Luvisutto Alberto, Algorithmic design exam

Homework: Sorting

Ex.1

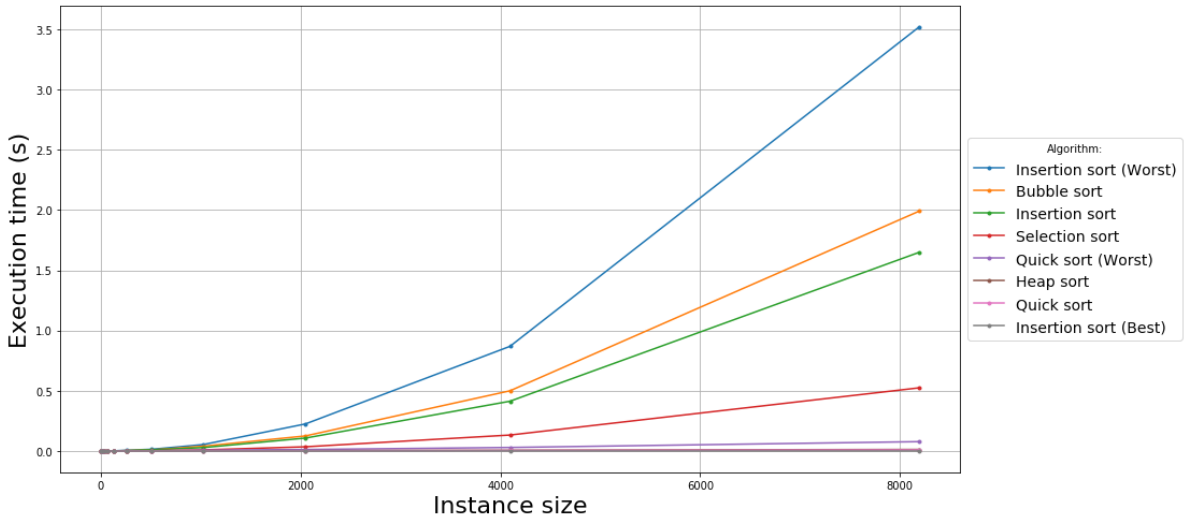
By using the given code, implement `insertion_sort`, `quick_sort`, `bubble_sort`, `selection_sort` and `heap_sort`.

The implementation of each of these algorithms can be found in the respective .c file in the `src` directory. I used the swap version of the binary heap library implemented for `heap_sort`.

Ex.2

For each of the implemented algorithms, draw a curve to represent the relation between the input size and the execution-time.

I report here all the curves requested in a single graph, reporting some of the times printed out by `test_sorting`.

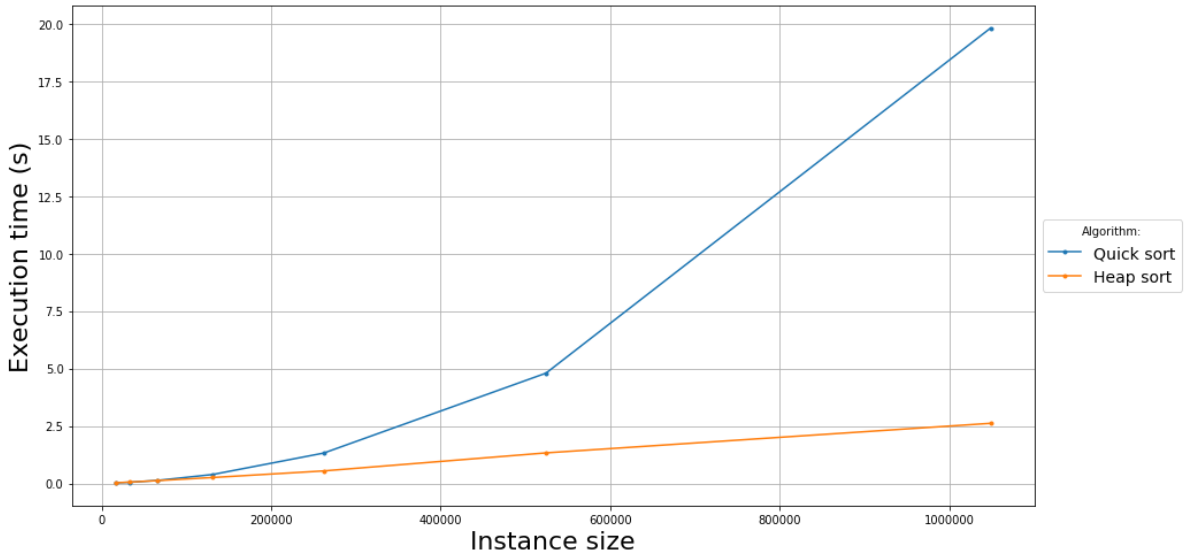


The legend specifies when the case considered isn't the random one.

As visible, considering only the random cases, the algorithms that performed better were `heap_sort` and `quick_sort`.

Insertion sort performs really well in the best case scenario, but is the worst when considering the worst case.

In the following graph instead, I report the two analysis for `heap_sort` and `quick_sort`, performed on bigger instance sizes.



It's clearly visible that, increasing the instance size, `heap_sort` outperforms `quick_sort`.

Ex.3

Argue about the following statement and answer the questions

- **HEAP SORT on an array A whose length is n takes time $O(n)$**

  In the general case, this statement is false. In fact, the Heap sort algorithm complexity is determined by:

  - The `build_heap` function, called once, which takes time $\Theta(n)$;
  - n calls of the `extract_min` function, which takes time $O(\log n)$, for a total of $n \times O(\log n) \approx O(n \log n)$

  So, globally, the complexity of the general case is $\Theta(n) + O(n \log n) \approx O(n \log n)$.

- **HEAP SORT on an array A whose length is n takes time $\Omega(n)$**

  In the (best) case in which, for example, we have that all the terms of the array are equal, the `extract_min` function takes time $\Theta(1)$, since we don't need to restore the heap property after the extraction.

  In this way, we have a global complexity of $\Omega(n) + n \times \Theta(1) \approx \Omega(n)$.

- **What is the worst case complexity for HEAP SORT?**

  As stated in the first answer, the worst case complexity of `heap_sort` is $\Theta(n) + O(n \log n) \approx O(n \log n)$.

- **QUICK SORT on an array A whose length is n takes time $O(n^3)$**

  Technically speaking, the previous statement is true, since the worst case complexity for `quick_sort` is $O(n^2) \in O(n^3)$. However, it's always better to use bounds as strictly as possible.

- **What is the complexity of QUICK SORT?**

  As stated before, the worst case complexity of `quick_sort` is $O(n^2)$, reached when we obtain at each time the most unbalanced possible partition. The best case complexity instead, reached when we perform a balanced partitioning at each step, is $\Theta(n \log n)$, which is also the average case complexity.

- **BUBBLE SORT on an array A whose length is n takes time $\Omega(n)$**

The previous statement is true. In fact, if we write the code of `bubble_sort` as in the pseudo-code reported in the slides, we have two nested for-loops that, in the end, brings the best case complexity (already sorted array) to $\Omega(n^2) \in \Omega(n)$ (as in point d, we could say that it is always better to use bounds as strictly as possible).

However, if we vary a bit the algorithm, inserting a `return` when we see that no swaps have been performed in an inner for-loop, the best case scenario of an already sorted array consists in a simple linear check with no swaps after the first n iterations, bringing the time complexity to $\Omega(n)$, which again makes the previous statement true.

- **What is the complexity of BUBBLE SORT?**

  As stated before, in the "modified" algorithm, the best case complexity of `bubble_sort` is $\Omega(n)$, while the worst case complexity is $O(n^2)$.

  If instead we consider the algorithm presented in the slides, the time complexity of `bubble_sort` becomes $\Theta(n^2)$.

Ex.4

Solve the recursive equation (reported in 05_homework.pdf)

Let's take $cn^{\frac{3}{2}}$, with $c > 0$, as the function representative for the complexity class $\Theta(n^{\frac{3}{2}})$.

Drawing the recursive tree, we obtain a tree with 3^i nodes per level, with a complexity of $c\left(\frac{n}{4^i}\right)^{\frac{3}{2}}$ per node, where i is the level of the node, with the root level being $i = 0$.

The last level, characterized by definition by a cost of $\Theta(1)$ per node, will be reached when $\frac{n}{4^i} = 32$, which means the last level will be

$$i = \log_4 n - \frac{5}{2}$$

If we substitute this value of i we found in the previously found formula, we obtain a cost for the last level of

$$\Theta(1) \cdot 3^{\log_4 n - \frac{5}{2}} = \Theta(3^{\log_4 n} \cdot 3^{-\frac{5}{2}}) = \Theta(n^{\log_4 3} \cdot 3^{-\frac{5}{2}}) \approx \Theta(n^{\log_4 3})$$

So this brings the global complexity to

$$\Theta(n^{\log_4 3}) + \sum_{i=0}^{log_4 n - \frac{5}{2} - 1} c \cdot \left(\frac{3}{8}\right)^i \cdot n^{\frac{3}{2}} < \Theta(n^{\log_4 3}) + c \cdot n^{\frac{3}{2}} \cdot \sum_{i=0}^{+\infty} \left(\frac{3}{8}\right)^i = \Theta(n^{\log_4 3}) + c \cdot n^{\frac{3}{2}} \cdot \frac{8}{5} \approx O\left(n^{\frac{3}{2}}\right)$$

Since $\frac{3}{2} > \log_4 3$.

So the overall complexity is $O\left(n^{\frac{3}{2}}\right)$.