**package** lesson9.year16\_17;

/\*\*

\* **@author** Programming Professors. Bachelor's Degree in Computer Science and Engineering. UC3M

\* **@since** December 2013

\* **@version** 1.2

\*

\* Implementation of three sorting algorithms:

- Bubble sort

- Insertion sort

- Selection sort

\* Code is organized using Structured Programming, so static methods are created

\* and then used in the main method.

\*/

**public** **class** Sort {

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* B U B B L E \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*\*

\* Easiest implementation of bubble sort

\*

\* **@param** list list to order (arrays are passed by reference)

\*/

**public** **static** **void** bubbleEasy(**int** [] list){

**int** aux = 0;

//outer loop: (elements-1) loops max

**for** (**int** i=1; i<list.length; i++){

//Uncomment these lines to print the trace

// System.out.println();

// System.out.println("Iteration number "+i);

////Inner loop (n-i comparisons)

**for** (**int** j=0; j<list.length-i; j++){

//if the lower index element is bigger than the next element, we swap

**if** (list[j]>list[j+1]){

//we use aux to swap

aux = list [j+1];

list [j+1] = list [j];

list [j] = aux;

}//end if

// Uncomment these lines to print the trace

// for (int k = 0; k<list.length; k++)

// System.out.print (list[k]+" ");

// System.out.println();I

}//end for j

}//end for i

}// end bubble

/\*\*

\* Bubble sort to sort integer numbers in ascending order

\* It is able to detect if the list is already ordered at any intermediate step,

\* so it finishes earlier

\*

\* **@param** list list to order (arrays are passed by reference)

\*/

**public** **static** **void** bubble (**int** [] list){

// defining auxiliary variables

**int** aux = 0;

**boolean** change = **true**;

//outer loop: (elements-1) loops max

**for** (**int** i=1; i<list.length && change; i++){

//Uncomment these lines to print the trace

// System.out.println();

// System.out.println("Iteration number "+i);

//At the beginning of each iteration change is false

change = **false**;

//Inner loop (n-i comparisons)

**for** (**int** j=0; j<list.length-i; j++){

// if the lower index element is bigger than the next element, we swap

**if** (list[j]>list[j+1]){

//we did a change, so we update the variable

change = **true**;

//we use aux to swap

aux = list [j+1];

list [j+1] = list [j];

list [j] = aux;

}//end if

// Uncomment these lines to print the trace

// for (int k = 0; k<list.length; k++)

// System.out.print (list[k]+" ");

// System.out.println();

}//end for j

}//end for i

}// end bubble

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* I N S E R T I O N \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*\*

\* Insertion sort algorithm implementation

\*

\* **@param** list list to order (arrays are passed by reference)

\*/

**public** **static** **void** insertionSort (**int** [] list){

// the first element is a sorted sublist, so we start in the second one

// at each iteration the i first elements will be already sorted

**for** (**int** i=1; i<list.length; i++){

//we store the value of the first unsorted element

**int** auxiliar = list[i];

//now we go backwards looking for its place in the sorted sublist

**int** j=i-1;

//we iterate while there are elements in the sorted sublist and the value of the element

//we want to insert into it is lower than the value of the current element (j-element)

**while** (j>=0 && auxiliar<list[j]){

//we shift elements

list [j+1]=list[j];

// decrementing j to walk through the list backwards

j--;

}// end while

//once found where the element should be placed we place it

list[j+1]=auxiliar;

// and we start with the next unsorted one

}// end for

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S E L E C T I O N \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*\*

\* Selection sort algorithm implementation

\*

\* **@param** list list to order (arrays are passed by reference)

\*/

**public** **static** **void** selectionSort (**int** [] list){

//defining auxiliary variables

**int** minValue, minPos;

//we iteratively search for the lowest element of the list

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

//in each iteration the first element is the "lowest" (to start comparisons)

minPos=i;

minValue=list[i];

//We use this loop to look for the lowest

**for** (**int** j=i+1; j<list.length; j++) {

//If the current element is lower than the minValue, its value is now minValue

**if** (list[j]<minValue) {

minValue=list[j];

//We mark also its position

minPos=j;

}// end if

}// end for (j)

//once the lowest is found, we swap it with the element at position i

list[minPos]=list[i];

list[i]=minValue;

//and we keep iterating with the outer loop

} //end for (i)

}

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

//declaring an array to sort

**int** [] a= **new** **int** []{1,5,4,3,2,6,0,0,11,3,32,7,8,9,10,-1,45};

//Remove comments to invoke any of the methods

//bubble(a);

//insertionSort(a);

//selectionSort(a);

**for** (**int** k=0; k<a.length;k++)

System.out.print(a[k]+",");

}

}

**package** lesson9.year16\_17;

/\*\*

\* **@author** Programming Professors. Bachelor's Degree in Computer Science and Engineering. UC3M

\* **@since** December 2013

\* **@version** 1.2

\*

\* Implementation of linear and binary search over an integer array.

\* It has a method for each search type.

\* Code is organized using Structured Programming, so static methods are created

\* and then used in the main method.

\*/

**public** **class** Search {

/\*\*

\* Linear search.

\*

\* **@param** list list of integers

\* **@param** element element to look for

\* **@return** position of the first appearance of the element or -1 if it is not in the list

\*/

**public** **static** **int** linearSearch (**int** [] list, **int** element){

**for** (**int** counter = 0; counter < list.length; counter++){

//if we found the element, we return the position

**if** (list[counter]==element) {

**return** counter;

}// end if

}// end while

//If not found, we return -1

**return** -1;

}

/\*\*

\* Linear search in part of the list (bounded linear search). It does not check that the

\* provided bounds are valid, so ArrayIndexOutOfBounds exceptions may occur.

\*

\* **@param** list list of integers

\* **@param** element element to look for

\* **@param** first index of the first element to start searching.

\* **@param** last index of the element to stop searching

\* **@return** position of the first appearance of the element or -1 if it is not in the sublist

\*/

**public** **static** **int** boundedLinearSearch (**int** [] list, **int** element, **int** first, **int** last){

**for** (**int** counter = first; counter <= last; counter++) {

**if** (list[counter]==element) {

**return** counter;

}// end if

}// end while

**return** -1;

}

/\*\*

\* Binary Search

\*

\* **@param** list list of integers

\* **@param** element element to look for

\* **@return** position of the element or -1 if it is not in the list

\*/

**public** **static** **int** binarySearch (**int** [] list, **int** element){

//auxiliary variables to define search interval

**int** biggest = list.length-1, lowest = 0, central= (biggest+lowest)/2;

// while we have an interval to search in

**while** (biggest>=lowest){

// to see the algorithm trace, uncomment this line

// System.out.println(biggest+ " "+lowest+" " +central);

// if our element is at central position, we found it

**if** (list[central] == element) **return** central;

// if not, if it is lower than central, we discard the upper half of the list

**else** **if** (list[central] > element) biggest = central-1;

// if not, it will we higher, so we discard the lower half of the list

**else** lowest = central+1;

//we compute again the center of the list

central = (biggest+lowest)/2;

}//end while

//if not found, return -1

**return** -1;

}

/\*\* Bounded binary search

\* **@param** list list of integers

\* **@param** element element to look for

\* **@param** first index of the first element to start searching

\* **@param** last index of the element to stop searching

\* **@return** position of the element or -1 if it is not in the sublist

\*

\* Not very efficient implementation, we are duplicating the array, but allows us to reuse previous method.

\* It does not check that the provided bounds are valid, so ArrayIndexOutOfBounds exceptions may occur.

\*/

**public** **static** **int** boundedBinarySearch (**int** [] list, **int** element, **int** first, **int** last){

//creating a new array with the sublist

**int** [] newarr = **new** **int** [last-first+1];

System.arraycopy(list, first, newarr, 0, last-first+1);

//we invoke binarySearch method with this array

**int** position = binarySearch(newarr,element);

//we cannot return position because it is the position on the sublist, not in the original one

//except if we didn't find it

**if** (position==-1)

**return** position;

**else**

**return** position+first;

}

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

//Examples of use

**int** [] unsorted = **new** **int** [] {1,12,7,9,3,2,4,8,6};

**int** [] sorted = **new** **int** [] {1,3,4,6,7,8,10,12};

//Searching for the 3 (found in both)

System.out.println(linearSearch(unsorted, 3));

System.out.println(binarySearch(sorted, 3));

//Searching for the 5 (not present)

System.out.println(linearSearch(unsorted, 5));

System.out.println(binarySearch(sorted, 5));

//Searching for the 5 in the 4 first elements of the sorted list

System.out.println(boundedBinarySearch(sorted, 5,0,3));

//Searching for the 3 in the elements 1 to 4

System.out.println(boundedBinarySearch(sorted, 3,1,4));

}

}