Homework 1: Sort Algorithms

Fabián Andrés Prada Niño

1) **Algorithm's Implementation**

The sort algorithms were implemented in C++ (look at the attached file *main.cpp*). The order of implementation difficulty (in my project) was the following:

Heapsort>>**Mergesort**>>**Quicksort**>>**Insertion**>>**Selection**

- **Selection** and **Insertion** were easily implemented since these algorithms can be executed using a single method (*selecao()* and *insercao()* respectively). **Selection** only requires a *for* loop, and **Insertion** a *for* loop and a *while*
- **Mergesort** is executed in two methods: *mergesort()* and *merge()*. *mergesort()* works recursively and invokes *merge()* which works in a single step. *merge()* requires the construction of a new vector of memory to store the mixed lists.
- **Quicksort** has a similar implementation structure than **Mergesort**: it is also executed in two methods, *quicksort()* and *particionar()*, where the first works recursively and invokes the second which works in a single step. However, the implementation of *particionar()* is easier than *merge()* because it does not require extra memory. In both cases you require two pointers to perform each process.
- **Heapsort** was the hardest to be implemented since it requires of three methods: *heapify(), construcaoMaxHeap*(), and *heapsort(). Heapify()* is a recursive method that assigns to a node the correct position in a subtree in order to make such subtree a correct heap. *construcaoMaxHeap()* works in a single step and invokes *heapify()* to construct the initial heap. Finally, *heapsort()* is the method that organizes the sorting process: first, it calls *construcaoMaxHeap()*, and iteratively it changes the root with the last node of the heap and invokes *heapify()* applied to the new root.

2) Comparison of performance in general random lists

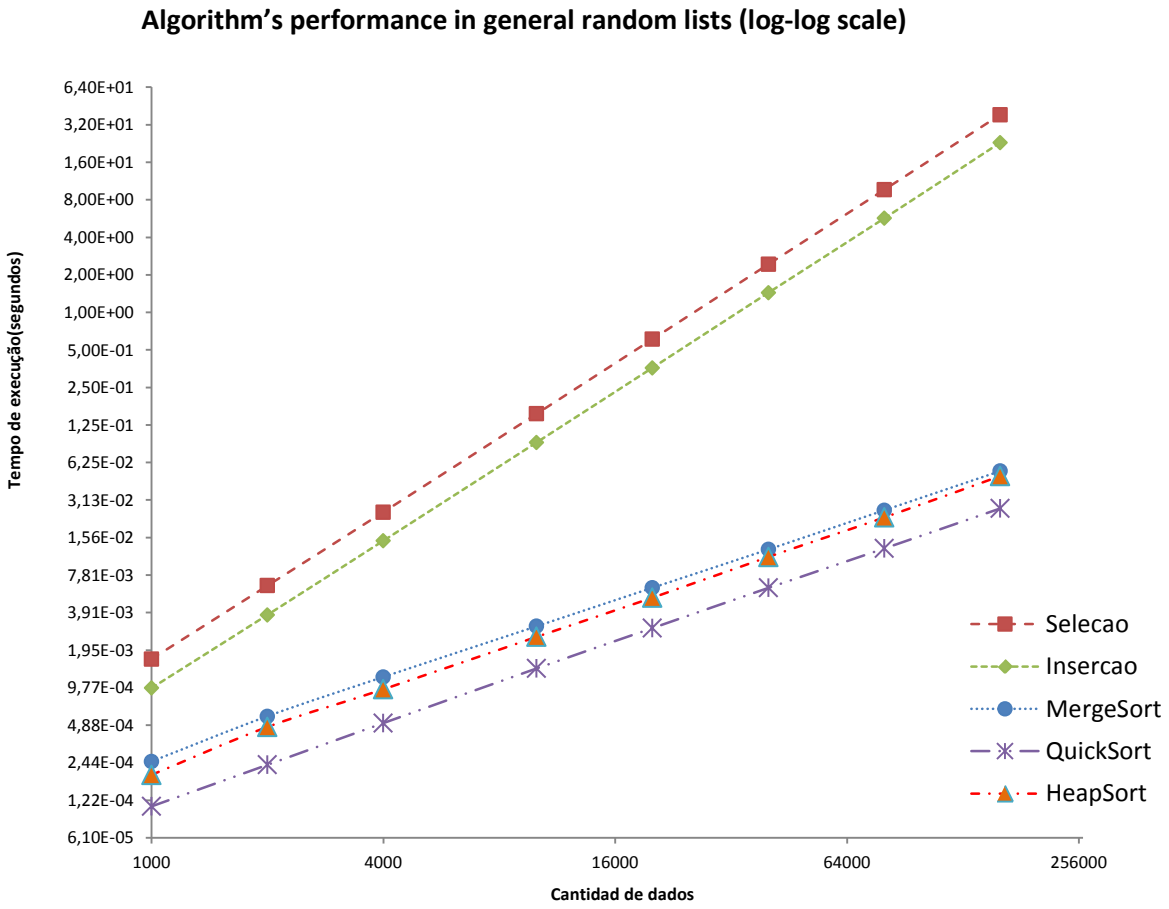
In order to compare the performance of sort algorithms I developed the following experiment:

- Each algorithm was executed with lists of 8 different sizes: 1000, 2000, 4000, 10000, 20000, 40000, 80000, and 160000.
- The list of size *n* was constructed taking a random sample of numbers in the set *{1,2,3,…n}.* (see *generateRandomVector()*).
- For each size, the sort algorithms were executed 13 times (ordering a different list in each execution) and the mean time of performance was calculated from the last 10 execution.

The order of performance I obtained for general random list was this:

Quicksort>>**Heapsort**>>**Mergesort**>> **Insertion**>>**Selection**

The following figure summarizes the obtained results:

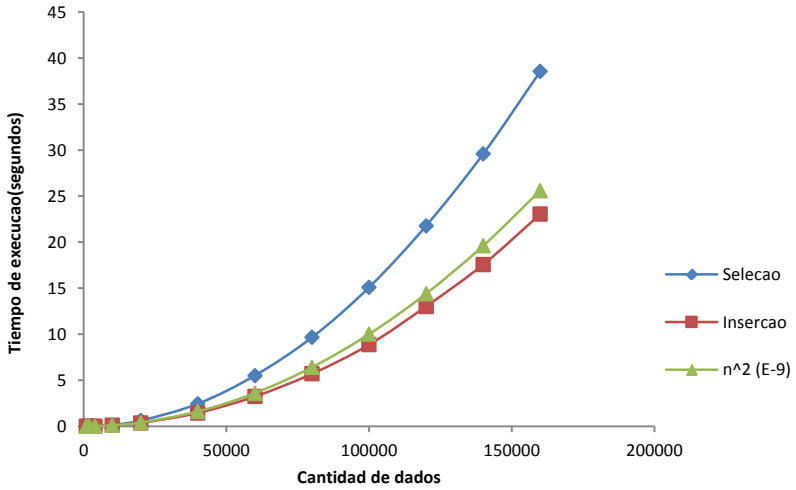


As expected, we can classify the previous sort algorithms in two order of performance: **Selection** and **Insertion** belongs to an order of performance $O(n^2)$, while **Mergesort, Quicksort** and **Heapsort** have an order of performance $O(nlogn)$.

We can deduce the approximate order of performance of these algorithms as follows:

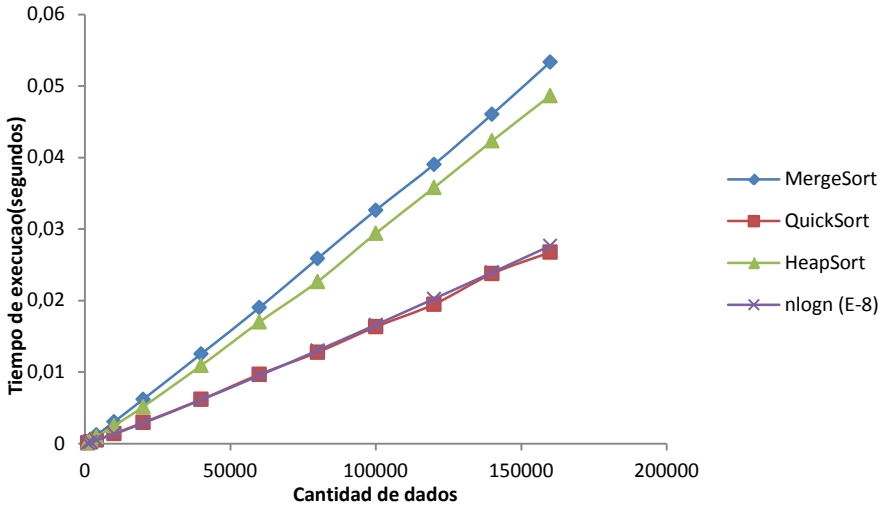
- **Selection**-**Insertion**: Since these curves are almost parallels, we can deduce that they share the same functional type (order) up to a multiplicative factor. In the log-log figure we can check that the slope of these curves is approximately 2, this means that if the size of the sample is duplicated, the sort time will increase approximately 4 times. This is precisely the behavior that defines $O(n^2)$ functions . Therefore, the experimental results obtained are coherent with the theoretical $O(n^2)$ order of these algorithms. The following linear-linear scale graph corroborates the hypothesis (I made extra experiments with 60000, 1000000, 120000, and 140000 for better precision):

Selection and Insertion performance in general random lists (linear-linear scale)



- **Mergesort-Quicksort-Heapsort:** Again, the parallelism between lines lead us to confirm that these algorithms share the same functional type (order) up to a multiplicative factor. In this case the slope analysis must be more elaborated: If we just observe the mean slopes of the curves we get it is approximately 1 (duplicating sample approximately duplicates times). This behavior is almost true for functions *O(nlogn)* (when *n* tend to infinity) but this behavior is specially characteristic of *O(n)* functions. In practice, differentiate between *O(n)* and *O(nlogn)* behavior from a simple graph is a difficult task. Here is a figure which illustrates the results in linear-linear scale:

Mergesort-Quicksort-Heapsort performance in general random lists (linear-linear scale)

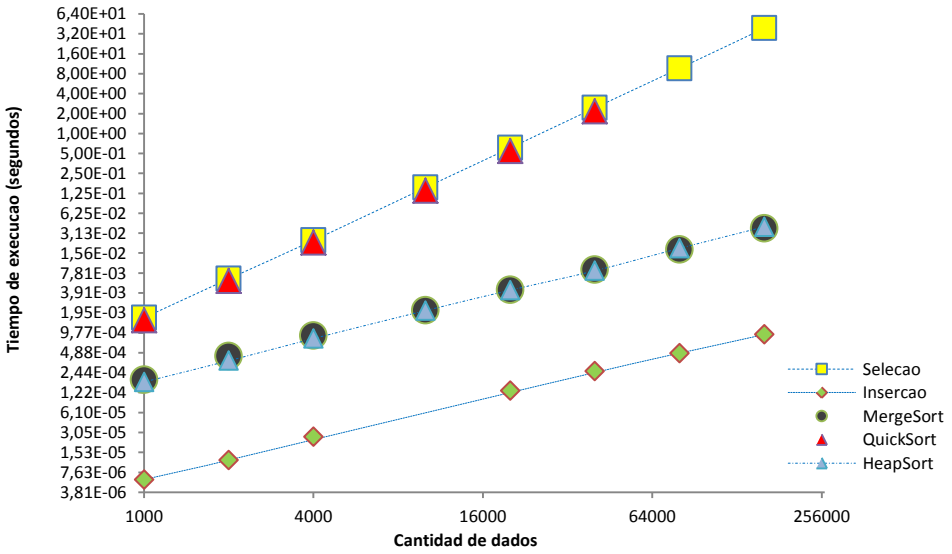


It is important to notice that *nlogn* is a strongly convex function (i.e., second derivative is strictly positive) while *n* is not. In the graph, the curves of the algorithms seem a bit strongly convex!

3) Comparison of performance in special lists

a) Increasing order lists

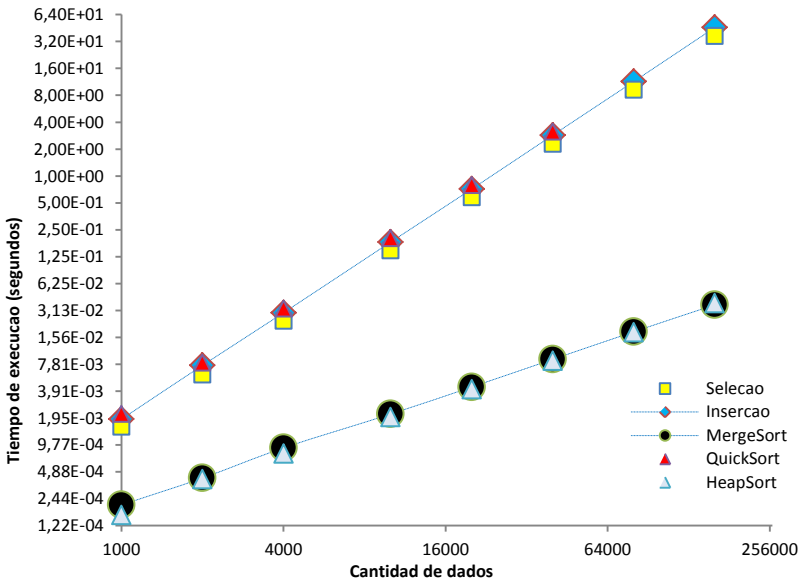
Algorithm's performance in increasing order lists (log-log scale)



- As expected, **Selection** does not change its performance in comparison to the general random list. Even in this situation it is $O(n^2)$.
- **Insertion** improved a lot its performance and was the best algorithm for this case. It is possible to check that in ordered lists it operated in $O(n)$.
- **Quicksort** become worse than in general random lists, and in this case its behavior is identical to **Selection** (since in each step it must visit the tail of the list and it only gets to sort one element). Then it run in $O(n^2)$. (Some data is missing because I got Segmentation Faults for Quicksort in lists of size greater than 80000).
- **Mergesort** and **Heapsort** shares a similar behavior in ordered lists (still $O(nlogn)$), but they perform a bit better than in random list. In the case of **Mergesort** the merge steps are easily run since there are no intercalation of lists (i.e., one list is pasted just after the other).

b) Decreasing order lists:

Algorithm's performance in decreasing order lists (log-log scale)



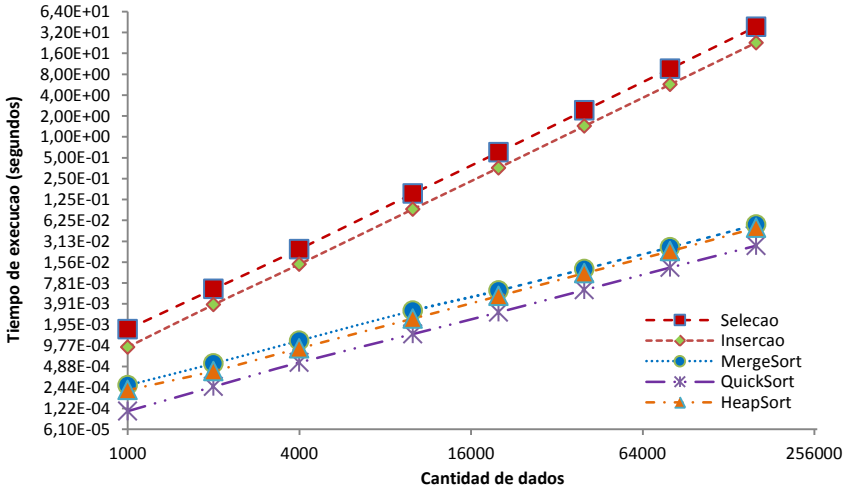
- **Selection**, **Insertion** and **Quicksort** must visit the tail of the list in each step and they only get to fix a number. For this reason they present the a similar $O(n^2)$ behavior. As expected, **Insertion** performs worse with decreasing lists than in general random lists.

- **Mergesort** and in **Heapsort** still behaves in a similar way. In decreasing lists they both perform better than in general random lists. **Mergesort** performed almost identically in increasing and decreasing sort, while **Heapsort** improved a little in the decreasing case. Take into account that for decreasing lists the construction of the *maxHeap()* does not require to call *heapify()* anytime.

c) List with few repetitions:

In this case I take random lists of size *n* in the interval *{1, 2,…, 1000n}.*

Algorithm's performance in lists with few repetitions (log-log scale)

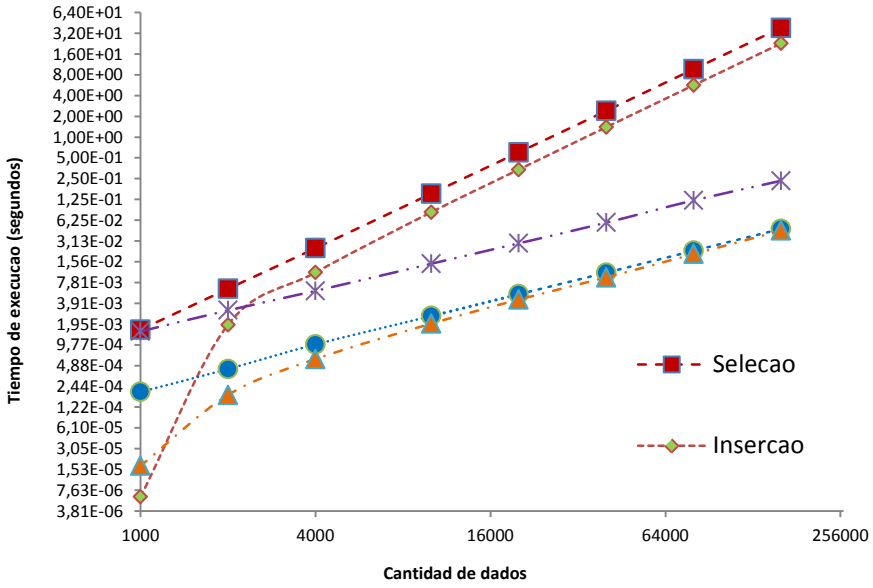


- The results were almost identical to the explained in the general random case. There were no remarkable changes.

d) Lists with many repetitions:

In this case I take random lists of size *n* in the interval *{1,2,…,n/1000}*

Algorithm's performance in lists many repetitions (log-log scale)

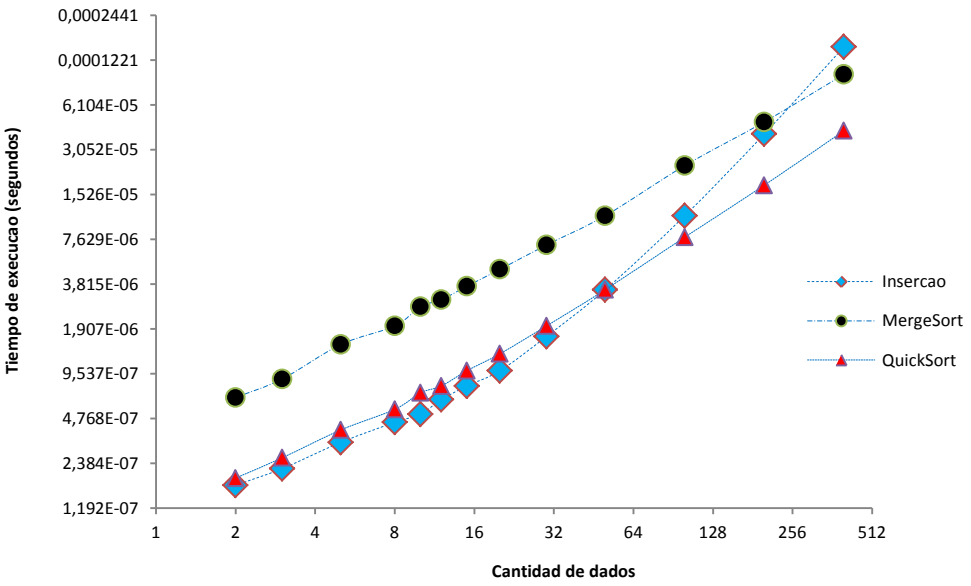


- **Insertion** started being a good algorithm (when the list only has few categories of values), but when the quantity of categories increased, the behavior of insertion is almost the same than in the general random case.
- **Quicksort** performs worse in this case than in the general random case. However it is important to realize that in this case its behavior seems to still being of order *O(nlogn),* instead of order $O(n^2)$ as in increasing or decreasing order lists.
- **Heapsort** and **Mergesort** had a slightly better behavior in this case than in general random lists. Also, their behavior was slightly worse than in increasing and decreasing lists. This phenomenon was not studied in detail during the project, but probably the block structures that are built along the algorithm execution (in the case of many repetitions) lead to a reduction of the computational time. For instance, when you merge "blocks", it is more likely that once you have concluded to visit one of the lists (left/ right), there are many "blocks" in the other list that have not been visited yet. These "blocks" are carried to the end of the new list using a *for* loop. Instead, when the lists (left/right) are completely random they tend to conclude at the same time and this requires many more comparisons.

4) **Adding Insertion to Quicksort and Mergesort**

Before proposing the new algorithm's implementation, let's observe the behavior of **Insertion, Mersgesort,** and **Quicksort** in small general random lists. I obtained the following results:

Algorithm's performance in small general random lists (log-log scale)



Size(n)	*Insertion(n)/Quicksort(n)*	*Insertion(n)/Mergesort(n)*
2	0,89	0,26
3	0,85	0,25
5	0,83	0,22
8	0,82	0,23
10	0,72	0,19
12	0,81	0,21
15	0,79	0,21
20	0,77	0,21
30	0,85	0,24
50	1,00	0,32
100	1,39	0,46
200	2,21	0,83
400	3,66	1,53

- **Insertion** was faster than **Quicksort** for lists of less than *50* numbers. The lowest ratio *Insertion(n)/Quicksort(n)* was obtained at *t=10,* and it was *0,72* (i.e, **Insertion** expends *0,72* times the expenditure of **Quicksort**). In the interval *{2,…,20}* this ratio was about *0,8*.

- Insertion was faster than Mergesort for lists of less than *200* numbers. The lowest ratio *Insertion(n)/Mergesort(n)* was at *t=10* , and it was *0,19*. In the interval *{2,…,30}* this ratio remains about *0,25*.

From the previous observations, I selected the size ***L=20*** as the ideal length of subproblem such that it is more convenient to invoke **Insertion** to sort that segment, instead of invoke **Mergesort** or **Quicksort** again.

The algorithms implemented were the following:

a) **Quicksort+ Partial Insertion**: Quicksort is recursively executed (as standard) and once you get a partition of length less or equal to *L,* this partition is immediately sorted using Insertion algorithm.
b) **Quicksort+ Final Insertion**: Quicksort is recursively executed (as standard) and once you get a partition of length less or equal to *L,* this partition is ignored and it continues partitioning the greater partitions. Once all the partitions are of length less or equal to *L* the Insertion algorithm is executed.
c) **Mergesort+ Partial Insertion**: Mergesort is recursively executed (as standard) and once your left and right lists are of length less or equal to *L,* they are sorted independently using Insertion algorithm, and then effectively merged.
d) **Mergesort+ Final Insertion**: Mergesort is recursively executed (as standard) and once your left and right lists are of length less or equal to *L,* Mergesort stops "getting depth"

and immediately merges these two lists (certainly, this could have no sense since the merge is done with unsorted lists). Finally Insertion is run over the whole list. (I prefer this implementation, instead of the one where you apply merge except in the last step (with the halves of the original list) and then run Insert. For me this proposed implementation is more curious. In the other implementation, it is expected that once the Insertion get the second half of the list, the element in position *n/2+k* will be moved *n/2-k* positions ahead, what leads to $O(n^2)$ behavior).

The following table summarizes the results obtained:

Algorithm's performance in general random lists (time in seconds)

Size(n)	Quicksort	Q+PI	Q+FI	Mergesort	M+PI	M+FI
10000	0,00139226	0,00135037	0,00125783	0,00303494	0,00182724	0,080013
40000	0,00616702	0,0058013	0,0057037	0,012551	0,00781569	1,26136
80000	0,0127756	0,0123808	0,012241	0,0258848	0,0170605	5,1634
160000	0,0267531	0,0259126	0,0255233	0,0533563	0,0353732	20,7891

- The **Insertion** subroutine improved the performance of **Mergesort** and **Quicksort**. In the case of **Quicksort**, implementing **Partial Insertion** improved performs a bit and **Final Insertion** even a bit more. In both cases improvement was less than 5%, which is still far to the 20% that could be expected from the analysis of small lists sorting. Probably the improvement is not so good because the time that **Quicksort** expends partitioning big lists is much higher than the time it expends sorting all the small lists.
- The improvement from **Mergesort** to **M+PI** is very notorious and it was about 40%. Since **Insertion** is much better than **Mergesort** for small lists, it was expected a good improvement, and indeed we get it. Again, we are yet far to the 75% improvement that could be expected from the analysis of small lists sorting.
- As expected, merging unsorted lists is a nonsense operation and effectively it leads to bad results. We observe that **M+FI** had a behavior $O(n^2)$, and it is a bit better than **Insertion**. This is because merging unsorted lists produce lists that are "sorted by segments" (like 5,6,7,1,2,3,6,8,9) and do **Insertion** in such cases is probably faster than in completely random lists.