import java.io.\*;

public class Recursion\_examples

{

 public static void main( String[] args )

 {

 boolean result;

 int [] a = {3,3,3,3,3};

 int [] b = {3,3,3,4,5,3};

 result = allEqual( a, a.length );

 System.out.println( "All the elements are equal: " + result );

 result = allEqual( b, b.length );

 System.out.println( "All the elements are equal: " + result );

 char [] c = {'a', 'b', 'd', 'e', 'f'};

 swap\_rec( c, 0, c.length - 1);

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

 {

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

 }

 System.out.println();

 int [] x = {1, 5, 6, 8, 20, 34, 55, 56, 67};

 result = searchRec( x, 54, 0, x.length - 1);

 System.out.println( "The element was found: " + result );

 }

 /\* Given an array A of more than N numbers, return TRUE

 \* if all the numbers in positions 0…N of A are equal, and false otherwise.

 \*/

 public static boolean allEqual (int [] a, int n)

 {

 boolean allEQ; // RESULT

 if (n <= 1)

 {

 allEQ = true;

 }

 else

 {

 if (a[n-1] == a[0])

 {

 allEQ = allEqual(a, n-1);

 }

 else

 {

 allEQ = false; // no need for the recursive call!

 }

 }

 return allEQ;

 }

 /\* Given a non-empty array A of N characters, reverse the values stored in

 \* positions Start to Finish

 \*/

 public static void swap\_rec (char [] a, int low, int high)

 {

 int newHigh; // INTERMEDIATE: "smaller" high

 int newLow; // INTERMEDIATE: "higher" low

 char temp; // INTERMEDIARTE

 if (high - low <= 1)

 {

 ; // base case: do nothing

 }

 else

 {

 newLow = low + 1;

 newHigh = high - 1; // shorten the part of the array that is considered

 swap\_rec( a, newLow, newHigh );

 }

 // swap a[low] and a[high]

 temp = a[low];

 a[low] = a[high];

 a[high] = temp;

 // RETURN RESULT : NONE the array is modified!

}

 /\*

 \* Binary search

Suppose that you have an array of integers that is already known to be sorted in ascending order, and

to contain no duplicate values. A binary search on such an array is done by locating the middle

element of an array and comparing the search value with that element. If we have not found the

value for which we are searching, we can choose the appropriate sub-interval (either the left or the

right side) to restrict the search.

Write a recursive method that will perform a binary search on an array a for the value findMe

between positions startIndex and endIndex. You can assume that both positions are less than

the length of the array. The method should return true if the value is contained in the array, and

false otherwise.

 \*/

 public static boolean searchRec( int[] valueList, int findMe, int leftIndex, int rightIndex )

 {

 // DECLARE VARIABLES / DATA DICTIONARY

 boolean found; // RESULT: True if search is successful, and false otherwise.

 int mid; // Index of array closest to the midpoint between leftIndex and rightIndex

 // BODY OF ALGORITHM

 // Check for base case.

 // The base case covers 2 situations: leftIndex and rightIndex are the

 // same, or they are two consecutive array positions. The latter case is

 // needed as there is no useful midpoint between two consecutive array

 // positions, and the possibility of not reducing the size of the interval.

 if ( leftIndex + 1 >= rightIndex )

 {

 // For the base case, if the value doesn't match one of the two possible

 // endpoints, the value is not in the array.

 found = findMe == valueList[leftIndex] || findMe == valueList[rightIndex];

 }

 else

 {

 // Determine array position closest to the midpoint between leftIndex and rightIndex.

 mid = ( leftIndex + rightIndex ) / 2;

 // Compare with value at midpoint.

 if ( findMe == valueList[mid] )

 { // We got lucky and found the value.

 found = true;

 }

 else

 { // Decide whether the value, if it were present, would be to the

 // left of the midpoint or to the right of the midpoint.

 if ( findMe < valueList[mid] )

 {

 // Value is on left side of midpoint. Search left half of array recursively.

 found = searchRec( valueList, findMe, leftIndex, mid );

 }

 else

 {

 // Value is on the right side of the midpoint. Search right half of array recursively.

 found = searchRec( valueList, findMe, mid, rightIndex );

 }

 }

 }

 return found; // RETURN RESULT

 }

}