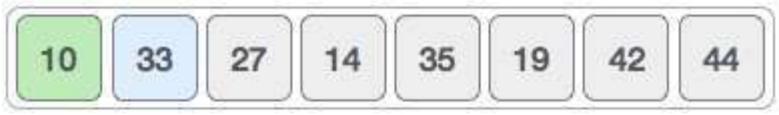
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



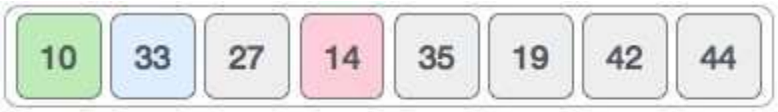
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



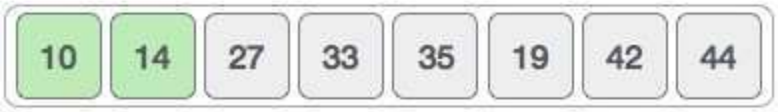
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

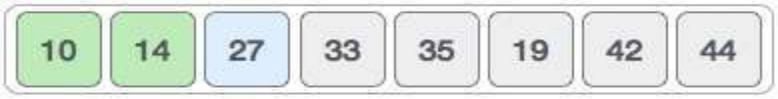


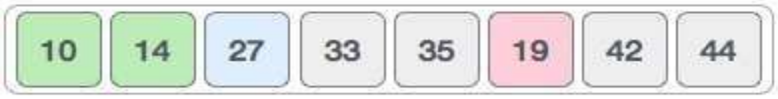
After two iterations, two least values are positioned at the beginning in a sorted manner.

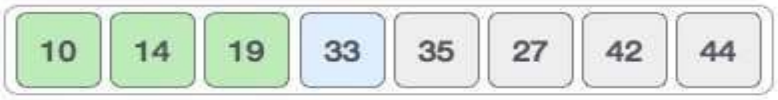


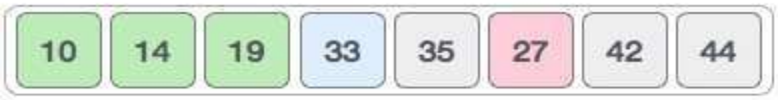
The same process is applied to the rest of the items in the array.

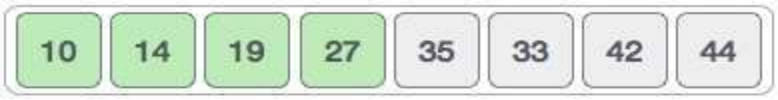
Following is a pictorial depiction of the entire sorting process −

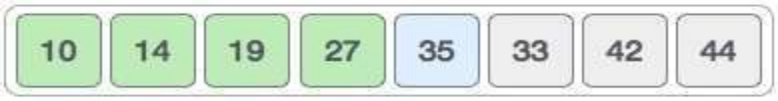


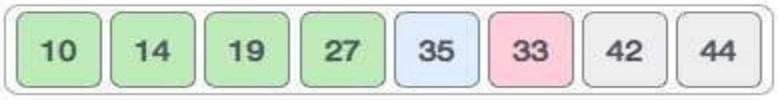


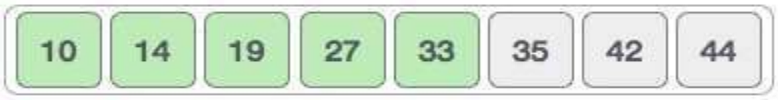


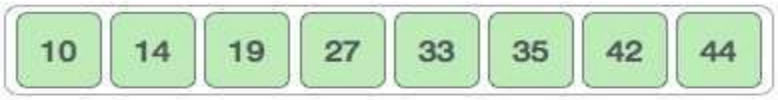










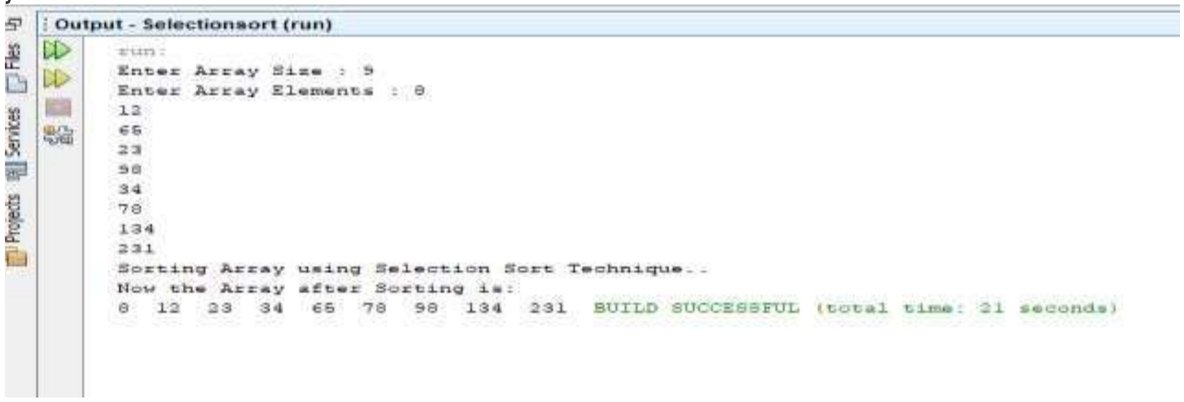


Algorithm

```
Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted
```

```java
package selectionsort;
import java.util.Scanner;
public class Selectionsort
{
    public static void main(String args[])
   {
       int size, i, j, temp;
       int arr[] = new int[50];
       Scanner scan = new Scanner(System.in);

       System.out.print("Enter Array Size : ");
       size = scan.nextInt();

       System.out.print("Enter Array Elements : ");
       for(i=0; i<size; i++)
       {
          arr[i] = scan.nextInt();
       }
       System.out.print("Sorting Array using Selection Sort Technique..\n");
       for(i=0; i<size; i++)
       {
          for(j=i+1; j<size; j++)
          {
             if(arr[i] > arr[j])
             {
                temp = arr[i];
                arr[i]  =  arr[j];
                arr[j] = temp;
             }
          }
```

```
        }
System.out.print("Now the Array after Sorting is:\n");
        for(i=0; i<size; i++)
        {
            System.out.print(arr[i]+ "  ");
        }
    }
}
```



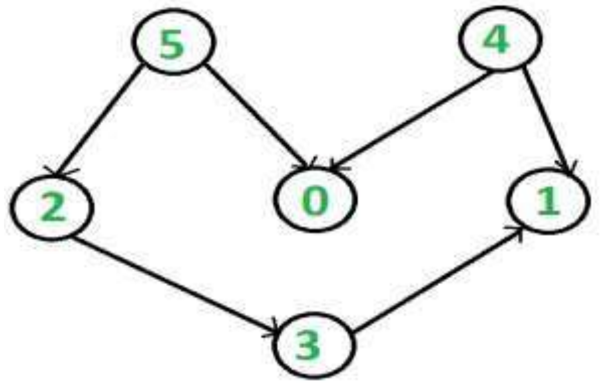
Output - Selectionsort (run)

```
run:
Enter Array Size : 9
Enter Array Elements : 8
12
65
23
98
34
78
134
231
Sorting Array using Selection Sort Technique..
Now the Array after Sorting is:
8  12  23  34  65  78  98  134  231  BUILD SUCCESSFUL (total time: 21 seconds)
```

PROGRAM: 5

Write a Java program to implement Topological sorting for the given Directed acyclic graph.

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG. For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



DFS Traversal Technique Algorithm to find Topological sort:

Step 1: Select any vertex and start exploring it till you get the sink vertex.

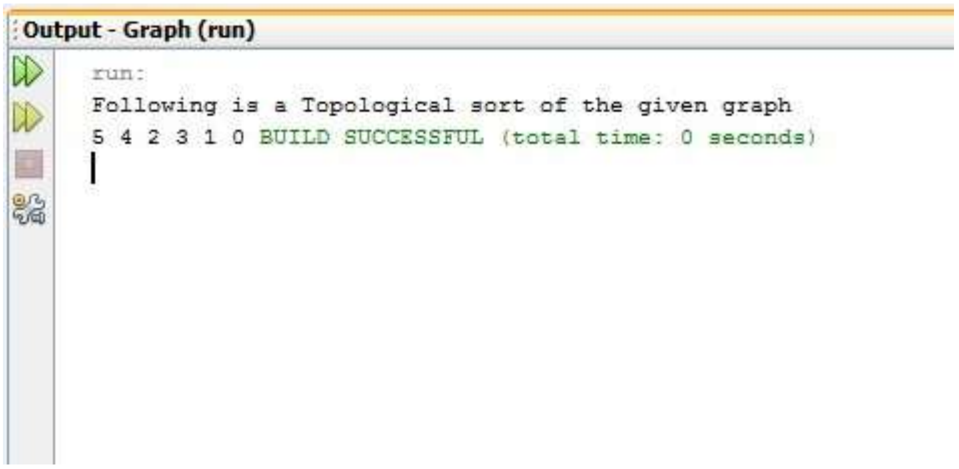
Step 2: Put that sink vertex in the stack and disconnect that vertex from the graph.

Step 3: Backtrack and search for the next sink vertex along the search path.

Step 4: Repeat steps 2, 3 until all the vertices in the graph are visited. (If you’re done with exploring a vertex and wants to choose another you can always start with the smallest number and explore its neighbors first)

Step 5: Then perform the pop operation from the stack and print the popped elements from the stack which will give you the topological ordering.

```
package graph;
//A Java program to print topological sorting of a DAG
import java.io.*;
import java.util.*;

// This class represents a directed graph
// using adjacency list representation
class Graph {
        // No. of vertices
        private int V;

        // Adjacency List as ArrayList of ArrayList's
        private ArrayList<ArrayList<Integer> > adj;

        // Constructor
        Graph(int v)
        {
                V = v;
                adj = new ArrayList<ArrayList<Integer> >(v);
                for (int i = 0; i < v; ++i)
                        adj.add(new ArrayList<Integer>());
        }

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

        // A recursive function used by topologicalSort
        void topologicalSortUtil(int v, boolean visited[],Stack<Integer> stack)
        {
                // Mark the current node as visited.
                visited[v] = true;
                Integer i;

                // Recur for all the vertices adjacent
                // to thisvertex
                Iterator<Integer> it = adj.get(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<Integer> stack = new Stack<Integer>();

                // 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 code
        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");
                // Function Call
                g.topologicalSort();
        }
}
```



Output - Graph (run)

```
run:
Following is a Topological sort of the given graph
5 4 2 3 1 0 BUILD SUCCESSFUL (total time: 0 seconds)
```

PROGRAM: 6

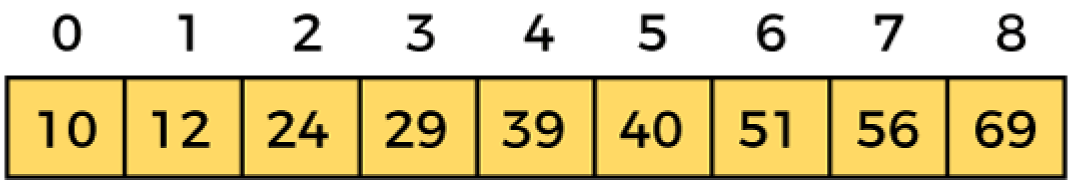
Write a Java program for implementing Binary Search.

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -



Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

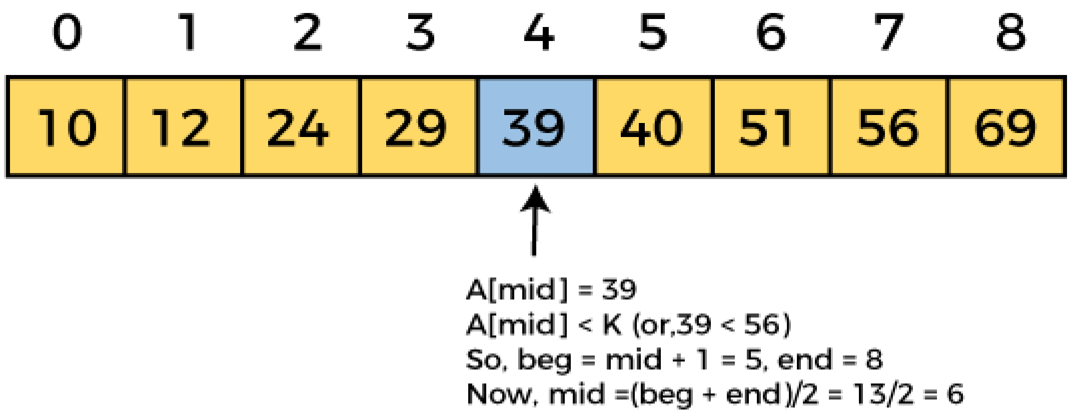
mid = (beg + end)/2

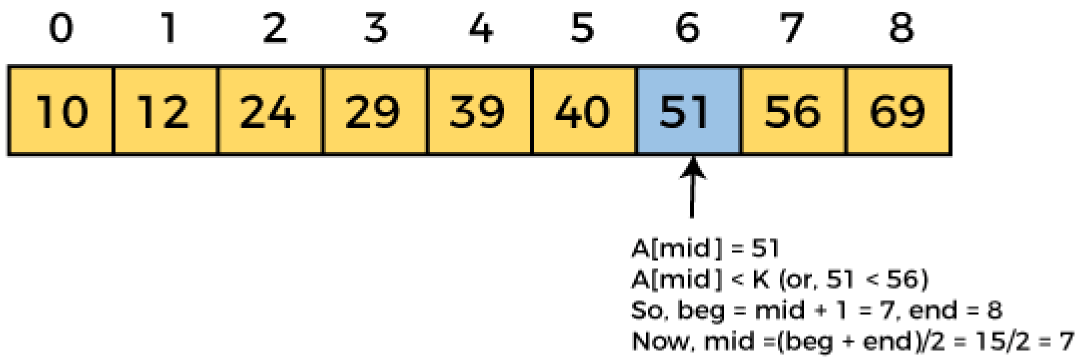
So, in the given array -

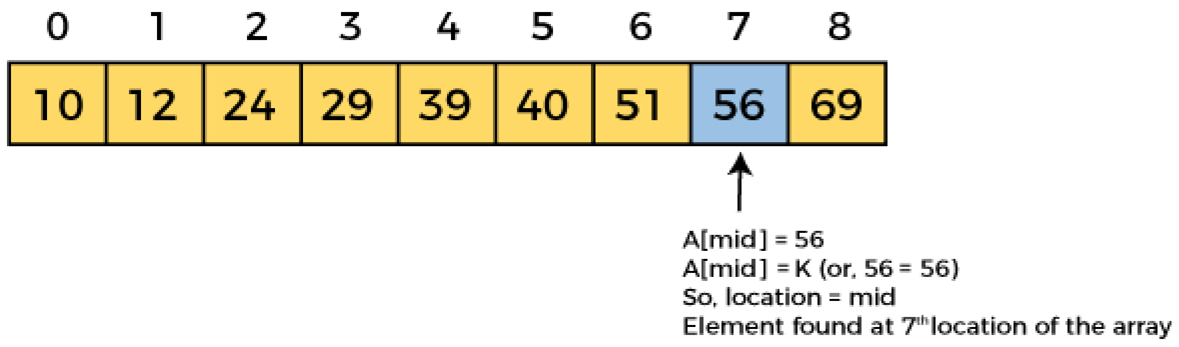
beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.







Now, the element to search is found. So algorithm will return the index of the element matched.

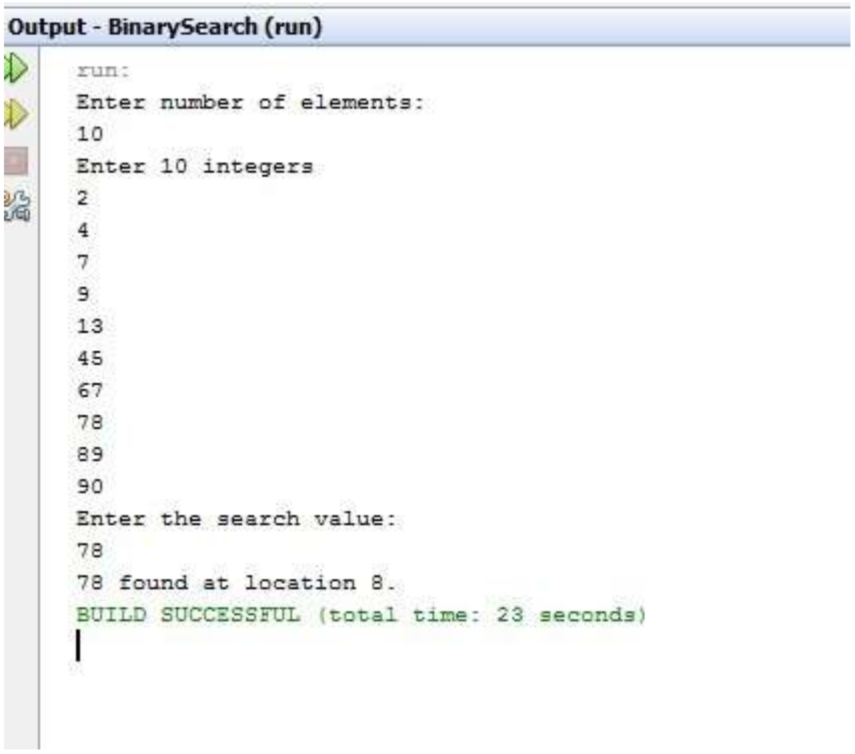
```
1.  Binary_Search(a, lower_bound, upper_bound, val)
    // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_boun
    d' is the index of the last array element, 'val' is the value to search
2.  Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
3.  Step 2: repeat steps 3 and 4 while beg <=end
4.  Step 3: set mid = (beg + end)/2
5.  Step 4: if a[mid] = val
6.  set pos = mid
7.  print pos
8.  go to step 6
9.  else if a[mid] > val
10. set end = mid - 1
```

11. else

```
12. set beg = mid + 1
13. [end of if]
14. [end of loop]
15. Step 5: if pos = -1
16. print "value is not present in the array"
17. [end of if]
18. Step 6: exit
```

```java
package binarysearch;

 import java.util.Scanner;
public class BinarySearch {
  public static void main(String args[])
  {
    int counter, num, item, array[], beg, end, middle;
    //To capture user input
    Scanner input = new Scanner(System.in);
    System.out.println("Enter number of elements:");
    num = input.nextInt();

    //Creating array to store the all the numbers
    array = new int[num];

    System.out.println("Enter " + num + " integers");
    //Loop to store each numbers in array
    for (counter = 0; counter < num; counter++)
        array[counter] = input.nextInt();

    System.out.println("Enter the search value:");
    item = input.nextInt();
    beg= 0;
   end = num - 1;
    middle = (beg + end)/2;

    while( beg <= end )
    {
      if ( array[middle] < item )
        beg= middle + 1;
      else if ( array[middle] == item )
```

```
        {
System.out.println(item+"found at location "+(middle + 1)+ ".");
          break;
        }
        else
        {
           end= middle - 1;
        }
        middle = (beg+ end)/2;
      }
      if (beg> end)
         System.out.println(item + " is not found.\n");
   }
}
```



Output - BinarySearch (run)

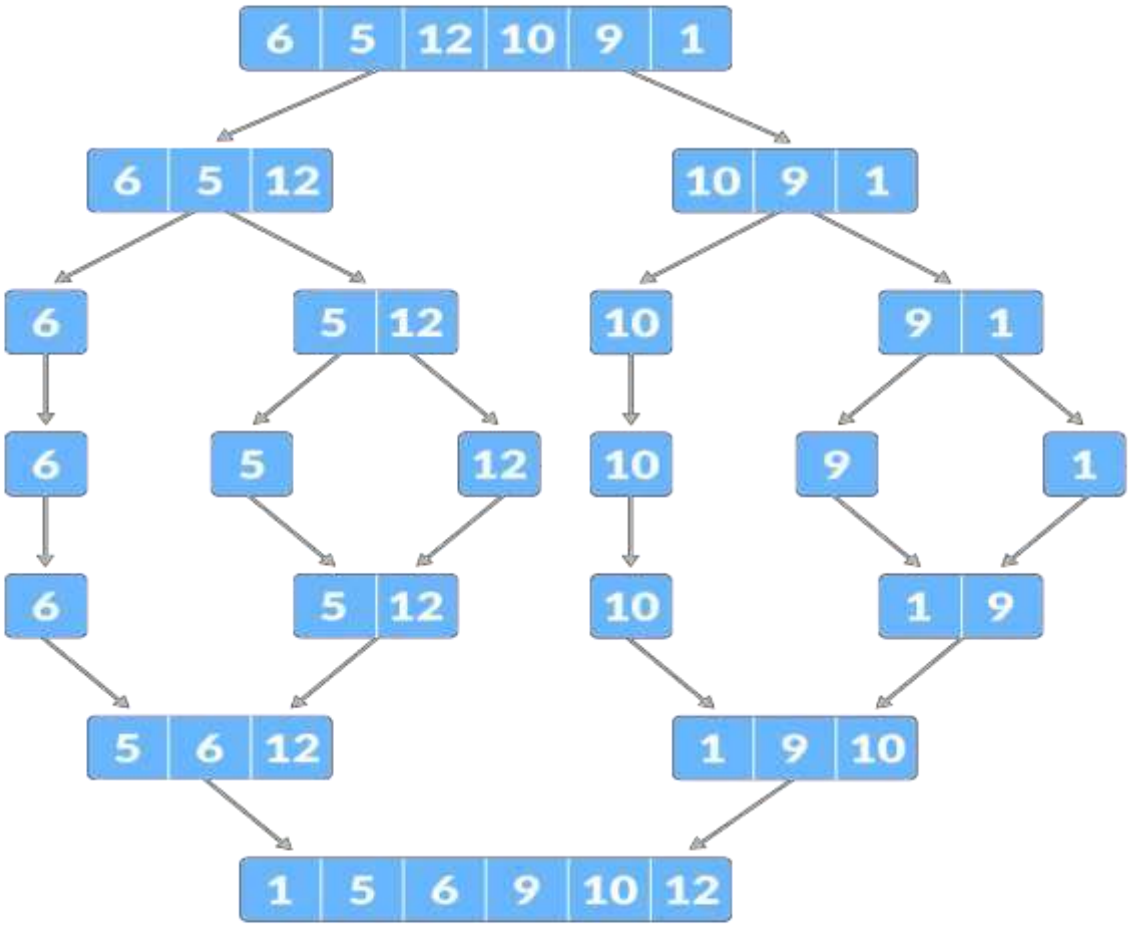
```
run:
Enter number of elements:
10
Enter 10 integers
2
4
7
9
13
45
67
78
89
90
Enter the search value:
78
78 found at location 8.
BUILD SUCCESSFUL (total time: 23 seconds)
```

PROGRAM: 7

Write a Java program to implement Merge sort algorithm for sorting a list of integers in ascending order.

Merge Sort is an efficient O(nlog n) sorting algorithm and It uses the divide-and-conquer approach. The algorithm works as follows:

- **Divide:** Divide the n elements sequence into two equal size subsequences of n/2 element each
- **Conquer:** Sort the two sub-sequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce a single sorted sequence.



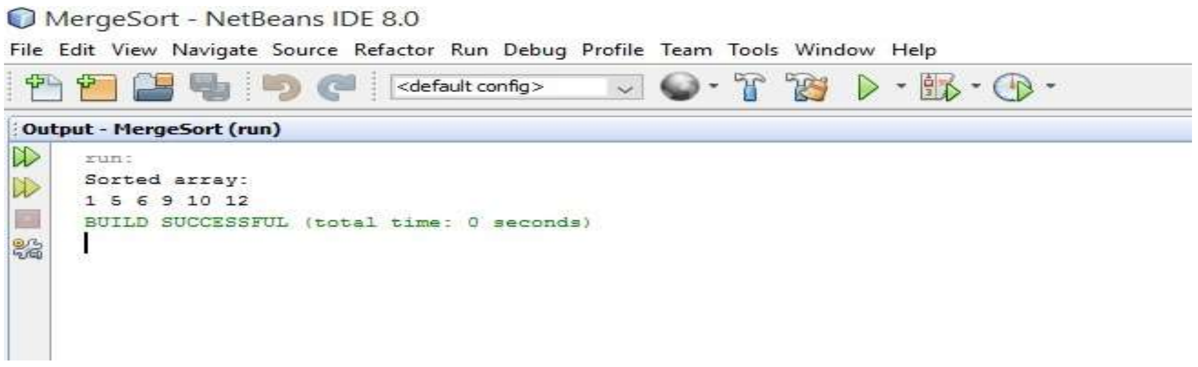
MergeSort Algorithm

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.
After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

```
MergeSort(A, p, r):

    if p > r

        return

    q = (p+r)/2

    mergeSort(A, p, q)

    mergeSort(A, q+1, r)

    merge(A, p, q, r)
```

```
package mergesort;

public class MergeSort {
  // Merge two subarrays L and M into arr
  void merge(int arr[], int p, int q, int r) {

    // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[] = new int[n1];
    int M[] = new int[n2];

    for (int i = 0; i < n1; i++)
      L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
      M[j] = arr[q + 1 + j];

    // Maintain current index of sub-arrays and main array
    int i, j, k;
    i = 0; j
    = 0; k
    = p;

    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[p..r]
    while (i < n1 && j < n2) {
      if (L[i] <= M[j]) {
        arr[k] = L[i];
        i++;
      } else {
        arr[k] = M[j];
        j++;
      }
      k++;
    }

    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
    }

    while (j < n2) {
      arr[k] = M[j];
      j++;
      k++;
    }
  }

  // Divide the array into two subarrays, sort them and merge them
  void mergeSort(int arr[], int l, int r) {
    if (l < r) {

      // m is the point where the array is divided into two subarrays
```

```
      int m = (l + r) / 2;

      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);

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

  // Print the array
  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[] = { 6, 5, 12, 10, 9, 1 };

    MergeSort ob = new MergeSort();
    ob.mergeSort(arr, 0, arr.length - 1);

    System.out.println("Sorted array:");
    printArray(arr);
  }
}
```


MergeSort - NetBeans IDE 8.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Output - MergeSort (run)

```
run:
Sorted array:
1 5 6 9 10 12
BUILD SUCCESSFUL (total time: 0 seconds)
```

PROGRAM: 8

Write a java program to implement Quick sort algorithm for sorting a list of integers in ascending order.

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into sub problems, then solving the sub problems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

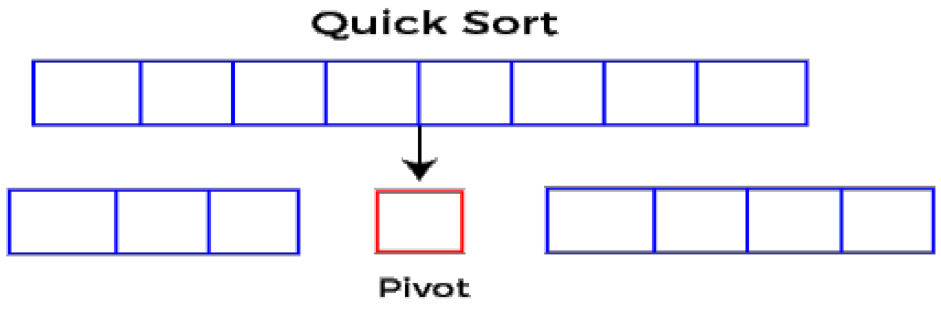
Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

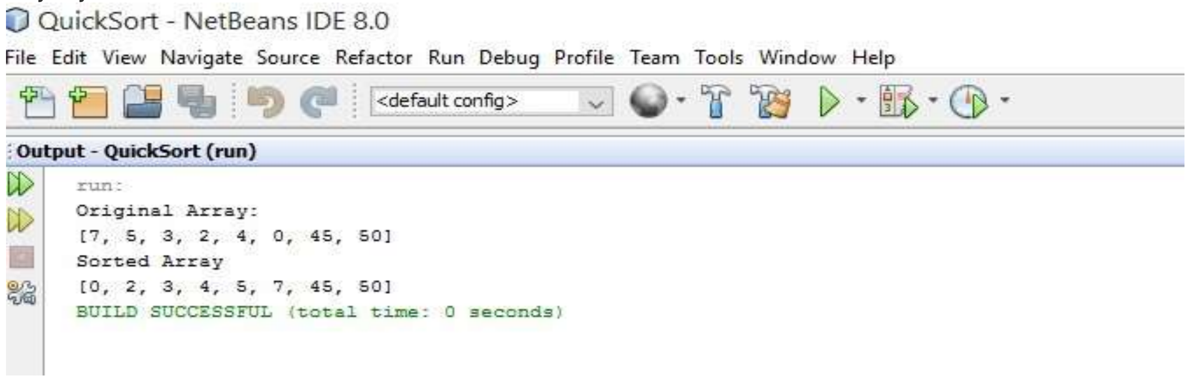
- ❖ Pivot can be random, i.e. select the random pivot from the given array.
- ❖ Pivot can either be the rightmost element of the leftmost element of the given array.
- ❖ Select median as the pivot element.



Algorithm:

```
QUICKSORT (array A, start, end)
{
 if (start < end)
 {
p = partition(A, start, end)
QUICKSORT (A, start, p - 1)
QUICKSORT (A, p + 1, end)
}
}
```

```
package quicksort;

import java.util.Arrays;
public class QuickSort {

    private int temp_array[];
    private int len;

    public void sort(int[] nums) {

        if (nums == null || nums.length == 0) {
            return;
        }
        this.temp_array = nums;
        len = nums.length;
        quickSort(0, len - 1);
    }
     private void quickSort(int low_index, int high_index) {

        int i = low_index;
        int j = high_index;
        // calculate pivot number
     int pivot = temp_array[low_index+(high_index-low_index)/2];
        // Divide into two arrays
        while (i <= j) {
             while (temp_array[i] < pivot) {
                i++;
            }
            while (temp_array[j] > pivot) {
                j--;
            }
            if (i <= j) {
                exchangeNumbers(i, j);
                //move index to next position on both sides
                i++;
                j--;
            }
        }
        // call quickSort() method recursively
        if (low_index < j)
            quickSort(low_index, j);
        if (i < high_index)
            quickSort(i, high_index);
    }
    private void exchangeNumbers(int i, int j) {
        int temp = temp_array[i];
        temp_array[i] = temp_array[j];
        temp_array[j] = temp;
    }
     public static void main(String args[])
    {
        QuickSort ob = new QuickSort();
        int nums[] = {7, 5, 3, 2, 4, 0, 45,50};
        System.out.println("Original Array:");
        System.out.println(Arrays.toString(nums));
        ob.sort(nums);
        System.out.println("Sorted Array");
        System.out.println(Arrays.toString(nums));
    }   }
```


QuickSort - NetBeans IDE 8.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Output - QuickSort (run)

```
run:
Original Array:
[7, 5, 3, 2, 4, 0, 45, 50]
Sorted Array
[0, 2, 3, 4, 5, 7, 45, 50]
BUILD SUCCESSFUL (total time: 0 seconds)
```

PROGRAM: 9

Write a program in Java to implement Horner Rule for Polynomial Evaluation.

Horner Rule:

Given a polynomial of the form $c_nx^n + c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \dots + c_1x + c_0$ and a value of x, find the value of polynomial for a given value of x. Here c_n, c_{n-1},..are integers (may be negative) and n is a positive integer.
Input is in the form of an array say *poly[ ]* where poly[0] represents coefficient for x^n and poly[1] represents coefficient for x^{n-1} and so on.

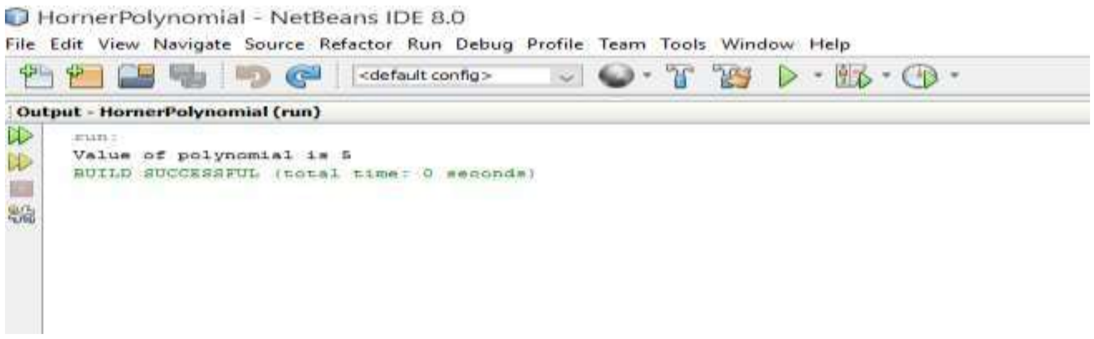
// Evaluate value of $2x^3 - 6x^2 + 2x - 1$ for x = 3
Input: poly[] = {2, -6, 2, -1}, x = 3
Output: 5

// Evaluate value of $2x^3 + 3x + 1$ for x = 2
Input: poly[] = {2, 0, 3, 1}, x = 2
Output: 23

A simple way to evaluate a polynomial is to one by one evaluate all terms. First calculate x^n, multiply the value with c_n, repeat the same steps for other terms and return the sum. Time complexity of this approach is **O(n²)** if we use a simple loop for evaluation of x^n. Time complexity can be improved to **O(nlogn)** if we use O(logn) approach for evaluation of x^n. Horner's Rule can be used to evaluate polynomial in **O(n) time**. To understand the method, let us consider the example of $2x^3 – 6x^2 + 2x – 1$. The polynomial can be evaluated as $((2x – 6)x + 2)x – 1$. The idea is to initialize result as coefficient of x^n which is 2 in this case, repeatedly multiply result with x and add next coefficient to result.

```
package hornerpolynomial;
import java.io.*;

class HornerPolynomial
{
// Function that returns value of poly[0]x(n-1) + poly[1]x(n-2) + .. + poly[n-1]
        static int horner(int poly[], int n, int x)
        {
                // Initialize result
                int result = poly[0];

        // Evaluate value of polynomial using Horner's method
```

```
                    for (int i=1; i<n; i++)
                              result = result*x + poly[i];

                    return result;
          }

          // Driver program
          public static void main (String[] args)
          {
          // Let us evaluate value of 2x3 - 6x2 + 2x - 1 for x = 3
                    int[] poly = {2, -6, 2, -1};
                    int x = 3;
                    int n = poly.length;
System.out.println(" Value of polynomial is "+ horner(poly,n,x));
          }
}
```


HornerPolynomial - NetBeans IDE 8.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Output - HornerPolynomial (run)

```
run:
Value of polynomial is 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

PROGRAM: 10

Write a Program in Java to implement Boyer Moore string matching Algorithm

The Boyer-Moore Algorithm

Robert Boyer and J Strother Moore established it in 1977. The B-M String search algorithm is a particularly efficient algorithm and has served as a standard benchmark for string search algorithm ever since.

The B-M algorithm takes a 'backward' approach: the pattern string (P) is aligned with the start of the text string (T), and then compares the characters of a pattern from right to left, beginning with rightmost character.

If a character is compared that is not within the pattern, no match can be found by analyzing any further aspects at this position so the pattern can be changed entirely past the mismatching character.

For deciding the possible shifts, B-M algorithm uses two preprocessing strategies simultaneously. Whenever a mismatch occurs, the algorithm calculates a variation using both approaches and selects the more significant shift thus, if make use of the most effective strategy for each case.

The two strategies are called heuristics of B - M as they are used to reduce the search. They are:

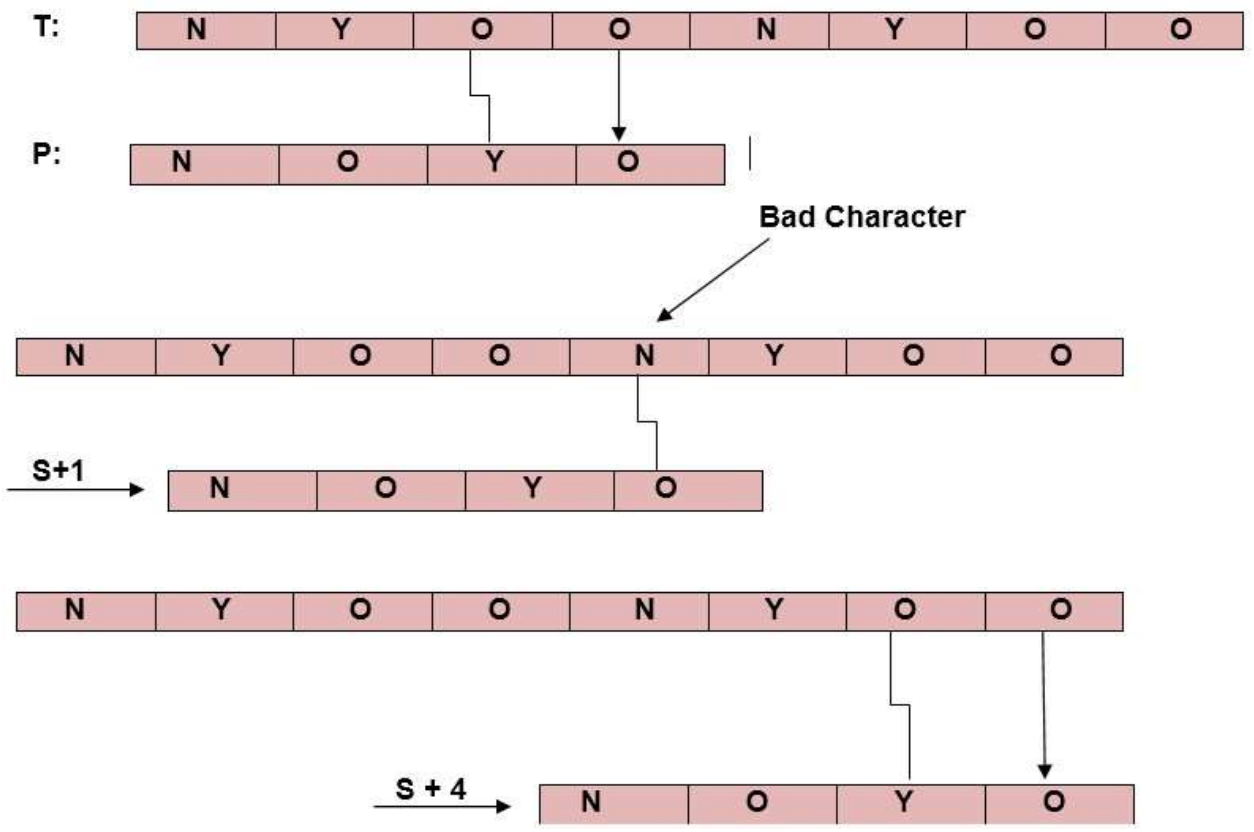
1. Bad Character Heuristics
2. Good Suffix Heuristics

1. Bad Character Heuristics

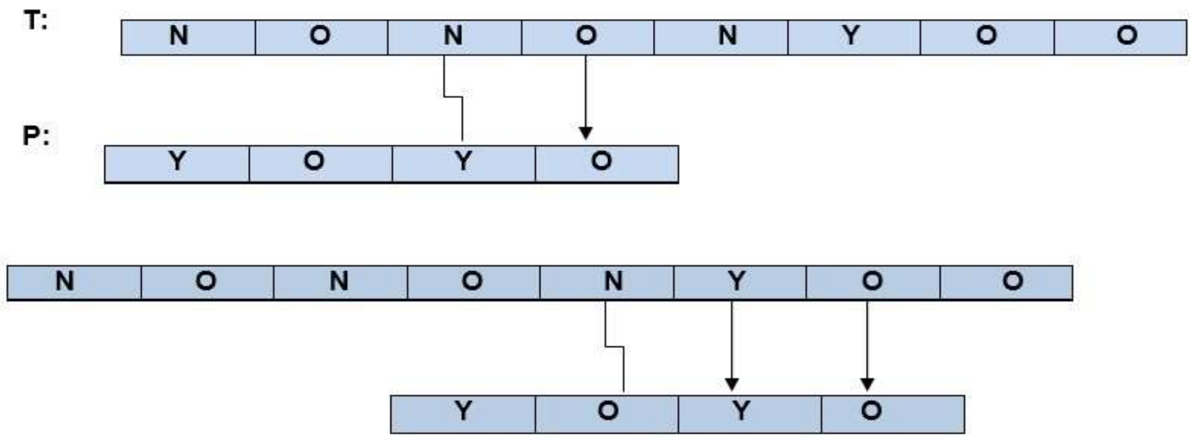
This Heuristics has two implications:

- Suppose there is a character in a text in which does not occur in a pattern at all. When a mismatch happens at this character (called as bad character), the whole pattern can be changed, begin matching form substring next to this 'bad character.'
- On the other hand, it might be that a bad character is present in the pattern, in this case, align the nature of the pattern with a bad character in the text. Thus in any case shift may be higher than one.

Example1: Let Text T = <nyoo nyoo> and pattern P = <noyo>



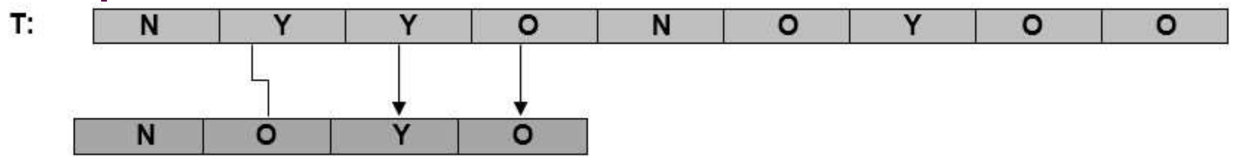
Example2: If a bad character doesn't exist the pattern then.

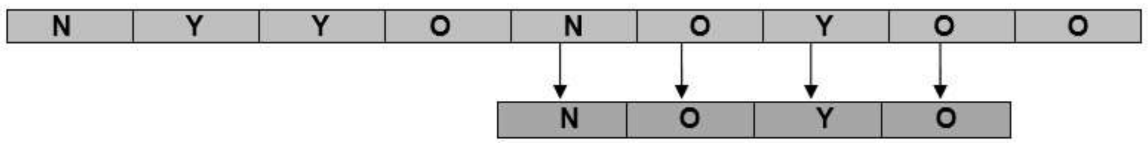


2. Good Suffix Heuristics:

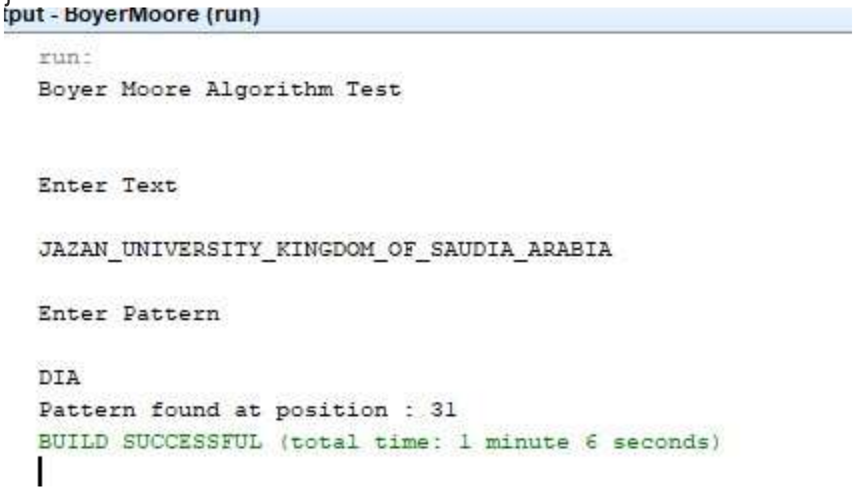
A good suffix is a suffix that has matched successfully. After a mismatch which has a negative shift in bad character heuristics, look if a substring of pattern matched till bad character has a good suffix in it, if it is so then we have an onward jump equal to the length of suffix found.

Example:





```
package boyermoore;

//Java Program to implement Boyer Moore Algorithm

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

/** Class BoyerMoore **/
public class BoyerMoore
{
   /** function findPattern **/
   public void findPattern(String t, String p)
   {
      char[] text = t.toCharArray();
      char[] pattern = p.toCharArray();
      int pos = indexOf(text, pattern);
      if (pos == -1)
         System.out.println("\nNo Match\n");
      else
     System.out.println("Pattern found at position : "+ pos);
   }
   /** Function to calculate index of pattern substring **/
   public int indexOf(char[] text, char[] pattern)
   {
      if (pattern.length == 0)
         return 0;
      int charTable[] = makeCharTable(pattern); int
      offsetTable[] = makeOffsetTable(pattern); for
      (int i = pattern.length - 1, j; i < text.length;)
      {
         for (j = pattern.length - 1; pattern[j] == text[i]; --i, --j)
                if (j == 0)
               return i;

    i += Math.max(offsetTable[pattern.length - 1 - j], charTable[text[i]]);
      }
      return -1;
    }
    /** Makes the jump table based on the mismatched character information **/
    private int[] makeCharTable(char[] pattern)
    {
      final int ALPHABET_SIZE = 256;
      int[] table = new int[ALPHABET_SIZE];
      for (int i = 0; i < table.length; ++i)
            table[i] = pattern.length;
      for (int i = 0; i < pattern.length - 1; ++i)
      table[pattern[i]] = pattern.length - 1 - i;
      return table;
    }
 /* Makes the jump table based on the scan offset which mismatch occurs. */
    private static int[] makeOffsetTable(char[] pattern)
    {
      int[] table = new int[pattern.length]; int
      lastPrefixPosition = pattern.length; for
      (int i = pattern.length - 1; i >= 0; --i)
      {
         if (isPrefix(pattern, i + 1))
               lastPrefixPosition = i + 1;
table[pattern.length - 1 - i] = lastPrefixPosition - i + pattern.length - 1;
      }
      for (int i = 0; i < pattern.length - 1; ++i)
      {
           int slen = suffixLength(pattern, i);
           table[slen] = pattern.length - 1 - i + slen;
      }
      return table;
   }
    /** function to check if needle[p:end] a prefix of pattern **/
    private static boolean isPrefix(char[] pattern, int p)
    {
        for (int i = p, j = 0; i < pattern.length; ++i, ++j)
            if (pattern[i] != pattern[j])
                return false;
        return true;
    }
    /** function to returns the maximum length of the sub string ends at p and is a suffix **/
    private static int suffixLength(char[] pattern, int p)
    {
        int len = 0;
for (int i = p,j = pattern.length - 1;i>= 0&&pattern[i] == pattern[j];--i,--j)
            len += 1;
        return len;
    }
    /** Main Function **/
    public static void main(String[] args) throws IOException
    {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Boyer Moore Algorithm Test\n");
        System.out.println("\nEnter Text\n");
        String text = br.readLine();
        System.out.println("\nEnter Pattern\n");
        String pattern = br.readLine();
        BoyerMoore bm = new BoyerMoore();
        bm.findPattern(text, pattern);
    }
}
```



tput - BoyerMoore (run)

```
run:
Boyer Moore Algorithm Test


Enter Text

JAZAN_UNIVERSITY_KINGDOM_OF_SAUDIA_ARABIA

Enter Pattern

DIA
Pattern found at position : 31
BUILD SUCCESSFUL (total time: 1 minute 6 seconds)
```

PROGRAM: 11

Write a java program for Warshall's Algorithm to find the transitive closure of a Graph.

Warshall's Algorithm

Warshall's algorithm is used to determine the transitive closure of a directed graph or all paths in a directed graph by using the adjacency matrix. For this, it generates a sequence of n matrices. Where, n is used to describe the number of vertices.

$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}$

A sequence of vertices is used to define a path in a simple graph. In the kth matrix ($R^{(k)}$), ($r_{ij}^{(k)}$), the element's definition at the ith row and jth column will be one if it contains a path from v_i to v_j. For all intermediate vertices, w_q is among the first k vertices that mean $1 \le q \le k$.

The $R^{(0)}$ matrix is used to describe the path without any intermediate vertices. So we can say that it is an adjacency matrix. The $R^{(n)}$ matrix will contain ones if it contains a path between vertices with intermediate vertices from any of the n vertices of a graph. So we can say that it is a transitive closure.

Now we will assume a case in which $r_{ij}^{(k)}$ is 1 and $r_{ij}^{(k-1)}$ is 0. This condition will be true only if it contains a path from v_i to v_j using the v_k. More specifically, the list of vertices is in the following form

v_i, w_q (where $1 \le q < k$), v_k. w_q (where $1 \le q < k$), v_j

The above case will be occur only if $r_{ik}^{(k-1)} = r_{kj}^{(k-1)} = 1$. Here, k is subscript.

The $r_{ij}^{(k)}$ will be one if and only if $r_{ij}^{(k-1)} = 1$.

So in summary, we can say that

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } (r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)})$$

Now we will describe the algorithm of Warshall's Algorithm for computing transitive closure

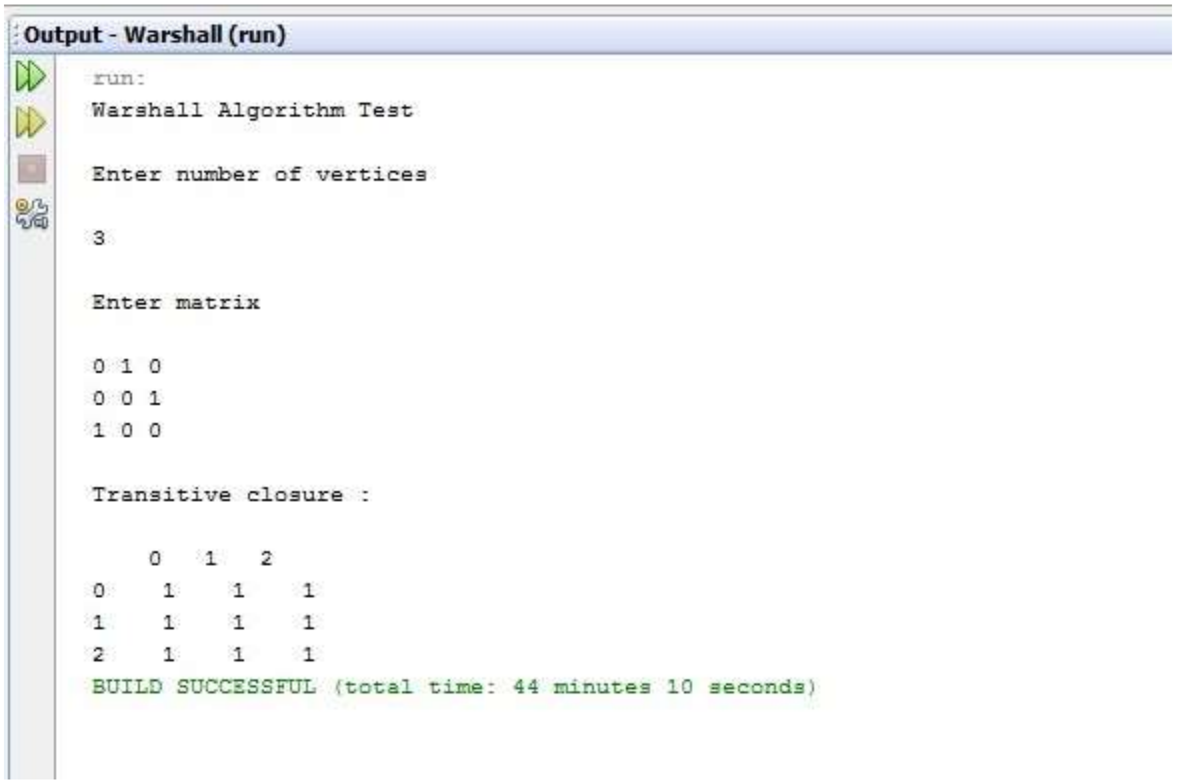
```
Warshall (A[1...n, 1...n]) // A is the adjacency matrix
R(0) ← A
for k ← 1 to n do
for i ← 1 to n do
for j ← to n do
R(k)[i, j] ← R(k-1)[i, j] or (R(k-1)[i, k] and R(k-1)[k, j])
return R(n)
```

Here,

- Time efficiency of this algorithm is (n3)
- In the Space efficiency of this algorithm, the matrices can be written over their predecessors.
- Θ(n^3) is the worst-case cost. We should know that the brute force algorithm is better than Warshall's algorithm. In fact, the brute force algorithm is also faster for a space graph.

```
package warshall;

import java.util.Scanner;
public class Warshall {

    private int V;
    private boolean[][] tc;
    /** Function to make the transitive closure **/
    public void getTC(int[][] graph)
    {
        this.V = graph.length;
        tc = new boolean[V][V];
        for (int i = 0; i < V; i++)
        {
            for (int j = 0; j < V; j++)
            if (graph[i][j] != 0) tc[i][j]
                     = true;
            tc[i][i] = true;
        }
        for (int i = 0; i < V; i++)
        {
            for (int j = 0; j < V; j++)
            {
                if (tc[j][i])
                    for (int k = 0; k < V; k++)
                        if (tc[j][i] && tc[i][k])
                            tc[j][k] = true;
            }
        }
    }
    /** Funtion to display the trasitive closure **/
    public void displayTC()
    {
        System.out.println("\nTransitive closure :\n");
        System.out.print(" ");
        for (int v = 0; v < V; v++)
           System.out.print("   " + v );
        System.out.println();
        for (int v = 0; v < V; v++)
        {
            System.out.print(v +" ");
            for (int w = 0; w < V; w++)
            {
                if (tc[v][w])
                    System.out.print("   1 ");
                else
                    System.out.print("   0 ");
            }
            System.out.println();
        }
    }

    /** Main function **/
    public static void main (String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Warshall Algorithm Test\n");
        /** Make an object of Warshall class **/
        Warshall w = new Warshall();

        /** Accept number of vertices **/
        System.out.println("Enter number of vertices\n");
        int V = scan.nextInt();

        /** get graph **/
```

```
        System.out.println("\nEnter matrix\n");
        int[][] graph = new int[V][V];
        for (int i = 0; i < V; i++)
            for (int j = 0; j < V; j++)
                graph[i][j] = scan.nextInt();

        w.getTC(graph);
        w.displayTC();
    }
}
```



Output - Warshall (run)

```
run:
Warshall Algorithm Test

Enter number of vertices

3

Enter matrix

0 1 0
0 0 1
1 0 0

Transitive closure :

    0   1   2
0    1    1    1
1    1    1    1
2    1    1    1
BUILD SUCCESSFUL (total time: 44 minutes 10 seconds)
```

PROGRAM: 12

Write a Java Program to find the all pair shortest path using Floyd Algorithm.

Floyd’s Algorithm for the All-Pairs Shortest-Paths Problem

Given a weighted connected graph (undirected or directed), the ***all-pairs shortest paths Problem*** asks to find the distances—i.e., the lengths of the shortest paths from each vertex to all other vertices. It is convenient to record the lengths of shortest paths in an n × n matrix D called the ***distance matrix***: the element d in the i th row and the j th column of this matrix indicates the length of the shortest path from the i th vertex to the j th vertex. Floyd’s algorithm computes the distance matrix of a weighted graph with n vertices through a series of n × n matrices:

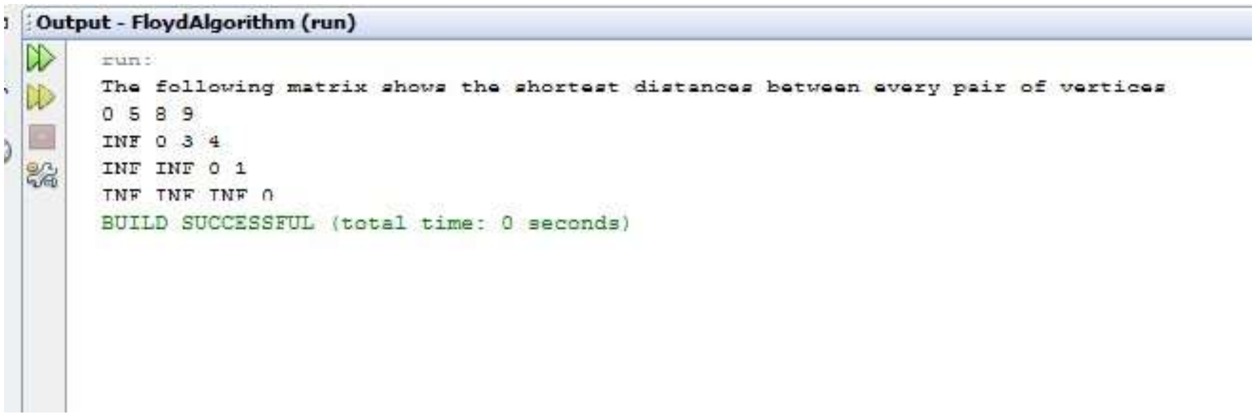
$$D^{(0)}, \dots, D^{(k-1)}, D^{(k)}, \dots, D^{(n)}.$$

```
ALGORITHM   Floyd(W[1..n, 1..n])
    //Implements Floyd’s algorithm for the all-pairs shortest-paths problem
    //Input: The weight matrix W of a graph with no negative-length cycle
    //Output: The distance matrix of the shortest paths’ lengths
    D ← W  //is not necessary if W can be overwritten
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
    return D
```

```
package floydalgorithm;

import java.util.*;
import java.lang.*;
import java.io.*;
public class FloydAlgorithm {
final static int INF = 99999, V = 4;

        void Floyd(int graph[][])
        {
                int dist[][] = new int[V][V];
                int i, j, k;
```

```
                    /* Initialize the solution matrix
                    Same as input graph matrix.
                    Or we can say the initial values
                    of shortest distances
                    are based on shortest paths
                    Considering no intermediate
                    Vertex. */
                    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];
                                    }
                            }
                    }

                    // Print the shortest distance matrix
                    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)
                {
                        int graph[][] = { {0, 5, INF, 10},
                                                        {INF, 0, 3, INF},
                                                        {INF, INF, 0, 1},
                                                        {INF, INF, INF, 0}
                                                        };
                        FloydAlgorithm a = new FloydAlgorithm();

                        // Print the solution
                        a.Floyd(graph);
                }
}
```



Output - FloydAlgorithm (run)

```
run:
The 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
BUILD SUCCESSFUL (total time: 0 seconds)
```

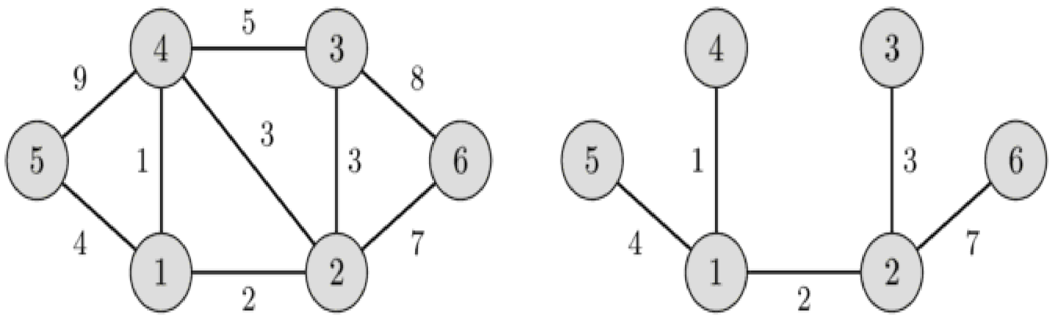
PROGRAM: 13

Write a Java program that implements Prim's algorithm to generate minimum cost spanning tree.

Minimum spanning tree - Prim's algorithm

Given a weighted, undirected graph G with n vertices and m edges. You want to find a spanning tree of this graph which connects all vertices and has the least weight (i.e. the sum of weights of edges is minimal). A spanning tree is a set of edges such that any vertex can reach any other by exactly one simple path. The spanning tree with the least weight is called a minimum spanning tree.

In the left image you can see a weighted undirected graph, and in the right image you can see the corresponding minimum spanning tree.

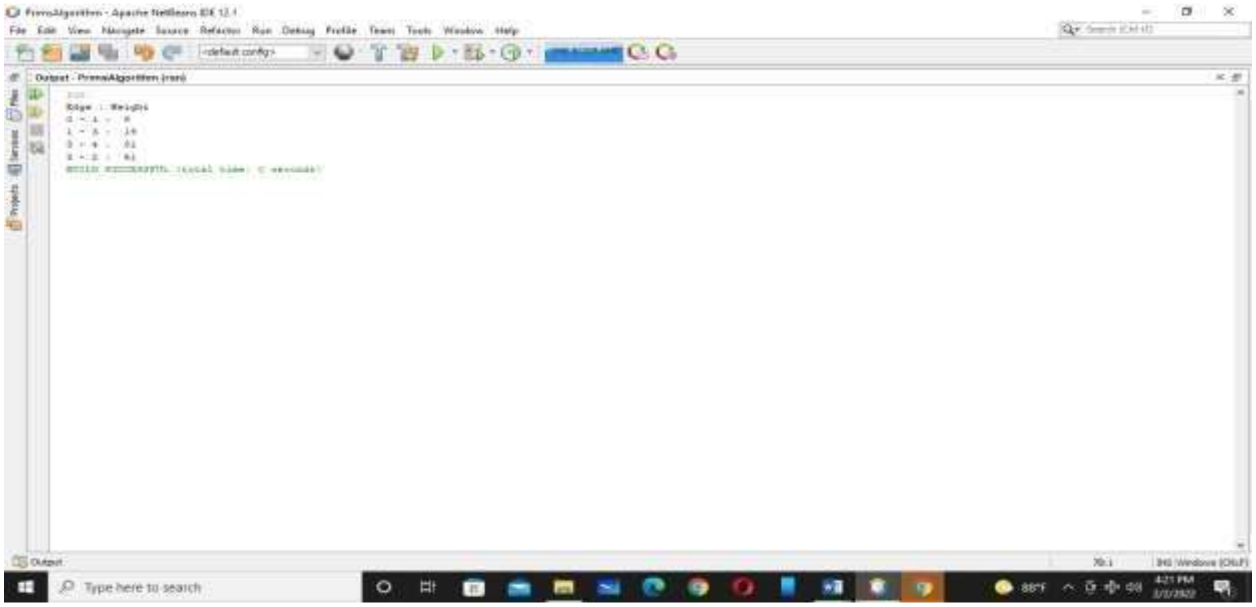


It is easy to see that any spanning tree will necessarily contain n−1 edges.

This problem appears quite naturally in a lot of problems. For instance in the following problem: there are n cities and for each pair of cities we are given the cost to build a road between them (or we know that is physically impossible to build a road between them). We have to build roads, such that we can get from each city to every other city, and the cost for building all roads is minimal.

```
ALGORITHM  Prim(G)
    //Prim's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    $V_T \leftarrow \{v_0\}$    //the set of tree vertices can be initialized with any vertex
    $E_T \leftarrow \varnothing$
    for $i \leftarrow 1$ to $|V| - 1$ do
        find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
        such that $v$ is in $V_T$ and $u$ is in $V - V_T$
        $V_T \leftarrow V_T \cup \{u^*\}$
        $E_T \leftarrow E_T \cup \{e^*\}$
    return $E_T$
```

```java
package primsalgorithm;
import java.util.Arrays;
public class PrimsAlgorithm {


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

    int INF = 9999999;

    int no_edge; // number of edge

    // create a array to track selected vertex
    // selected will become true otherwise false
    boolean[] selected = new boolean[V];

    // set selected false initially
    Arrays.fill(selected, false);

    // set number of edge to 0
    no_edge = 0;

    // the number of egde in minimum spanning tree will be
    // always less than (V -1), where V is number of vertices in
    // graph

    // choose 0th vertex and make it true
    selected[0] = true;
  // print for edge and weight
  System.out.println("Edge : Weight");

  while (no_edge < V - 1) {
    // For every vertex in the set S, find the all adjacent vertices
    // , calculate the distance from the vertex selected at step 1.
    // if the vertex is already in the set S, discard it otherwise
    // choose another vertex nearest to selected vertex at step 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) {
 PrimsAlgorithm g = new PrimsAlgorithm();

  // number of vertices in graph
  int V = 5;
int[][] G = { { 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 } };

  g.Prim(G, V);
  }
}
```



```
run:
Edge : Weight
0 - 1 : 8
1 - 3 : 16
3 - 4 : 51
4 - 2 : 61
BUILD SUCCESSFUL (total time: 0 seconds)
```

PROGRAM: 14

Write a Java program that implements Krushal’s algorithm to generate minimum cost spanning tree.

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

How does Kruskal's algorithm work?

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- ❖ First, sort all the edges from low weight to high.
- ❖ Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- ❖ Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

The applications of Kruskal's algorithm are -

- ❖ Kruskal's algorithm can be used to layout electrical wiring among cities.
- ❖ It can be used to lay down LAN connections.

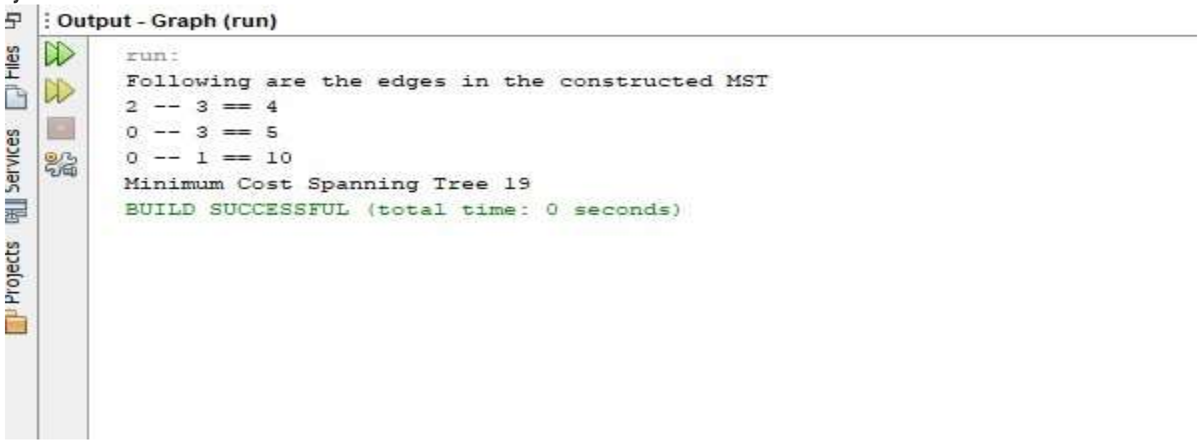
```
ALGORITHM  Kruskal(G)
    //Kruskal's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of G
    sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
    $E_T \leftarrow \varnothing$;  $ecounter \leftarrow 0$     //initialize the set of tree edges and its size
    $k \leftarrow 0$                          //initialize the number of processed edges
    while $ecounter < |V| - 1$ do
        $k \leftarrow k + 1$
        if $E_T \cup \{e_{i_k}\}$ is acyclic
            $E_T \leftarrow E_T \cup \{e_{i_k}\}$;  $ecounter \leftarrow ecounter + 1$
    return $E_T$
```

```
package graph;
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph {
// A class to represent a graph edge
class Edge implements Comparable<Edge>
{
 int src, dest, weight;
// Comparator function used for
// sorting edgesbased on their weight
public int compareTo(Edge compareEdge)
{
return this.weight - compareEdge.weight;
}
};
// A class to represent a subset for
// union-find
class subset
{
int parent, rank;
};
int V, E; // V-> no. of vertices & E->no.of edges
Edge edge[]; // collection of all edges
// Creates a graph with V vertices and E edges
Graph(int v, int e)
{
V = v;
E = e;
edge = new Edge[E];
for (int i = 0; i < e; ++i)
edge[i] = new Edge();
}
// A utility function to find set of an
// element i (uses path compression technique)
int find(subset subsets[], int i)
{
// find root and make root as parent of i
// (path compression)
   if (subsets[i].parent != i)
subsets[i].parent= find(subsets, subsets[i].parent);
```

```
return subsets[i].parent;
}
// A function that does union of two sets
// of x and y (uses union by rank)
void Union(subset subsets[], int x, int y)
{
int xroot = find(subsets, x);
int yroot = find(subsets, y);
// Attach smaller rank tree under root
// of high rank tree (Union by Rank)
if (subsets[xroot].rank < subsets[yroot].rank)
subsets[xroot].parent = yroot;
else if (subsets[xroot].rank > subsets[yroot].rank)
subsets[yroot].parent = xroot;
// If ranks are same, then make one as
// root and increment its rank by one
else {
subsets[yroot].parent = xroot;
subsets[xroot].rank++;
}
}

// The main function to construct MST using Kruskal's
void KruskalMST()
{
// Tnis will store the resultant MST
Edge result[] = new Edge[V];
// An index variable, used for result[]
int e = 0;
// An index variable, used for sorted edges
int i = 0;
for (i = 0; i < V; ++i)
result[i] = new Edge();
// Step 1:  Sort all the edges in non-decreasing
// order of their weight.  If we are not allowed to
// change the given graph, we can create a copy of
// array of edges
Arrays.sort(edge);
// Allocate memory for creating V subsets
subset subsets[] = new subset[V];
for (i = 0; i < V; ++i)
subsets[i] = new subset();
// Create V subsets with single elements
```

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

subsets[v].parent = v;
subsets[v].rank = 0;
}
i = 0; // Index used to pick next edge
// Number of edges to be taken is equal to V-1
while (e < V - 1)
{
// Step 2: Pick the smallest edge. And increment
// the index for next iteration
Edge next_edge = new Edge();
next_edge = edge[i++];
int x = find(subsets, next_edge.src);
int y = find(subsets, next_edge.dest);
// If including this edge does't cause cycle,
// include it in result and increment the index
// of result for next edge
if (x != y) {
result[e++] = next_edge;
Union(subsets, x, y);
}
// Else discard the next_edge
}
// print the contents of result[] to display
// the built MST
System.out.println("Following are the edges in "+"the constructed MST");
int minimumCost = 0;
for (i = 0; i < e; ++i)
{
System.out.println(result[i].src+" - "+ result[i].dest + "== "+result[i].weight);
 minimumCost += result[i].weight;
 }
System.out.println("Minimum Cost Spanning Tree "+ minimumCost);
}
// Driver Code
public static void main(String[] args)
{
int V = 4; // Number of vertices in graph
int E = 5; // Number of edges in graph
Graph graph = new Graph(V, E);
// add edge 0-1
```

```
graph.edge[0].src = 0;
graph.edge[0].dest = 1;
graph.edge[0].weight = 10;
// add edge 0-2
graph.edge[1].src = 0;
graph.edge[1].dest = 2;
graph.edge[1].weight = 6;
// add edge 0-3
graph.edge[2].src = 0;
graph.edge[2].dest = 3;
graph.edge[2].weight = 5;
// add edge 1-3
graph.edge[3].src = 1;
graph.edge[3].dest = 3;
graph.edge[3].weight = 15;
// add edge 2-3
graph.edge[4].src = 2;
graph.edge[4].dest = 3;
graph.edge[4].weight = 4;
// Function call
graph.KruskalMST();
}
 }
```



```
run:
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree 19
BUILD SUCCESSFUL (total time: 0 seconds)
```