Searching and Sorting algorithms

1. Searching and Sorting algorithms (I)

1.1. Searching algorithms

In this exercise, you are going to use two searching algorithms: linear search and binary search.

Section 1. Linear search

Linear search takes an array, which can store any type of elements with any value. The algorithm goes from the beginning to the end of the array comparing each element with a reference value. If the algorithm finds an element whose value is the same as the reference value, it will return the position of that element in the array. If there is no element with that value, you can return -1 (or alternatively throw an exception).

Program a class in Java called `LinearSearch`. This class contains a method `linearSearch(int a[], int x)`, which is provided in this exercise. Furthermore, the class contains a `main` method, which creates an array of 10000 random integer values between 0 and 9999 and invokes method `linearSearch(int a[], int x)` with a random value `x` in the same range. This method also prints the resulting value of the function or indicates that it was not possible to find an element with the `x` given.

```
public static int linearSearch(int a[], int x) {
    int result = -1;
    for (int i = 0; i < a.length ; i++) {
        if (a[i] == x){
            return result=i;
        }else{
            result=-1;
        }
    }
    return result;
}
```

How long does the algorithm take to find the value which receives as parameter in the array? In order to compute this value, you can use methods `System.currentTimeMillis()` and `System.nanoTime()`. Perform at least 1000 tests to obtain the average of the results. Take into account that your processor may be parallelizing several tasks and because of that, the time for performing a single task may be sensibly higher than the average time for performing several tasks.

Section 2. Binary search

Binary search starts from an **ordered array** and a given reference value. It compares the reference value with the value which occupies the intermediate position of the array. If both values match, it means that the reference value has been found and the position is returned. If the reference value is lower than the value which occupies the central position, as we know the array is ordered (assuming ascending order), we will repeat the search in the left part of the array, which contains values lower than the element in the central position. If the reference value is higher, we will repeat the search in the right part of the array. In case the value of any element of the array is the same as the reference value, we will return the position of this element of the array. Otherwise, you can return -1 or throw an exception.

Program a class in Java called `BinarySearch`. This class contains a method `binarySearch(int a[], int x)` which is provided in the exercise. Furthermore, the class contains a `main` method, which creates an array of 10000 sorted integer values between 0 and 9999 and invokes method `binarySearch(int a[], int x)` with a random value `x` in the same range. This method also prints the resulting value of the function or indicates that there has not been found any element with the `x` given.

```
public static int binarySearch(int a[], int x) {
    int medium;
    int start = 0;
    int end = a.length -1;
    while (start <= end) {
        medium = (start + end)/2;
        if(a[medium] == x){
            return medium;
        }if(a[medium] < x){
            start = medium+1;
        }else if (a[medium] > x){
            end = medium-1;
        }
    }
    return -1;
}
```

How long does the algorithm take to find the value which receives as parameter in the array? In order to compute this value, you can use methods `System.currentTimeMillis()` and `System.nanoTime()`. Perform at least 1000 tests to obtain the average of the results. Take into account that your processor may be parallelizing several tasks and because of that, the time for performing a single task may be sensibly higher than the average time for performing several tasks.

1.2. Basic sorting algorithms

In this exercise, you are going to use three different basic sorting algorithms: `BubbleSort`, `SelectionSort`, and `InsertionSort`. You will be able to see the differences in terms of efficiency of each algorithm when sorting elements of an array.

Section 1. BubbleSort

`BubbleSort` starts from the beginning of the array and swaps pairs of neighboring data which are not in the correct order. After this process, the highest element goes to the end. There are several iterations to repeat this process in the array until there are no more elements to swap. Elements which have advanced to their final position in the end of the array do not need to be checked in the following iterations, so each iteration requires one less check.

In order to make use of this sorting algorithm, you are provided with these two methods.

```
    public static void bubbleSort (int[] a) {
        for (int i=0; i<a.length-1; i++) {
            for (int j=0; j<a.length-1-i; j++) {
                if (a[j]>a[j+1]){
                    swap(a, j, j+1);
                }
            }
        }
    }

    public static void swap (int[] a, int i, int j) {
        int aux=a[i];
        a[i]=a[j];
        a[j]=aux;
    }
```

Try to understand the behavior of these two methods.

You are asked to:

a) Implement these methods, a method `main` and all auxiliary methods you need in a Java class called `BubbleSort`. The `main` method will receive an array of integers as parameter and will print the ordered array.
b) Add a counter of iterations and a counter of swap actions to have an idea of the complexity of the algorithm. Try to guess the content of the array in each following iteration. In order to do that, set a breakpoint to inspect the array step by step. Perform tests with arrays of different values and sizes.
c) A problem of `BubleSort` is that it always performs the same number of iterations, although there may be no swaps to perform in the last iterations (because the array is already sorted with all its element in the final position). Optimize the algorithm including a variable which stores if there have been changes in the internal loop in the last iteration of the external loop and stop the algorithm otherwise. Check how many iterations and swaps are produced with this enhanced version.
d) To check the efficiency of this algorithm and to be able to compare it with other algorithms, sort an array of 10,000 random integer values between 0 and one million. In order to perform this computation, you can use methods `System.currentTimeMillis()` and `System.nanoTime()`. Comment all auxiliary lines that you have added in the previous section to know the number of iterations and swaps because they can affect the computation of time. Perform tests with arrays of different values and sizes.
e) Modify the code to perform the sorting in descending order, so that the element with highest value will be on the left side of the array and the element with lowest value will be on the right side.

Section 2. SelectionSort

SelectionSort improves the limitation of the excessive number of swaps that are produced in the BubbleSort. In order to do that (considering ascending order), it searches the minimum element of the array and puts it in the first place (to be ordered) by a swap. Afterwards, the algorithm advances a position and searches the minimum element in the rest of the array and puts it in the second position, and so on. The algorithm needs two loops, one to go through every element of the array and another to identify the minimum in the unsorted part. In each iteration, a new element is placed in its final ordered position.

In order to make use of this sorting algorithm, you are provided with these two methods.

```
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        int m = i;
        for (int j = i; j < a.length; j++) {
            if (a[j] < a[m]) {
                m = j;
            }
        }
        swap(a, i, m);
    }
}

public static void swap(int[] a, int i, int j) {
    int aux = a[i];
    a[i] = a[j];
    a[j] = aux;
}
```

Try to understand the behavior of these two methods.

You are asked to:

a) Implement these methods, a method main and all auxiliary methods you need in a Java class called SelectionSort. The main method will receive an array of integers as parameter and will print the ordered array.

b) Add a counter of iterations and a counter of swap actions to have an idea of the complexity of the algorithm. Try to guess the content of the array in each following iteration. In order to do that, set a breakpoint to inspect the array step by step. Perform tests with arrays of different values and sizes.

c) To check the efficiency of this algorithm and to be able to compare it with other algorithms, sort an array of 10,000 random integer values between 0 and one million. In order to perform this computation, you can use methods System.currentTimeMillis() and System.nanoTime(). Comment all auxiliary lines that you have added in the previous section to know the number of iterations and swaps because they can affect the computation of time. Perform tests with arrays of different values and sizes.

d) Modify the code to perform the sorting in descending order, so that the element with highest value will be on the left side of the array and the element with lowest value will be on the right side.

Section 3. InsertionSort

InsertionSort algorithm divides the array in two parts where the first one (left part) is sorted and the second one (right part) is unsorted. In each iteration, the first element in the second part is removed from that part and inserted in the corresponding positing in the first part. When the algorithm finishes, all the elements are sorted in the first part.

In order to make use of this sorting algorithm, you are provided with this method.

```
public static void insertionSort(int[] a){
    for (int i=0; i<a.length; i++){
        int tmp=a[i];
        int j=i;
        while (j>0 && tmp<a[j-1]){
            a[j]=a[j-1];
            j--;
        }
        a[j]=tmp;
    }
}
```

Try to understand the behavior of this method.

You are asked to:

a) Implement this method, a method main and all auxiliary methods you need in a Java class called InsertionSort. The main method will receive an array of integers as parameter and will print the ordered array.

b) Add a counter of iterations and a counter of swap actions to have an idea of the complexity of the algorithm. Try to guess the content of the array in each following iteration. In order to do that, set a breakpoint to inspect the array step by step. Perform tests with arrays of different values and sizes. NOTE: The concept of swapping in `InsertionSort` is not the same as the concept of swapping in `BubbleSort` and `SelectionSort`.
c) To check the efficiency of this algorithm and to be able to compare it with other algorithms, sort an array of 10,000 random integer values between 0 and one million. In order to perform this computation, you can use methods `System.currentTimeMillis()` and `System.nanoTime()`. Comment all auxiliary lines that you have added in the previous section to know the number of iterations and swaps because they can affect the computation of time. Perform tests with arrays of different values and sizes.
d) Modify the code to perform the sorting in descending order, so that the element with highest value will be on the left side of the array and the element with lowest value will be on the right side.

Section 4. Establishing conclusions.

Test with the different algorithms and take notes about the number of iterations, swaps and average time (repeat the test several times and compute an average time) for each of the following input arrays:
a) An array of 8 ordered elements
b) An array of 8 elements where 4 of them are out of order
c) An array of 8 elements which are in reverse order (from higher to lower value)
d) An array of 100 random elements with values between 0 and one million
e) An array of 10,000 random elements with values between 0 and one million

Which of the three algorithms is the most efficient? Is that algorithm always the most efficient or does it depend on the input? What does it affect (or prejudice) the most to each algorithm, the input size (number of elements of the array) or the initial sort state?

Hint: Create a new class that uses the other three classes to invoke the different sorting methods with different arrays programmatically and not using arguments in the command line. In order to execute different algorithms with exactly the same array, you can use the method `close()` to create and duplicate an array. This is important because after finishing the execution of an algorithm, the array is ordered and will not be valid to be used in the following algorithm.