University of Virginia - cs3102: Theory of Computation Spring 2010

Problem Set 6 - Comments

Problem 1: Asymptotic Notation. The way we advocated using asymptotic notation to describe algorithm complexity in Classes 22 and 23 is different from how Sipser uses it in Section 7.1. Identify at least two ways in which Sispers use of the asymptotic operators differs from the way we advocated using them in class, and illustrate each with a specific example from Sipsers text.

Answer. Sipsers book sometimes uses the asymptotic notations as sets, but sometimes as if they produce actual numbers. For example, "Each scan uses $O(n)$ steps." (p. 251). This is certainly easier than saying, "A function from the input size n to the number of steps used in each scan would be a member of O(n).", but it is not strictly correct if we view the asymptotic operators are producing sets of functions.

Sipser uses equality (e.g., $f(n) = O(n^3)$) to show a function is a member of a $O(g(n))$ class. In class, we advocated using set membership (e.g., $f(n) \in O(n^3)$) to clearly distinguish between the case where a function is a member of a set and the case where two sets are equal (e.g., $O(n) \cup O(logn) = O(n)$).

Sipser uses O to analyze algorithms, when a tight Θ bound could be easily stated. For example, in the first example above, the number of steps is also known to be in $\Theta(n)$. It is not incorrect to say it is in $O(n)$, but provides much less information than providing the tight bound.

Problem 2: Asymptotic Propositions. For each sub-part below, state whether the statement is **true**, **false**, or **unknown** (where unknown means no one on Earth knows the answer, not just that you don't know it!). Include a brief but convincing argument supporting your answer.

a. $\Theta(n \log n) \subset O(n^2)$

Answer. True. Intuitively, $O(n^2)$ is the set of all functions that grow no faster than n^2, and $\Theta(n \log n)$ is the set of functions that grow as fast as $n \log n$. Since $n \log n$ grows slower than n^2, $\Theta(n \log n)$ is a proper subset of $O(n^2)$.

More formally, we will first show that $\Theta(n \log n) \subseteq O(n^2)$ and then that the inclusion is proper. Recall from the definition of Θ that $\Theta(n \log n) = \Omega(n \log n) \cap O(n \log n)$. So, $f(n) \in \Theta(n \log n)$ implies $f(n) \in O(n \log n)$.

By the definition of O this means that there exist positive numbers c, n_0 such that $\forall n \geq n_0, f(n) \leq cn \log n$. For all $n > 1$, $n \log n < n^2$. So, $f(n) \leq cn \log n > cn^2$ implies $f(n) \in O(n^2)$.

This proves $\Theta(n \log n) \subseteq O(n^2)$. To prove the inclusion is proper (that is, $\subset$), we need to identify some function that is in $O(n^2)$ but not in $\Theta(n \log n)$. One such function is $f(n) = n^2$. This is obviously in $O(n^2)$ (choose $c = 1, n_0 = 1$), but is not in $\Theta(n \log n)$.

b. $\exists f \in \mathcal{N} \to \mathcal{R}^+$ such that $\Omega(f) \cap O(f) = \emptyset$

Answer. False. The set $\Omega(f)$ consists of functions that grow at least as fast as f, so $f \in \Omega(f)$. The set $O(f)$ is consists of functions that grow no faster than f, so $f \in O(f)$. Thus, their intersection must always include at least f. Thus, $\Omega(f) \cap O(f) \neq \emptyset$.

c. set of languages that can be recognized by a Janus machine (as defined in Exam 2) in $O(n)$ steps $\subseteq$ set of languages that can be recognized by a TM in $O(n)$ steps

Answer. False. (This is the question as modified in the comments.) There are some languages that can be recognized by a Janus machine in $\Theta(n)$ steps but cannot be recognized by any TM in $O(n)$ steps. For example,

$$PAL = \{ww^R | w \in \Sigma^*\}$$

We can recognize this in $\Theta(n)$ steps with a Janus machine by having rule like,

$$\delta(q_c, a, a) \to (q_c, \$, R, \$, L)$$

for all symbols $a \in \Sigma$. So, on any matching symbol, the left head moves Right and the right head moves Left, writing \$. The machine accepts if the heads meet in the middle:

$$\delta(q_c, \$, \$) \to (q_{accept}, \$, R, \$, L)$$

Because of the quirky way we defined the Janus machine (with the right head starting on the first blank), we would need a rule for q_0 to move to the last input symbol:

$$\delta(q_0, a, \sqcup) \to (q_c, a, L, \sqcup, R)$$

All other δ rules transition to q_{reject}. This takes $n/2$ steps in the worst case (when the input is in PAL, so has running time in $\Theta(n)$.

Proving there is no regular TM that can recognize this language with running time in $O(n)$ is tougher, since we need to consider the best possible TM algorithm. Since the number of states is finite, the TM can match only a finite number of input symbols at a time. To do the match, it needs to look at the corresponding symbols in the second half of the string. The average distance between matching symbols is $n/2$ (the distances range from 1 for the innermost symbols to n for the outer symbols), and there are $n/2$ symbols to match. Thus, we need $\Theta(n^2)$ steps to decide PAL on a regular TM.

The original question asked this for $O(n^3)$ instead of $O(n)$. I suspect there are languages that a Janus machine can decide in $\Theta(n^3)$ steps that cannot be decided by a TM in $O(n^3)$ steps, but I'm not sure. A very convincing proof of this (or the opposite) is worth a full-credit exemption on the final.

d. set of languages that can be recognized by a Janus machine (as defined in Exam 2) in $O(n^3)$ Steps $\subset$ set of languages that can be recognized by a TM in $O(n^3)$ steps

Answer. False. This must be false, since we know a Janus machine can simulate a regular TM with the same number of steps (just let the right head wander down the blank tape to the right).

e. $n2^n \in O(2^n)$

Answer. False. By the definition of O, if $f(n) \in O(2^n)$ if and only if there exists positive constants c and n_0 such that for all $n \geq n_0$, $f(n) \leq c2^n$. We show $n2^n$ is not in $O(2^n)$ by showing that no matter what values are chosen for c and n_0 there is some value n where $n2^n > c2^n$. Choose $n = c + 1$. (If $c < n_0$, choose $n = n_0$.)

Problem 3: Program Complexity. Describe the asymptotic time complexity of deciding the language, $MULT = \{1^x * 1^y = 1^z \mid x, y, z \in \mathcal{N}, z = x \times y\}$, as precisely as you can. A good answer would provide the tightest big-O and Ω bounds you can. A correct Θ bound for the stated problem (that is, an argument that there is no asymptotically faster algorithm exists) is worth bonus points, but be aware that an incorrect answer us worse than a correct but less precise answer.

Answer. We know $MULT \in \Omega(n)$ since we must at least look at all the input.

We know $MULT \in O(n^2)$ since we know an algorithm that decides $MULT$ in $\Theta(n^2)$ steps:

a. Start by crossing off (with an X) the first 1.
b. Move to the end of the input (search for the $\sqcup$ and then move right).
c. If the square contains an $=$, skip to step 6.
d. Otherwise, replace the 1 with a $\sqcup$ and move left, past the $=$. Cross off (replace with a Y) the first 1 found moving left. Return to step 2.
e. If the $*$ is reached without finding a 1, replace all the Ys with 1s, and move back to the left until an X is found and move right. Return to step 1.
f. Check the tape contains $X^* * Y^* = \sqcup$.

Thus, we know $MULT \in \Omega(n)$ and $MULT \in O(n^2)$. We do not have a Θ bound, though, and getting one is tougher. This would require arguing that there is not asymptotically faster algorithm. (Note that the fact that there are known algorithms for computing binary multiplication with running times in better than $\Theta(n^2)$, does not mean there exist faster algorithms for unary multiplication.)

Problem 4: Unary Subset Sum. (Sipser's 7.16) Let *UNARY_SSUM* be the subset sum problem in which all numbers are represented in unary.

a. Why does the NP-completeness proof for *SUBSET_SUM* fail to show *UNARY_SSUM* is NP-complete?

Answer. The crux of the question is that we measure complexity as a function from the size of the input to the number of steps required to run the decided on that input. When the input is represented in unary, the size of the input is equal to the value (that is, it takes x squares to represent the input value x). When the input is represented in binary (or any higher base), it takes $\log x$ squares to represent the input value x. Hence, the if size of the input in binary is N, the size of the same input in unary is 2^N. This, the size of the input increases exponentially, even though the problem is of the same difficulty as it would be for the corresponding binary input. The reduction proof fails because the reduction from $3SAT$ to *UNARY_SSUM* requires more than polynomial time. The input size increases exponentially, so the transformation to generate the corresponding input for *UNARY_SSUM* requires a number of steps that is exponential in the size of the input.

b. Show that $UNARY\_SSUM \in$ P.

Answer. To show a language is in **P**, we need to show there is a polynomial-time algorithm that decides it. Note the we can simulate a nondeterministic TM in exponential time using a deterministic TM. Hence, if *SUBSET_SUM* is in **NP**, we know *UNARY_SSUM* (the same problem, but with exponentially larger input size) is in **P**, since we can solve it by first converting the input to binary representation (which can be done in polynomial time) and shrinks the input size to $\log N$. Then, we can simulate the nondeterministic polynomial-time TM running on this input (which has size $\log N$) in time that is exponential in its input size, which is polynomial in $\log N$.

Problem 5: Genome Assembly. In order to assemble a genome, it is necessary to combine snippets from many reads into a single sequence. The input is a set of n genome snippets, each of which is a string of up to k symbols. The output is the smallest single string that contains all of the input snippets as substrings. For example, if the input is

$$\{\texttt{ACCAGAATACC}, \texttt{TCCAGAATAA}, \texttt{TACCCGTGATCCA}\}$$

the output should be `ACCAGAATACCCGTGATCCAGAATAA`:

```
ACCAGAATACC
                 TCCAGAATAA
       TACCCGTGATCCA
```

a. Prove that the genome assembly problem is in NP.

Answer. First, we state the problem as a decision problem:

$L_{GA} = \{\langle R, m\rangle | R$ is a a set of reads, $R = \{r_1, r_2, \ldots, r_n\}$ where each $r_i \in \Sigma^*$, and there exists some string x where $|x| = m$ and each r_i is a substring of x. $\}$

To prove L_{GA} is in NP, we show that there is a polynomial time verifier for L_{GA}. The verifier is the string x that contains all the substrings in R and has length m. It is easy to test if x is a valid certificate: (1) Check the length of x is less than or equal to m; (2) For each string r_i, check that x contains r_i as a substring. This can be done in P since we can check each string by scanning x (that takes at most $|x|$ steps, and we know from step 1 that $|x| \leq m$), and we need to do this once for each of the n substrings.

b. Prove that the genome assembly problem is NP-Complete.

Answer. To show the L_{GA} problem is NP-Complete, we need to show that it is in NP and that every problem in NP can be reduced in polynomial time to L_{GA}. We showed the first part in part a. To show the second part, we need to show that some known NP-Complete problem can be reduced in polynomial time to L_{GA}. By the definition of NP-Complete, we could reduce any NP-Complete problem to L_{GA}, but our proof will be easier if we pick a problem where the mapping is more straightforward. The most obvious choices are the Hamiltonian Path problem and the Traveling Seller Problem. We will use $HAMPATH$. To do the reduction proof, we need to show how any instance of the $HAMPATH$ problem can be transformed into an input to the L_{GA} problem, where the output of the L_{GA} problem can be used to solve the HAMPATH problem instance. So, what we need to do is take the graph that describes the $HAMPTAH$ input and convert it to a set of strings and k value for the L_{GA} problem. The basic strategy is to map it to strings that will align into a superstring of the correct length if and only if there is a Hamiltonian path. For details on the reduction, see Class 27.

c. Speculate on how the human genome was sequenced even though it involves solving an NP-Hard problem for a large input size. The human genome is about 3 Billion base pairs. Readers at the time were able to read about 700 bases per read fragment, and sequencing the genome involved aligning about 30 million fragments.

Answer. Discussed in Class 27.