Homework 04: Sorting

Roberto Corti

July 11, 2020

Exercise 1

By using the code at:

`https://github.com/albertocasagrande/AD_sorting`

implement `INSERTION SORT`, `QUICK SORT`, `BUBBLE SORT`, `SELECTION SORT`, and `HEAP SORT`.

These functions are respectively implemented into the files `src/insertion_sort.c`, `src/quick_sort.c`, `src/bubble_sort.c`, `src/selection_sort.c` and `src/heap_sort.c`.

Exercise 2

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

Once implemented the sorting algorithms, in `src/main.c` I tested their performances by measuring the execution time for several input sizes n. We expect the following asymptotic complexities:

- `INSERTION SORT` is $O(n^2)$ (worst-case) and $\Omega(n)$ (best-case),
- `QUICK SORT` is $\Theta(n \log n)$ (average-case and best-case) and $\Theta(n^2)$ (worst-case),
- `BUBBLE SORT` is $\Theta(n^2)$,
- `SELECTION SORT` is $\Theta(n^2)$,
- `HEAP SORT` is $O(n \log n)$.

Figure 1 reports the results of the code. As expected, `HEAP SORT` and `QUICKSORT` are the best solutions, while `BUBBLE SORT` represents the less efficient algorithm. However, we see something unexpected; `SELECTION SORT` looks to perform better than `INSERTION SORT`, that looks to behave in case of a random input as a $\Theta(n^2)$.

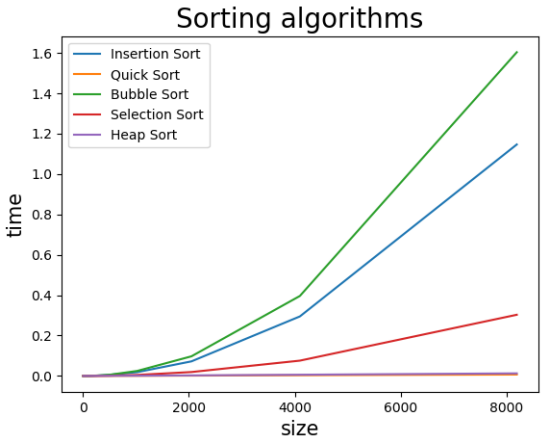

Figure 1: Benchmark of all the implemented sorting algorithms.

In Figure 2 we compare the two best algorithms, `HEAP SORT` and `QUICKSORT`, by considering a wider range of input sizes.

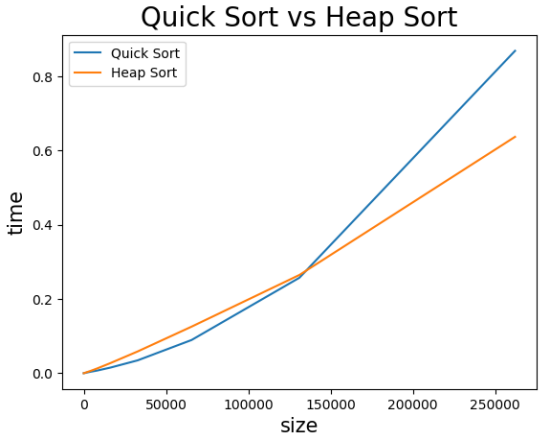

Figure 2: Benchmark of `HEAP SORT` and `QUICKSORT` with an input size that reaches $n = 2^{18}$.

It can be observed that starting from a size $n \simeq 1.5 \cdot 10^5$ `HEAP SORT` becomes faster than `QUICK SORT`.
Figure 3 and 4 report respectively the behaviour of `QUICK SORT` and `INSERTION SORT` for different scenarios.

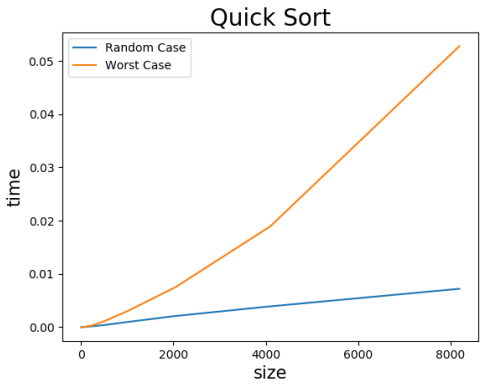

Figure 3: Benchmark of `INSERTION SORT`

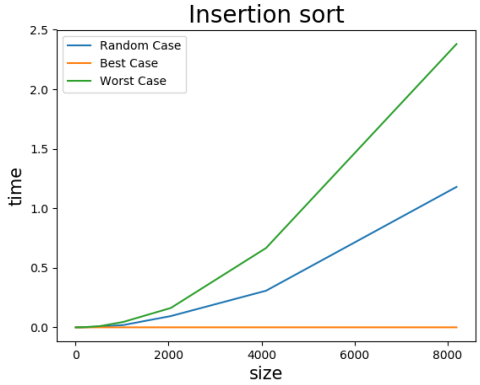

Figure 4: Benchmark of `INSERTION SORT`.

The curves show what expected from the theoretical results. Figure 3 shows that for `QUICK SORT` the worst-case looks to have a quadratic behaviour, while the random-case is more fast, confirming a complexity that is $\Theta(n \log n)$. Figure 4 shows the result for `INSERTION SORT`; confirming the observation made before in Figure 1, the random case seems to be closer to the worst-case behaviour, which shows a quadratic shape, rather than the best-case that results to increase linearly.

Exercise 3

Argue about the following statement and answer the questions:

(a) `HEAP SORT` **on a array** A **whose length is** n **takes time** $O(n)$. Since `HEAP SORT` complexity is given by the complexity of one single call of `BUILD HEAP` (which is $\Theta(n)$) and n calls of `EXTRACT MIN` (which is $O(\log i)$, where i is relative number of nodes in the heap) the overall cost of `HEAP SORT` is $O(n \log n)$. This higher bound represent a greater one respect to $O(n)$; thus in general `HEAP SORT` on a array A of length n doesn't take time $O(n)$.
However, in a specific case in which the heap is built in such a way that `EXTRACT MIN` costs $\Theta(1)$, then the overall cost is $\Theta(n)$ and the statement holds.

(b) `HEAP SORT` **on a array** A **whose length is** n **takes time** $\Omega(n)$. As we explained in the point (a), we know that `HEAP SORT` in the best-case scenario has an overall cost of $\Theta(n)$. Then, since $T_{HS}(n) = \Theta(n)$ this is equivalent to say that $T_{HS}(n) = \Omega(n)$ and, at the same time, $T_{HS}(n) = O(n)$. Thus, if this lower bound holds for the best scenario will also hold for every possible case and we conclude that the statement is true.

(c) **What is the worst case complexity for** `HEAP SORT`**?**
The worst-case scenario in `HEAP SORT` is that one in which we do n calls of

`EXTRACT MIN` operation and for every iteration i the cost of this function is $\Theta(\log i)$, where i is the relatives number of nodes down the heap. Thus,

$$T_{HS}(n) = \Theta(n) + \sum_{i=1}^{n} \Theta(\log i) = \Theta(n) + \Theta(n \log n) = \Theta(n \log n).$$

(d) **`QUICK SORT` on a array A whose length is n takes time $O(n^3)$.** On an array A of length n `QUICK SORT` has on average a time performance $T_{QS} = \Theta(n \log_2 n)$. This will mean that on average T_{QS} is both $O(n \log_2 n)$ and $\Omega(n \log_2 n)$. Since $O(n \log_2 n) \subset O(n^3)$ we can say that on average `QUICK SORT` takes time $O(n^3)$. However, using such higher bound it cannot be useful during performance analysis since we have found, in this case $n \log_2 n$, a cheaper function that can be a bound to our complexity function. The same reasoning can be applied to the worst-case scenario which has complexity equal to $\Theta(n^2)$.

(e) **What is the complexity of `QUICK SORT`?**
As already stated in the previous point the complexity of `QUICK SORT` is on average and in the optimal case $\Theta(n \log n)$, while for the worst-case is $\Theta(n^2)$.

(f) **`BUBBLE SORT` on a array A whose length is n takes time $\Omega(n)$.** Since `BUBBLE SORT` involves two loops over the length n of the array A, for any scenario the complexity of `BUBBLE SORT` is $\Theta(n^2)$. Automatically, `BUBBLE SORT` takes time $\Omega(n^2)$ and because $\Omega(n^2) \subset \Omega(n)$ we can conclude that the statement is formally true. However, we know that there exists a lower bound, in this case n^2, higher than n and so this sentence is not useful in a performance analysis.

(g) **What is the complexity of `BUBBLE SORT`?**
As said before, `BUBBLE SORT` involves two loops over the length n of the array A, then for any scenario the complexity of `BUBBLE SORT` is $\Theta(n^2)$.

Exercise 4

Solve the following recursive equation:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 32 \\ 3T(\frac{n}{4}) + \Theta(n^{3/2}) & \text{otherwise} \end{cases}$$

Let's use the above the relation in order to build a recursion tree. each node represents the cost of a single sub-problem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion. For convenience, we assume that n is an

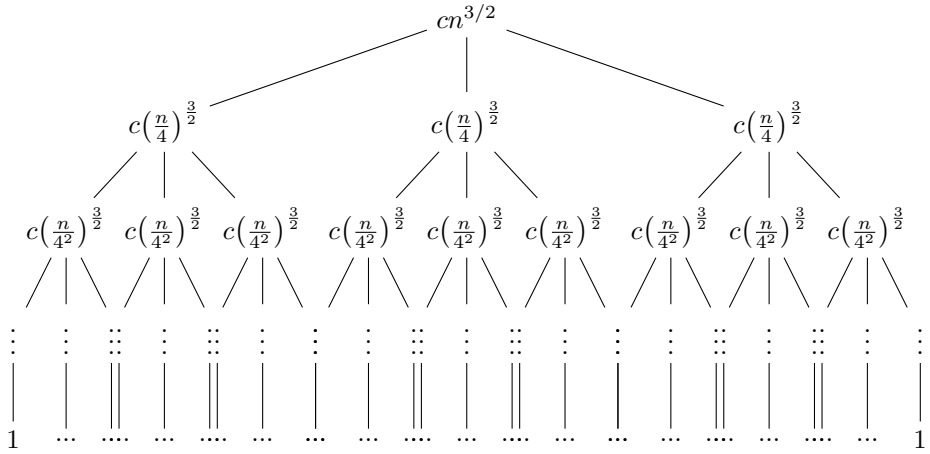

Figure 5: Recursion tree

exact power of 4, so that all sub-problem sizes are integers.

Given the tree at Figure 5, I add up the costs over all levels to determine the cost for the entire tree:

$$\begin{aligned}
T(n) &= cn^{3/2} + \frac{3}{4^{3/2}}cn^{3/2} + ... + \left(\frac{3}{4^{3/2}}\right)^{\log_4(n/32)-1} cn^{3/2} + \Theta(n^{\log_4 3}), \\
&= cn^{3/2} + \frac{3}{8}cn^{3/2} + ... + \left(\frac{3}{8}\right)^{\log_4(n/32)-1} cn^{3/2} + \Theta(n^{\log_4 3}), \\
&= cn^{3/2} \sum_{i=0}^{\log_4(n/32)-1} \left(\frac{3}{8}\right)^i + \Theta(n^{\log_4 3}),
\end{aligned}$$

$$\begin{aligned}
&= cn^{3/2} \sum_{i=0}^{\log_4(n/32)-1} \left(\frac{3}{8}\right)^i + \Theta(n^{\log_4 3}), \\
&\leq cn^{3/2} \sum_{i=0}^{+\infty} \left(\frac{3}{8}\right)^i + \Theta(n^{\log_4 3}), \\
&= \frac{8}{5}cn^{3/2} + \Theta(n^{\log_4 3}) \quad \in O(n^{3/2}).
\end{aligned}$$

Thus, $T(n) \in O(n^{3/2})$.