CS 620 Fall 2010 at ISU, Model Solutions for Homework 1

Prepared by assistant professor Jeff Kinne on September 2, 2010.

Problem 1

This problem deals with big-O asymptotic notation, proving useful rules and providing practice.

Part a Let f and g be two functions that are polynomially-bounded asymptotically. We wish to show that if we define h_1 and h_2 such that $h_1(n) = f(n) \cdot g(n)$ and $h_2(n) = f(g(n))$ then both h_1 and h_2 are polynomially-bounded asymptotically as well.

Let us argue informally first. By assumption we know that $f(n) = O(n^{c_f})$ and $g(n) = O(n^{c_g})$ for some constants c_f and c_g (this is what I mean by "polynomially-bounded asymptotically"). Then we have that $h_1(n) = f(n) \cdot g(n) = O(n^{c_f}) \cdot O(n^{c_g}) = O(n^{c_f + c_g})$ and $h_2(n) = f(g(n)) = O((O(n^{c_g}))^{c_f}) = O(n^{c_g \cdot c_f})$. Then both h_1 and h_2 are big-O polynomial of slightly higher degree than f and g.

Notice we have used the "rule" that $O(n^{c_f}) \cdot O(n^{c_g}) = O(n^{c_f + c_g})$ in the calculation for h_1 and have used similar reasoning in the calculation for h_2. We need to justify that these "rules" with big-O notation are correct. I see two ways to do this. One way would be to show that the "rules" we wanted to apply really are valid. This could be done, and is not too difficult. The other way is to apply the definitions more precisely in the calculations for h_1 and h_2, and this is what I will show now.

Applying the definitions more precisely, we have that there exist constants k_f, k_g, n_f, and n_g such that

$$f(n) \leq k_f \cdot n^{c_f}$$

for all $n \geq n_f$ and

$$g(n) \leq k_g \cdot n^{c_g}$$

for all $n \geq n_g$. Then

$$h_1(n) \leq (k_f k_g) \cdot n^{c_f + c_g}$$

for all n greater than the *maximum* of n_f and n_g. So for $n \geq max(n_f, n_g)$, h is at most a constant times $n^{c_f + c_g}$; in other words $h_1(n) = O(n^{c_f + c_g})$. Also,

$$h_2(n) \leq k_f(k_g \cdot n^{c_g})^{c_f} = (k_f k_g^{c_f}) \cdot n^{c_g \cdot c_f}$$

for all n greater than the maximum of n_g and $(n_f/k_g)^{1/c_g}$. (The fact that n should be at least $max(n_g, (n_f/k_g)^{1/c_g})$ is slightly tricky – do you see why?) This means that $h_2(n) = O(n^{c_g \cdot c_f})$.

Part b We now want to show that if f and g are exponentially-bounded asymptotically ($2^{O(n^d)}$ for some constant d) then their product is exponentially-bounded asymptotically.

Again let us first argue informally. By assumption, we know that there exist constants c_f and c_g such that $f(n) = 2^{O(n^{c_f})}$ and $g(n) = 2^{O(n^{c_g})}$. If we let h_1 again be defined as their product, then $h_1(n) = 2^{O(n^{c_f})} \cdot 2^{O(n^{c_g})} = 2^{O(n^{c_f} + n^{c_g})}$ and thus $h_1(n) = 2^{O(n^d)}$ for $d = max(c_f, c_g)$.

As with part (a), we have implicitly used some "rules" of working with big-O, and we should either prove those rules or argue more precisely. We again argue using the definitions more precisely. We have that there exist constants k_f, k_g, n_f, and n_g such that $f(n) \leq 2^{k_f n^{c_f}}$ for all $n \geq n_f$ and $g(n) \leq 2^{k_g n^{c_g}}$ for all $n \geq n_g$. Then $h_1(n) \leq 2^{k_f n^{c_f} + k_g n^{c_g}}$ for all $n \geq max(n_f, n_g)$. Letting $k = max(k_f, k_g)$ and $c = max(c_f, c_g)$ we have that $h_1(n) \leq 2^{2kn^c}$ for all $n \geq max(n_f, n_g)$ meaning that $h_1(n) = 2^{O(n^c)}$.

Part (b) also asks if h_2 defined by $h_2(n) = f(g(n))$ is exponentially bounded. The answer is that we do not know; let me explain why. If we applied similar logic as above, we can show that $h_2(n) = 2^{2^{O(n^c)}}$ for some constant c. We have said that "exponentially bounded" means the function should be $2^{O(n^d)}$ for some constant d, and a function of the form $2^{2^{n^c}}$ is definitely *much bigger than* $2^{O(n^d)}$. We call functions of the form $2^{2^{n^c}}$ "doubly exponential". So we have no reason to believe that $h_2(n)$ is at most $2^{O(n^d)}$ for some constant d. But, note that it could be so. We have only said that $f(n) = 2^{O(n^{c_f})}$ and $g(n) = 2^{O(n^{c_g})}$. These are upper bounds on f and g but we have not stated any lower bounds. It could be that $f(n) = n$ and $g(n) = n^2$. These are certainly upper-bounded by $2^{O(n^d)}$ for any constant d, and in this case h_1 would be also.

Note on what to do on future homeworks Most of the time in this course we will not be so detailed in our application of big-O notation. We will generally be interested in the question – "is the running time polynomial or not?" By part (a), we know that if we have a program/algorithm/Turing Machine that has a polynomial number of steps, and each of those steps could be a call to a subroutine that itself runs in polynomial time, then the overall program still runs in polynomial time. So we will just generally use big-O in this way – each of the parts uses polynomial time, and there are a polynomial number of them, so altogether we run in polynomial time.

Problem 2

This problem provides some practice in giving big-O estimates of the running time of algorithms.

Part a In this part of the problem we give pseudocode for a brute-force algorithm to compute a non-trivial factor of a number that is given in binary.

1. **Input:** x, an n-bit binary string which we interpret as a non-negative integer.
2. If $x = 0$ return "0", and if $x = 1$ return "1".
3. If $x = 2$ return "prime".
4. Otherwise, for $i = 2$ up to $x - 1$ do the following. If i evenly divides x, then return i.
5. If no factors are found in the for loop, return "prime".

A simple optimization would be to search only up to $\lfloor \sqrt{x} \rfloor$, but this optimization would make little difference in the running time analysis that we perform in the next part.

Part b In this part, we give a big-O running time analysis of the algorithm of part (a) assuming that arithmetic operations on n-bit integers can be done in $O(n^2)$ time. The most important part of the running time is to give a bound on how many iterations the for loop can run for. If x is an n-bit string, then the value of x is at most $2^n - 1$ (this would correspond to the all 1's string). So we see right away that the algorithm runs in time that is exponential in the input length. Being slightly more precise, we can look at home much time is taken for each iteration of the for loop. The iteration consists of dividing x by i and checking if the remainder is 0. The division is on numbers that are at most n bits, which we have assumed can be done in $O(n^2)$ time. The remaining operations (checking if $x = 0, 1$, or 2) take constant time, and we have that the above algorithm runs in time $O(n^2 \cdot 2^n)$.

If we performed the optimization of only checking factors up to $\lfloor \sqrt{x} \rfloor$, we would get a running time of $O((n/2)^2 \cdot 2^{n/2}) = O(n^2 \cdot 2^{n/2})$. In fact the running time is $2^{\Theta(n)}$ because there are inputs on which $2^{\Omega(n)}$ time is needed (e.g., if $x = p \cdot q$ for primes p and q that are roughly equal).

Note for future use In the future, it would likely be enough to simply state that the for loop runs for at most 2^n iterations, and each iteration takes polynomial time, so that overall the algorithm runs in $n^{O(1)} \cdot 2^n$ time (at most exponential).

Problem 3

This problem gives practice in using induction proofs.

Part a We let A be any finite set, 2^A denote the power set of A (the set of all subsets), and let $|A|$ denote the cardinality of A. We want to show that $|2^A|$, the cardinality of the power set, is equal to $2^{|A|}$. We prove this by induction. In computer science, we usually deal with discrete finite objects and structures, so we will often use proofs by induction. In particular, when we need to show that something is true for all of a particular type of object, it is a good idea to try a proof by induction.

A proof by induction requires two parts: showing the claim holds for certain small "base cases" and then showing that if the claim holds for objects of a certain size then it must also hold for objects of one larger size. Here we use induction on the size of A.

For the base case, we let A be the smallest possible set – the empty set. Then $|A| = 0$, and the power set of A consists of the empty set (remember that the power set of a set always includes the empty set). So $|2^A| = 1 = 2^0 = 2^{|A|}$.

We now prove the inductive step. We assume the claim holds for all sets of size at most k and want to show it must then hold for sets of size $k+1$. Let A be a set of size $k+1$, and let a be some element of A. Because A is of size at least one, we know there must exist an element a of A. The power set of A can be broken into (i) sets that contain a, and (ii) sets that do not contain a. We can compute the size of 2^A by adding the number of (i) sets and the number of (ii) sets. There is a set in (i) for all possible subsets of $A - \{a\}$. $A - \{a\}$ is of size k, so by our induction hypothesis the number of possible subsets (the size of the power set) is equal to $2^{|A-\{a\}|} = 2^k$. Similarly, there is a set in (ii) for all possible subsets of $A - \{a\}$ as well, so the number of sets in (ii) is equal to 2^k. The size of 2^A is equal to the sum of the sizes of (i) and (ii), so equal to $2^k + 2^k = 2^{k+1} = 2^{|A|}$.

Part b In this part we prove how many binary strings there are that have exactly n bits. I will describe two ways to do this. First, we can use part (a) of this problem. A binary string x with n bits can be viewed as indicating a subset of $\{1, 2, 3, ..., n\}$ – by placing i in the set X if and only if the i-th bit of x is equal to 1. Thus the number of binary strings with exactly n bits is equal to the number of possible subsets of $\{1, 2, 3, ..., n\}$. By part (a), we know this is equal to 2^n.

We could prove this directly using the "product rule" of counting. Each bit of an n-bit string can take two possible choices, so the total number of possibilities is equal to $2 \cdot 2 \cdot ... \cdot 2$ where there are n two's in the product, so 2^n total.

Problem 4

In this problem we get some practice with two very common techniques we will use this semester. We prove that one problem is "as hard as" another problem. That is, if we can solve the one problem, then we could solve the other. This is called a "reduction", but means just what I said - we can solve the one problem using a solution to the other as a subroutine. We combine this "reduction" with the fact that we already know the one problem is undecidable to conclude that the other problem is also undecidable. We will write the proof as a "proof by contradiction".

We know that the halting problem is undecidable. The halting problem takes as input a pair (M, x) and asks whether the Turing Machine M halts in finite time when given x as the input. We want to show that the "all strings problem" is also undecidable. This problem takes as input the description of a Turing Machine M and asks if the Turing Machine computes the "all strings language", that is, does M halt in finite time and accept all possible input strings.

Suppose we could solve the all strings problem (by a Turing Machine that halts in finite time and outputs the correct answer for all possible inputs). We will arrive at a contradiction, namely that we could also solve the halting problem – which we know cannot be solved. Let A be the assumed algorithm that solves the all strings problem. We want to use A to also solve the halting problem. Let (M, x) be an input to the halting problem that we want to solve – we want to determine if M halts in finite time when given x as an input. Let M' be the following Turing Machine: it ignores its input, simulates $M(x)$, and outputs "yes/accept" if and only if $M(x)$ halts in finite time. Notice that M' halts in finite time and accepts all possible input strings if and only if $M(x)$ halts in finite time. Thus we can use our assumed algorithm A for solving the all strings problem – $A(M')$ gives the correct answer to the halting problem on input (M, x). Because we know the halting problem cannot be solved, then the assumed algorithm A cannot exist – the all strings problem is undecidable.

Note. Note that in the above, I have not given Turing Machine code, C/Python/Java code; I have not even given pseudocode for how to solve the halting problem if we had a solution to the all strings problem. But from the description I have given, we could easily produce the pseudocode or Turing Machine code (a tedious task, but easy). The level of detail given in the above argument is about what I normally expect form you in this course.