Activity 1. Time measurements for sorting algorithms

1. *BUBBLE*

n	sorted(t) milliseconds	inverse(t)	random(t)
10000	90	127	276
20000	150	427	727
40000	602	1983	3417
80000	1854	6867	12685
160000	8323	29645	58297
320000	21591	124782	227014
640000	129169	489111	828521

This algorithm has complexity O(n^2) in all the cases. As this is a bad complexity, the algorithm is not very useful. It can be used in small problems, but not with huge numbers (as in the third column, where we are using big random numbers) because it can last a lot of time to end its execution.

According to the following results, we can conclude that the values obtained meet the expectations of the complexity:

We know that **t2 = (f(n2) / f(n1)) x t1**, being f(n) = n^2

<u>(Sorted)</u> <u>Taking this values:</u>

n1 = 20000 t1 = 150

n2 = 40000 t2 = 602

t2 = (40000^2/20000^2) x 150 = 600 ≈ 602

Algorithmics	Student information	Date	Number of session
	UO: 284185	22/2/22	2
	Surname: Fernández-Catuxo Ortiz		
	Name: Rita		

(Inverted) Taking this values:

n1 = 20000 t1 = 427

n2 = 40000 t2 = 1983

t2 = (40000^2/20000^2) x 427 = 1708 ≈ 1983

(Random) Taking this values:

n1 = 20000 t1 = 727

n2 = 40000 t2 = 3417

t2 = (40000^2/20000^2) x 727 = 2908 ≈ 3417

2. *SELECTION*

n	sorted(t) milliseconds	inverse(t)	random(t)
10000	50	283	57
20000	78	530	391
40000	309	978	1198
80000	1529	3319	3664
160000	4202	11906	13281
320000	19623	62308	60086
640000	125696	208522	236164
1280000	362777	727267	864572

Algorithmics	Student information	Date	Number of session
	UO: 284185	22/2/22	2
	Surname: Fernández-Catuxo Ortiz		
	Name: Rita		

This algorithm is very similar to the bubble (they have the same complexity O(n^2) in all the cases). Comparing the execution times, it is faster and more efficient than the Bubble when sorting an already sorted list and a random list. However, it is still a bad algorithm.

According to the following results, we can conclude that the values obtained meet the expectations of the complexity (except from the inverted list, where the results don't make sense):

We know that **t2 = (f(n2) / f(n1)) x t1**, being f(n) = n^2

(Sorted) Taking this values:

n1 = 20000 t1 = 78

n2 = 40000 t2 = 309

t2 = (40000^2/20000^2) x 78 = 312 ≈ 309

(Inverted) Taking this values:

n1 = 20000 t1 = 530

n2 = 40000 t2 = 978

t2 = (40000^2/20000^2) x 530 = 2120 ≠ 978

(Random) Taking this values:

n1 = 20000 t1 = 391

n2 = 40000 t2 = 1198

t2 = (40000^2/20000^2) x 391 = 1564 ≈ 1198

Algorithmics	Student information	Date	Number of session
	UO: 284185	22/2/22	2
	Surname: Fernández-Catuxo Ortiz		
	Name: Rita		

3. *INSERTION*

n	sorted(t) milliseconds	inverse(t)	random(t)
10000	79	126	121
20000	71	462	379
40000	174	475	203
80000	387	1870	778
160000	780	7846	4821
320000	1676	36051	19326
640000	3420	137205	68439
1280000	8212	669841	278437

The insertion algorithm (with complexity O(n^2) and O(n) in its best case) is the one that has the best execution time when we are sorting an already sorted list (this is the case when the complexity is O(n)).

At first, I ran the algorithm with a sorted array with 1 repetition. The execution time was less than 50 milliseconds (as it is faster than the other algorithms). As these execution times were not reliable, I increased the repetitions up to 10.000 (when we started to get reliable times). The other two columns are measured with only one repetition.

However, when the list is not sorted, the execution times get worse so, as the other two previous algorithms, this one is still a bad one.

According to the following results, we can conclude that the values obtained meet the expectations of the complexity :

We know that **t2 = (f(n2) / f(n1)) x t1**, being f(n) = n

Algorithmics	Student information	Date	Number of session
	UO: 284185	22/2/22	2
	Surname: Fernández-Catuxo Ortiz		
	Name: Rita		

<u>(Sorted)</u> <u>Taking this values:</u>

n1 = 20000 t1 = 71

n2 = 40000 t2 = 174

t2 = (40000/20000) x 71= 142 ≈ 174

Now, f(n) = n^2

<u>(Inverted)</u> <u>Taking this values:</u>

n1 = 40000 t1 = 475

n2 = 80000 t2 = 1870

t2 = (80000^2/40000^2) x 475 = 1900 ≈ 1870

<u>(Random)</u> <u>Taking this values:</u>

n1 = 40000 t1 = 203

n2 = 80000 t2 = 778

t2 = (80000^2/40000^2) x 203 = 812 ≈ 778

4. QUICKSORT CENTRAL ELEMENT

n	sorted(t) milliseconds	inverse(t)	random(t)
10000	151	109	167
20000	155	117	234
40000	258	241	452
80000	229	418	880
160000	274	361	2145
320000	623	744	4365
640000	1447	1635	9351
1280000	2666	3578	18191

According to the results, this is the best sorting algorithm. It has a complexity of O(n logn) and O(n^2) in its worst case. To measure it and get reliable values I used 100 repetitions.

According to the following results, we can conclude that the values obtained meet the expectations of the complexity :

We know that **t2 = (f(n2) / f(n1)) x t1**, being f(n) = nlogn

(Sorted) Taking this values:

n1 = 20000 t1 = 155

n2 = 40000 t2 = 258

t2 = (40000*log 40000/20000*log 20000) x 155 = 331 ≈ 258

(Inverted) Taking this values:

n1 = 20000 t1 = 117

n2 = 40000 t2 = 241

t2 = (40000*log 40000/20000*log 20000) x 117 = 250 ≈ 241

Algorithmics	Student information	Date	Number of session
	UO: 284185	22/2/22	2
	Surname: Fernández-Catuxo Ortiz		
	Name: Rita		

(Random) Taking this values:

n1 = 20000 t1 = 234

n2 = 40000 t2 = 452

t2 = (40000*log 40000/20000*log 20000) x 234 = 500 ≈ 452

To sum up, the first three algorithms (buble, insertion and selection) are bad options because of their bad complexity and execution times. However, quicksort central element is a very good option because apart from its good execution times, it has a very good complexity (except for its worst case).

Activity 2. Quicksort Fateful

We must know that the pivot selection can affect the algorithm's performance. That's the reason why the quicksort central element has a good performance (because its pivot is a good choice to perform the algorithm). However, in the QuicksortFateful class the pivot selected is a bad choice.

This algorithm consists in selecting as the pivot the first (leftmost) element of the partition. This choice causes worst-case behavior on already sorted arrays, which is a rather common use-case. Once selected, we exchange it with the last right element and start the algorithm.
Its complexity degrades to O(n^2) for already sorted arrays. One of the reasons is because there are much more swaps than in the other options.

Therefore, the idea will work in not sorted arrays, and will not work in already sorted arrays. Some examples:

1. The algorithm working in not sorted arrays (inverted and random)

```
Time Measurement: Quicksort - Fateful
n=10000**TIME=172
n=20000**TIME=245
n=40000**TIME=416
n=80000**TIME=1124
n=160000**TIME=2508
n=320000**TIME=4497
n=640000**TIME=8697
n=1280000**TIME=17806
```

```
Time Measurement: Quicksort - Fateful
n=10000**TIME=223
n=20000**TIME=229
n=40000**TIME=464
n=80000**TIME=867
n=160000**TIME=1853
n=320000**TIME=3841
n=640000**TIME=9402
```

2. The algorithm not working in a sorted array

```
Time Measurement: Quicksort - Fateful
n=10000**TIME=2904
Exception in thread "main" java.lang.StackOverflowError
	at algstudent.s2.QuicksortFateful.quickSort(QuicksortFateful.java:44)
	at algstudent.s2.QuicksortFateful.quickSort(QuicksortFateful.java:55)
	at algstudent.s2.QuicksortFateful.quickSort(QuicksortFateful.java:55)
	at algstudent.s2.QuicksortFateful.quickSort(QuicksortFateful.java:55)
	at algstudent.s2.QuicksortFateful.quickSort(QuicksortFateful.java:55)
	at algstudent.s2.QuicksortFateful.quickSort(QuicksortFateful.java:55)
	at algstudent.s2.QuicksortFateful.quickSort(QuicksortFateful.java:55)
	at algstudent.s2.QuicksortFateful.quickSort(QuicksortFateful.java:55)
	at algstudent.s2.QuicksortFateful.quickSort(QuicksortFateful.java:55)
```