Searching and Sorting algorithms

1. Searching and Sorting algorithms (II)

1.1. Advanced sorting algorithms

In this exercise, you are going to use three different advanced sorting algorithms: `HeapSort`, `MergeSort`, and `QuickSort`. You will be able to realize the differences in terms of efficiency of each algorithm when sorting elements of an array. In the last part, you will compare these three advanced algorithms with the three basic algorithms covered before: `BubbleSort`, `SelectionSort`, and `InsertionSort`.

Section 1. HeapSort

`HeapSort` algorithm is based on the use of a complete heap for sorting. In the first step, a heap is built with the information of the array. It is necessary to take into account that if the parent is in the position `i` and you start with the root in `i` = 0, the left child will be in the position `2*i + 1`, and the right child in `2*i + 2`. Furthermore, if we assume a *max-heap* (a heap whose maximum value is in the root), all children will be lower than the parent. In a second step, the element with the highest value in the heap (root) will be removed and will be inserted at the end of the array. The heap will be reordered conveniently.

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

```java
public static void heapSort(int[] a) {
    for (int i = (a.length - 2) / 2; i >= 0; i--)
        heapify(a, i, a.length - 1);

    for (int i = a.length - 1; i >= 0; i--) {
        swap(a, 0, i);
        heapify(a, 0, i - 1);
    }
}

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

private static void heapify(int[] a, int i, int m) {
    int j;
    while(2*i+1<=m) {
        j=2*i+1;
        if(j<m) {
            if(a[j]<a[j+1])
                j++;
        }
        if(a[i]<a[j]) {
            swap(a,i,j);
            i=j;
        } else
            i=m;
    }
}
```

Try to understand the behavior of these three methods.

You are asked to:

a) Implement these methods, a method `main` and all auxiliary methods you need in a Java class called `HeapSort`. 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()`. Delete 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. To do that, make use of a *min-heap*, instead of a *max-heap*.

Section 2. MergeSort

`MergeSort` algorithm is based in the recursive division of the data structure which is going to be sorted in substructures with similar length and the subsequent ordered merging of the elements in each substructure. In the case of arrays, the array is divided in two subarrays of similar length recursively until reaching subarrays of length 1. Afterwards, subarrays are merged progressively and elements are ordered (from the lowest to the highest value in ascending order) in the merging process. Elements are only placed in their final ordered position after merging subarrays in a single array with the same length as the original array.

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

```java
public static void mergeSort (int[] a) {
    mSort(a, 0, a.length-1);
}


public static void mSort (int[] a, int l, int r) {
    if (l>=r){
        return;
    }
    int m = (l+r)/2;
    mSort(a, l, m);
    mSort(a, m+1, r);
    merge(a, l, m, r);
}

public static void merge (int[] a, int l, int m, int r){
    if (m+1>r){ return; }
    int[] b = new int[a.length];
    for (int i=l; i<m+1; i++) {
        b[i] = a[i];
    }
    for (int i=m+1; i<r+1; i++){
        b[i] = a[r+m+1-i];
    }
    int k=l;
    int j=r;
    for (int i=l; i<r+1; i++) {
        if (b[k] <= b[j]){
            a[i] = b[k];
            k++;
        } else {
            a[i] = b[j];
            j--;
        }
    }
}
```

Try to understand the behavior of these three methods.

You are asked to:

a) Implement these methods, a method `main` and all auxiliary methods you need in a Java class called `MergeSort`. 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. *Take into account that in this case you need to create an auxiliary array and hence you need to first transfer the non-ordered information from the original array to the auxiliary array, and then, the information is being ordered in the merging step from the auxiliary array to the original array. A swap of positions implies coping two elements from the original array to the auxiliary array and placing afterwards these two elements in their corresponding positions in the original array.* 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()`. Delete 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. QuickSort

`Quicksort` algorithm is based on the choice of a datum as a pivot. Elements which are lower than the pivot must be on the left, while elements higher than the pivot must be on the right. This process generates two partitions: one partition for the elements lower than the pivot and another for the elements higher than the pivot. Each partition can be sorted independently in a recursive way following the same process of choosing a datum as a pivot and resorting the elements by putting elements which are lower than the pivot on the left and elements higher than the pivot on the right.

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

```
public static void quickSort (int[] a) {
    qSort(a, 0, a.length-1);
}

public static void qSort (int[] a, int l, int r) {
    if (l>=r){
        return;
    }
    int m = partition(a, l, r);
    qSort(a,l,m-1);
    qSort(a,m+1,r);
}

public static int partition (int[] a, int l, int r) {
    int i=l+1; // left
    int j=r; // right
    int p=a[l]; // pivot
    while (i<=j) {
        if (a[i]<=p){
            i++;
        } else if (a[j]>p){
            j--;
        } else{
            swap(a,i,j);
        }
    }
    swap(a,l,j);
    return j;
}

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 four methods.

You are asked to:

a) Implement these methods, a method `main` and all auxiliary methods you need in a Java class called `QuickSort`. 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()`. Delete 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 the different advanced algorithms (`HeapSort`, `MergeSort`, and `QuickSort`) and basic algorithms (`BubbleSort`, `SelectionSort`, and `InsertionSort`) 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 6 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 six 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.