Algorithms homework related to exam 1, Summer 2021

Hw1: Due Tuesday, June 1, at 11:59pm NYC time

1. Suppose that $f(n) = \Theta(g(n))$. Assume that both are increasing functions.

   (a) Must it be true that $\log f(n) = \Theta(\log g(n))$? Prove or disprove.

   (b) Must it be true that $2^{f(n)} = \Theta(2^{g(n)})$? Prove or disprove.

2. (a) Suppose you have a function of two variables, n and k. What would it mean, mathematically, to say that this function is $O(n + k)$?

   (b) Let $f(n) = O(n)$ and $g(n) = O(n)$. Let c be a positive constant.
   Prove or disprove that $f(n) + c \cdot g(k) = O(n + k)$.

3. Let $f(n) = \sum_{y=1}^{n}(n^6 \cdot y^{23})$.

   Find a simple $g(n)$ such that $f(n) = \Theta(g(n))$, by proving that $f(n) = O(g(n))$, and that $f(n) = \Omega(g(n))$.

   Don't use induction / substitution, or calculus, or any fancy formulas.
   Just exaggerate and simplify for big-O, then underestimate and simplify for Ω.

Hw2: Due Monday, June 7, at 11:59pm

1. $S(n) = 3S(\frac{n}{3}) + b$.

   (a) Use a recursion tree to show that $S(n) = \Theta(g(n))$, for some function $g(n)$.

   (b) Use induction to prove that $S(n) = \Omega(g(n))$.

2. $T(n) = T(n-1) + \log n$. Assume that there is an appropriate base case so that you don't get an undefined function.

   (a) Draw a recursion tree for $T(n)$. Identify its depth and the amount of work per level. Use "exaggerate and simplify" to get an upper bound for $T(n)$, and then underestimate and simplify to get a matching lower bound. (Always using Theta notation). Your bound should be in a form that is easily recognizable (and easy to compare to simple functions like linear, quadratic, etc).

   (b) Derive the same upper bound for $T(n)$, by induction.

3. Solve $T(n) = T(\sqrt{n}) + 1$, using a change of variables: $n = 2^m$.

   To warm up, consider the following question. (Don't submit an answer for the warmup)
   Think of an abstract recursive algorithm that operates on a $n \times n$ matrix: It does non-recursive work proportional to the number of elements in the matrix (say just n^2 work), and then recurses on each of the four quadrants. Assume that quadrants are nicely defined.
   This could be expressed as $S(n) = 4S(\frac{n}{2}) + n^2$, meaning the parameter used in S to describe the problem is the matrix side length.
   But suppose that we hadn't thought of doing that, and instead we expressed the time complexity as $T(n^2) = 4T(\frac{n^2}{4}) + n^2$. Here, the parameter used for the problem size is the total number of elements. In this case, because this parameter in T is a function of n (other than just n itself), we don't really have a direct method to deal with it. But we could set $m = n^2$, so $T(m) = 4T(\frac{m}{4}) + m$. What's the solution in each case (in terms of n)? It should be the same.
   Besides that, the lesson here is that there can be multiple ways to quantify a problem size and recurse accordingly, e.g. $S(n)$ vs $T(n^2)$. You'll need to consider this after you apply the change of variables as suggested above for the actual homework problem.

4. Use the master method for the following (state case or explain what dominates, and state the answer), or explain why it's not possible.

   (a) $T(n) = 10 \cdot T(\frac{n}{3}) + \Theta(n^2 \log^5 n)$.

   (b) $T(n) = 256 \cdot T(\frac{n}{4}) + \Theta(n^4 \log^4 n)$.

   (c) $T(n) = T(\frac{19n}{72}) + \Theta(n^2)$.

   (d) $T(n) = n \cdot T(\frac{n}{2}) + n^{\log_2 n}$.

   (e) $T(n) = 16 \cdot T(\frac{n}{4}) + n^2$.

   (f) $T(n) = 3 \cdot T(\frac{n}{2}) + n^2$.

   (g) $T(n) = T(\frac{n}{n-1}) + 1$.

   (h) $T(n) = 4 \cdot T(\frac{n}{16}) + \sqrt{n}$.

Hw3: Due Saturday, June 12, at 11:59pm

1. You work at a hospital's emergency room. Patients come in and you evaluate the severity of their injury by assigning a **real** number from 1 to 100. Whoever has the highest number (and in case of ties, whoever arrived first) sees the doctors first. When a doctor is available, no time should be wasted, so if possible you want to *instantly* send in the next patient.
   (a) How will you handle the data, to automate this process? How fast can you find the next patient to be seen?
   (b) State how much time it will take you to process new patients, or to reorganize after a patient is sent through to the doctors.
   (c) Explain what to do if a patient gets worse while they're waiting, meaning you decide that their severity is now greater.
   (d) Explain what to do if the evaluation numbers are integers, and how this affects time complexity for the questions above.

2. Let [100, 50, 40, 45, 3] be a max-heap that was constructed using the *forward* method (scanning the input left-to-right). Let x be the last element that was inserted.
   For each element in the heap, explain why it might be x, or why it cannot be.

3. For an array $A = [a_1, a_2, a_3, a_4]$ of distinct numbers, there are two main ways to build a heap, as described in class. In parts (a) and (b) of this problem you must show what comparisons each method will make, in the form of a binary decision tree. Each leaf should contain output in the form of some permutation of the input subscripts in A (e.g., if you write 3124 it means that after building the heap we have $A = [a_3, a_1, a_2, a_4]$.

   (a) Do the above (draw the decision tree) for the forward method.

   (b) Do the above (draw the decision tree) for the reverse method.

   (c) Describe your own heap-building algorithm that specifically handles inputs of size 4, and draw the corresponding decision tree that uses fewer decisions in the worst-case compared to the methods in (a) and (b). Your algorithm should be described in English, not pseudocode.

4. Suppose that we have a set of n distinct numbers that we wish to sort. In class we see that every comparison-sort algorithm must take $\Omega(n \log n)$ time for at least one input permutation, but for all we know there might be an algorithm that is really fast for all other inputs. So now we will show that that's impossible.

   Show that **every** comparison-sort algorithm takes $\Omega(n \log n)$ time, not just in the worst case, but for almost all inputs. Specifically show that the best conceivable algorithm can't even handle $\frac{1}{2^n}$ of all possible inputs in *little-o*$(n \log n)$ time.

   Here $f(n) =$ *little-o*$(n \log n)$ means $f(n) = O(n \log n)$ and $f(n) \neq \Omega(n \log n)$.

   Your answer should not just contain math. You should have an explanation that shows us that you understand exactly what's going on.