Algorithms & Data Structures 2024/25

Practical Week 9

I realise that model answers tend to be easily available. Please however try to come up with your own solutions. If you do get stuck then ask the demonstrators for help.

0. Finish anything that may be left over from last week. (This is particularly important for any and all exercises on asymptotic notation.)
1. For each of the following functions $f(n)$ and $g(n)$, determine which of the following statements applies (and justify your answer): $f(n) = o(g(n))$, $f(n) = \Theta(g(n))$, or $f(n) = \omega(g(n))$.
   (a) $f(n) = 1000n^2$, $g(n) = n^3$.
   (b) $f(n) = 2.1^n$, $g(n) = 2^n$
   (c) $f(n) = 5n$, $g(n) = \log_2(3^n)$
   (d) $f(n) = n^2$, $g(n) = \frac{n^2}{\log_2 n}$

Solution:

(a) $1000n^2 = o(n^3)$ because $\frac{1000n^2}{n^3} = \frac{1000}{n} \overset{n\to\infty}{\longrightarrow} 0$

(b) $2.1^n = \omega(2^n)$ because $2^n = o(2.1^n)$, which follows from $\frac{2^n}{2.1^n} = \left(\frac{2}{2.1}\right)^n \overset{n\to\infty}{\longrightarrow} 0$

(c) $5n = \Theta(\log_2(3^n)$. Note that $\log_2(3^n) = n \log_2 3 \approx 1.58n$.
We have $5n = O(\log_2(3^n))$ because $5n \leq 5n \log_2 3$ for $n \geq 1$, so $C = 5$ and $k = 1$ are a pair of witnesses.
We have $\log_2(3^n) = O(5n)$ because $\log_2(3^n) \approx 1.58n \leq 5n$ for $n \geq 1$, so $C = 1$ and $k = 1$ are a pair of witnesses.

(d) $n^2 = \omega(\frac{n^2}{\log_2 n})$ because $\frac{n^2}{\log_2 n} = o(n^2)$, which holds because

$$\frac{\frac{n^2}{\log_2 n}}{n^2} = \frac{1}{\log_2 n} \overset{n\to\infty}{\longrightarrow} 0$$

2. For each of the following statements (where f, g, h are non-negative functions), determine whether it is true or false, and give a proof or a counterexample.
   (a) If $f(n) = \omega(g(n))$ and $g(n) = O(h(n))$, then $f(n) = \omega(h(n))$.
   (b) If $f(n) = o(g(n))$, then $f(n) = O(g(n))$.
   (c) If $f(n) = \omega(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \omega(h(n))$.
   (d) If $f(n) = O(g(n))$ but not $g(n) = O(f(n))$, then $f(n) = o(g(n))$.

Solution:

(a) Statement is false. Counterexample: $f(n) = n$, $g(n) = \sqrt{n}$, $h(n) = n^2$.

(b) Statement is true. If $f(n) = o(g(n))$, then $\forall C > 0 \exists k > 0 : C \cdot f(n) < g(n)$ for all $n \geq k$. Pick any $C > 0$. Then there is $k > 0$ so that $f(n) < \frac{1}{C} g(n)$ for all $n \geq k$. This means that $1/C$ and k are witnesses for $f(n) = O(g(n))$.

(c) Statement is true. $f(n) = \omega(g(n))$ means that $g(n) = o(f(n))$ and hence $\frac{g(n)}{f(n)} \overset{n\to\infty}{\longrightarrow} 0$. Furthermore, $g(n) = \Theta(h(n))$ implies $g(n) = \Omega(h(n))$ and hence $h(n) = O(g(n))$. So there are $C, k > 0$ with $h(n) \leq C \cdot g(n)$ for $n \geq k$. Then $\frac{h(n)}{f(n)} \leq \frac{C \cdot g(n)}{f(n)}$ for $n \geq k$ and hence:

$$\frac{h(n)}{f(n)} \leq \frac{C \cdot g(n)}{f(n)} = C \cdot \frac{g(n)}{f(n)} \overset{n\to\infty}{\longrightarrow} 0\,.$$

So $h(n) = o(f(n))$, and hence $f(n) = \omega(h(n))$.

(d) Statement is false. Let $f(n)$ be n for even n and 0 for odd n, and let $g(n) = n$. Then $f(n) = O(g(n))$. Furthermore, $g(n)$ is not $O(f(n))$ because $g(n) > C \cdot f(n)$ for all odd n, no matter what the value of C ($C > 0$) is. Furthermore, $f(n)$ is not $o(g(n))$ because $\frac{f(n)}{g(n)} = 1$ for all even n, so $\frac{f(n)}{g(n)}$ does not converge to 0 for $n \to \infty$.

3. Explain how one might modify the Merge function (used within MergeSort) to achieve a constant $O(1)$ number of comparisons in certain circumstances. What are those circumstances?

Solution:

The two sorted lists fed into the Merge function are each sorted, of course, but if their concatenation is sorted as well (that is, anything in the left one is smaller than everything in the right one, or vice versa) then no merging would be necessary—the two lists can simply be concatenated. Two comparisons at the very beginning of the Merge function could determine if this was the case, by comparing the rightmost element of the left list with the leftmost element of the right list (and the rightmost element of the right list with the leftmost element of the left list).

4. Suppose you are given an array $A[]$ of length n containing numbers. Suppose further that you are given access to a random number generator `random(k)` that returns a random integer from $\{0, \ldots, k-1\}$, for any parameter k. How do you generate a random permutation of the elements in $A[]$ in linear time?

Solution:

The permutation will be created in place.

For position 1 of the output permutation we pick any one element from $A[]$ at random and swap it into position 1.

For position 2 of the output permutation we pick any one element from $A[]$ except $A[1]$ at random and swap it into position 2.

For position 3 of the output permutation we pick any one element from $A[]$ except $A[1]$ and $A[2]$ at random and swap it into position 3.

For position i of the output permutation we pick any one element from $A[]$ except $A[1, \ldots, i-1]$ at random and swap it into position i.

Permute $(a_1, \ldots, a_n \in \mathbb{R}, n \geq 2)$

1: **for** $i = 1$ to $n - 1$ **do**
2: swap $A[i]$ and $A[i + \text{random}(n - i + 1)]$
3: **end for**

5. In terms of real-world performance (as opposed to asymptotic analysis), it may be beneficial for recursive sorting algorithms to not recurse all the way down to problems of size 1 but stop recursion earlier, replacing those recursive calls with (ideally) fast algorithms to sort those small subproblems. This is to avoid excessive amounts of expensive stack operations. For example, an algorithm that can sort any sequence of five numbers using as few comparisons as possible may be a candidate for replacing recursive calls on five elements.

   Design and implement (in a language of your choosing) an algorithm that sorts any sequence of five elements using at most seven comparisons. The algorithm should be "in place", that is, you shouldn't be using auxiliary arrays. Test on all $5! = 120$ permutations of the input $[1, 2, 3, 4, 5]$ (you may find it useful to write a function that automatically generates, and feeds into your sorting algorithm, those permutations).

   Solution:

   Suppose the elements are named a, b, c, d, e. The algorithm will first determine the relative order of the first four elements a, b, c, d:

   (a) Compare a with b, and c with d; 2 comparisons.

   (b) Compare the larger values from the previous two comparisons. Assuming $a > b$ and $c > d$, we would compare a with c. The larger of these two (assume it's a) would certainly be the largest of a, b, c, d. This would give us a partial ordering $a > c > d$ and $a > b$. 1 comparison.

   We assume that the partial ordering is as indicated above, i.e., $a > c > d$ and $a > b$.

   (a) Compare c with e. (In general, compare e with whatever happens to be the middle element of the partial ordering.) 1 comparison.

   (b) Two cases: if $e > c$, then compare e with a (the largest), otherwise compare e with d (the smallest of the partial ordering). 1 comparison.

   We now have determined e's place with respect to a, c, d, and it remains to find b's place in a sorted list of three elements (we have already established that $b < a$). This takes at most two more comparisons.

6. **Solving this question requires familiarity with basic probability theory. Please feel free to skip the calculation of the expected number of comparisons if you feel that you are not sufficiently familiar with probability theory.**

   Consider the following sorting algorithm.

   MonkeySort ($a_1, \ldots, a_n \in \mathbb{R}$)

   ```
   1: while (a_1, a_2, ..., a_n) is not sorted do
   2:     randomly permute a_1, a_2, ..., a_n
   3: end while
   ```

   In the literature this algorithm is also variously known as "StupidSort", "BogoSort", or "Slow-Sort".

   Provide more detailed code, in particular for testing for sortedness.

   This being a randomised algorithm, we need to express the number of comparisons it makes as a random variable. If you know your way around very basic probability theory: what's that random variable's expectation, asymptotically?

   You may assume that the input elements are pairwise distinct.

Solution:

Boolean **isSorted** $(a_1, \ldots, a_n \in \mathbb{R})$

```
1: for i = 1 to n − 1 do
2:     if a_i > a_{i+1} then
3:         return False
4:     end if
5: end for
6: return True
```

MonkeySort $(a_1, \ldots, a_n \in \mathbb{R})$

```
1: while not isSorted (a_1, a_2, ..., a_n) do
2:     for i = 1 to n − 1 do
3:         swap A[i] and A[i + random(n − i + 1)]
4:     end for
5: end while
```

The analysis follows a two-stage approach: (a) how many random permutations do we expectedly have to generate before one is found to be in sorted order, and (b) how long does it expectedly take to test one such random permutation for sortedness.

This argument can be made more precise.

(a) Only one permutation of $a_1, \ldots, a_n$ is in sorted order. There are a total of $n!$ of them, so each trial succeeds with a probability of $p = 1/(n!)$ – this is like tossing a biased coin that comes up heads with probability $1/n!$. When we ask for the number of trials until the first success, we are in fact looking at the geometric distribution. For a sequence of independent Bernoulli trials with success probability p, this expected value is known to be $1/p$. With $p = 1/n!$ we obtain

$$\frac{1}{1/n!} = n!$$

(b) Consider any random permutation π of $a_1, \ldots, a_n$. With pretty much the same argument as above, but of course different probabilities, in expectation we will need to inspect a constant number of pairs of adjacent elements before we find the first one for which $a_i > a_{i+1}$. Consider that already for (a_1, a_2), we have that a_1 is from the larger half with probability of $1/2$, and a_2 from the lower half with probability $1/2$, giving a probability of at least $1/4$ (other things can happen as well) for a mismatched pair right at the beginning. In fact, the probability that $a_1 > a_2$ can be shown to be $\frac{1}{2}$.

Wrapping up, there will be expected $\Theta(n!)$ many comparisons in expectation: We need to go through $n!$ permutations in expectations, and perform in expectation a constant number of comparisons for each of them.