Supplementary exercise

Remarks: In all algorithm, always explain how and why they work. ALWAYS, analyze
the complexity of your algorithms. In all algorithms, always try to get the fastest possible.
A correct algorithm with slow running time may not get full credit. The term $\Omega(f(n))$
means: of order at least $f(n)$. The term $\Theta(f(n))$ means exactly of order $f(n)$. For example,
$\Theta(n)$ is both $\Omega(n)$ and $O(n)$.

1. Question 1: In this question we will show that with less than $n - 1$ comparison, its
impossible to find the maximum. An element $a$ in an array $A$ is called a not-winner,
if $a$ lost at least one comparison. Consider an algorithm to find the maximum.

   • After the algorithm made $i$ comparisons, what is the maximum possible number
   of non-winners?
   Answer: Every comparison gives one new no-winner (well, at most one. If you
   compare two non-winners we get no new non-winners). So, after $i$ comparisons
   we have at most $i$ non-winners.

   • Show that any algorithm requires at least $n - 1$ comparisons to find the maximum
   Answer: In any algorithm, after $n - 2$ comparisons or less, we have at most $n - 2$
   non-winners. So we have (at least) 2 elements that never lost a comparison. So
   its impossible to determine the winner among them.

2. Question 2: In this question we deal with finding both the maximum and the second
largest element. Consider first the following algorithm to find the maximum:

   (a) If the array has one element, this is the maximum.

   (b) Otherwise, split the remaining elements to pairs. If the number of elements is
   odd, one element is left alone

   (c) Compare all pairs and take the maximum of each (with the element that was
   alone)

   (d) form a new array with those winners and go to (a)

   • Show that no matter what $n$ is, this algorithm performs $n - 1$ comparisons. (Hint:
   how many non-winners do we have?).
   Answer: We start with 0 non-winners. We must get to $n - 1$ non-winners.
   Every comparison produces exactly one new non-winner. This is because we
   only compare two elements, if they won all the time so far. So the number of
   comparison exactly equals the number of non-winners (because every comparison
   creates one non-winner, and any non-winner was created by some comparison.)
   Thus, we have $n - 1$ comparisons because in the end we have $n - 1$ non-winners.

   • Who among the elements could be the second largest element?
   Answer: Create a tournament tree as follows. For every two elements $x$ and $y$
competing, create a father, labeled by $x$ if $x > y$ and $y$ otherwise. See the next picture.

As one can see in the picture, the second largest element is not necessarily the one that got to the final round. But it MUST be an element that competed against the maximum (otherwise, it would get to the final round, and anyway compete with the maximum).

- Show that the maximum and the second largest elements can be found using no more than $n + \lceil \log n \rceil$ comparisons.

**Answer:** How many elements competed against the maximum? Well, it’s exactly the height of the tree, which is no more than $\lceil \log n \rceil$. Finding the maximum of them can be done with additional $\log n$ comparisons. This plus the $n - 1$ comparisons to find the minimum, gives the result.

3. **Question 3:** Rate the following functions by increasing $O$ order:

$$n^{\log n}, 2^{\log n}, (\log n)^{\log n}, n^{10}, n^{9.\log^{10} n}, 19, (1.00001)^{\sqrt{n}}, (\log \log n)^{20}, n^{1/3}\log n, \frac{\sqrt{n}}{\log^2 n}.$$  

**Answer:** The first one is of course 19. $19 = O(1)$ that is, a constant. The second is $(\log \log n)^{20}$. For example: take logs from $(\log \log n)^{20}$ and $\log^{10} n$: $\log(\log \log n) \ll \log(\log^{10} n) = O(\log \log n)$. Next comes $\log^{10} n$. For example, $\log(\log^{10} n) \ll \log(2\sqrt{n}) = \ldots$
\(O(\sqrt{n})\). Next comes \(2\sqrt{\log n}\). Compare this for example to \(\frac{\sqrt{n}}{\log n}\). Taking logs from both sides gives \(\sqrt{\log n} << O(\log n) - O(\log \log n) = O(\log n)\). Next comes \(\frac{\sqrt{n}}{\log^2 n}\) (while, for example \((\log n)^{\log n} = n^{\log \log n}\) (take logs from both sides to verify this equality) which is much larger than \(n\)). Next comes \(n^{1/3} \cdot \log n\) which is of smaller order than \(\frac{\sqrt{n}}{\log n}\) (the logs do not matter). Then comes \(\frac{\sqrt{n}}{\log n}\). Next \(n^{9/10}\cdot \log^{10} n\), and then \(n^{10}\) (divide \(n^{10}\) by \(n^{9}\cdot \log^{10} n\). Verify that this tends to \(\infty\) when \(n\) tends to \(\infty\).) Then \(((\log n)^{\log n} = n^{\log \log n}\). This is more than \(n^{10}\). Indeed taking log of \(n^{10}\) you get \(O(\log n)\). Taking log of \(n^{\log \log n}\) you get \(\log n \cdot \log \log n\). Since the log \(\log n\) is not constant, we have the claim. Now, for the last one: \(n^{\log \log n} << (1.0001)^{\sqrt{n}}\). Indeed, taking logs from both sides we get that \(\log n \cdot \log \log n << \sqrt{n}\).

4. **Question 4:** Write a sorting algorithm that works by each iteration choosing the maximum among the yet unsorted elements, and puts it in the end. What is the best worst and average running time of the procedure?

**Answer:**

**Algorithm 1 Selection-Sort**

(a) For \(i = 1\) to \(n - 1\)

i. \(k \leftarrow 1\) **\(k\) keeps the index of maximum in current iteration**

ii. For \(j = 2\) to \(n - i + 1\) do

    • If \(A[j] > M\) then \(k \leftarrow j\)

iii. Swap \(A[k]\) and \(A[n - i + 1]\)

The complexity is \(n^2\) (remember: complexity is the worst case). We have \((n - 1) + (n - 2) + \ldots + 1 = O(n^2)\). In fact all the inputs are equality bad (in all inputs the time complexity is \(n^2\)) so this is the best and average as well.

5. **Question 5:** In insertion sort, say that when we look for the location of \(A[i]\) inside \(A[1, ..., i - 1]\) we use binary search. Does this improve the running time?