

## Problem Set 6 Comments

**Problem 1: Asymptotic Notation.** The way we advocated using asymptotic notation to describe algorithm complexity in Classes 21 and 22 is different from how Sipser uses it in Section 7.1. Describe (at least) two ways in which the book's use differs from the recommended use in class, and illustrate each with a specific example from Sipser's text.

**Answer:** Sipser's book sometimes uses the asymptotic notations as sets, but sometimes as if they produce values. 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  $O(n)$ ."

Sipser uses equality (e.g.,  $f(n) = O(n^3)$ ) to show a function is a member of a  $O(g(n))$  class. The class lectures 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.,  $O(n) + O(\log n) = O(n)$ ).

Sipser uses  $O$  to analyze algorithms, when a tight  $\Theta$  bound could be easily stated. For example, in the previous sentence, 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: Algorithm Analysis.** What is the asymptotic running time of a Turing machine that can decide this language from Exam 2,  $A = 0^n 1^m 0^{n/m}$  where  $n \geq 0$  and  $m \geq 1$  (that is, the input is  $n$  zeros, followed by  $m$  ones, followed by  $n$  divided by  $m$  zeros)? A reasonable big- $O$  bound with a convincing argument is worth full credit. A correct  $\Theta$  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  $\Theta$  bound is worse than a correct big- $O$  bound.

**Answer:** We use  $N$  to represent the size of the input. For a string in the language,  $N = n + m + (n/m)$ . We can obtain a big- $O$  bound by analyzing an algorithm that decides the language. Here is one such algorithm as described in the Exam 2 comments (annotated with numbers to identify the major parts):

- (1) We can reverse the input by first copying it: start by writing a # at the end of the input, then go left to right over the input, find the first unmarked symbol, marking it, and finding the first blank on the tape and overwriting it with that symbol. Once all the input is marked (the # is reached),
- (2) copy the copy back into the input space but start from the end of the copy (replacing each symbol with a blank instead of marking it), but putting the symbols on the tape in reverse order. Then,
- (3) simulate the multiplication machine.

For part 1, the number of steps is in  $\Theta(N^2)$ . For each input symbol, we need to move from the # to that symbol (the number of steps varies from 1 to  $N$ , so the average number of

steps here is approximately  $N/2$ ). Then, we need to move to the first blank on the tape. This involves moving  $2d + 1$  squares, where  $d$  is the distance from the end of the input (the #) to the current square. Since the average value of  $d$  is  $N/2$ , this involves on average  $N + 1$  steps. We need to do this for each of the  $N$  input symbols, so the total number of steps is  $N(N/2 + N + 1)$  which is in  $\Theta(N^2)$ .

For part 2, we copy the input in reverse order. The average number of steps for each symbol is  $N$ , so the number of steps to copy all the symbols is in  $\Theta(N^2)$ .

For part 3, we simulate the multiplication machine (from Sipser's Example 3.11). The number of steps for this is also in  $\Theta(N^2)$ . This is a bit tricky to see, but the easy way to see it is to note that the algorithm works by marking out each square at most once; the maximum work to mark each square is a complete traversal of the tape. So, the total number of steps for this part is in  $\Theta(N^2)$ .

Since we perform each part once, we can compute the total number of steps of the algorithm by adding the number of steps for each part. Hence, the total number of steps of the algorithm is in  $\Theta(N^2) + \Theta(N^2) + \Theta(N^2)$  which is equivalent to  $\Theta(N^2)$ .

Thus, we have an algorithm that executes with running time in  $\Theta(N^2)$ . This gives us an upper bound on the language decision problem's running time:  $O(N^2)$ . It does not, of course, give us a tight bound, since we don't know if there is a better algorithm.

**Problem 3: Cyclical Turing Machines.** Describe a problem for which a cyclical Turing Machine (as defined in Problem 2b of Exam 2) can solve asymptotically faster than it can be solved on a regular Turing Machine. The problem does not need to be an interesting problem, but your answer should include a clear and convincing argument, and explain why the problem is in **TIME**( $t(n)$ ) for the cyclical Turing Machine by **not** in the same time complexity class for the regular TM.

**Answer:** Here's an easy one:

$$L_{LAST} = \{w \mid w \text{ ends with a } 0\}$$

The best algorithm to decide  $L_{LAST}$  with a regular TM must traverse the input to the end, to look at the last symbol to see if it is a 0. This requires  $n$  steps, where  $n$  is the length of the input, so it is in **TIME**( $n$ ).

With a cyclical TM, we can decide the language in two steps — just move left (cycling to the last non-blank square), and examine the last input symbol. This requires a constant (2) number of steps, regardless of the length of the input, so it is in **TIME**(1).

**Problem 4: Multidimensional Turing Machines.** Sipser's Theorem 7.8 shows that for every  $t(n)$  time multitape Turing machine there is an equivalent single-tape Turing machine that has time in  $O(t^2(n))$ . Consider the similar question for a  $t(n)$  time multi-dimensional Turing machine (as defined in Class 15). A correct proof for any polynomial bound is worth full credit. If you can prove that your bound is the lowest possible bound is worth at least 50 bonus points.

**Answer:** To get an upperbound on the number of steps required to simulate a multidimensional TM with a normal TM, we need to consider the maximum number of steps required to simulate one step. Without getting into the details of how the multidimensional tape is mapped to the single dimensional tape, we know the worst possible move for the MDTM would involve completely traversing the active part of the 1DTM (that is, going from the left edge of the tape to the rightmost active square). So, we need to figure out what the rightmost active square could be for a MDTM after  $t(n)$  steps. This depends on how we map the MDTM squares to the 1DTM. We solve for the case where the MDTM is two dimensional.

Suppose we map the squares in a spiral, starting from the center square (where the input and head start on the MDTM), spiraling outward. We can label the squares as a Cartesian plane using tuples, so  $(0, 0)$  is the center square,  $(1, 0)$  is right of the center square, and  $(-1, 0)$  is left of the center square. On the 1DTM, the first square is square 0, and we label the squares from left to right. So, the mapping between  $(x, y)$  is the number of squares in the spiral starting from  $(0, 0)$  to reach  $(x, y)$ . The number of squares in the first circle of the spiral is 8, in the second is 16, in the third is 24, etc., so the number of steps in the  $j^{\text{th}}$  circle is  $8j$ , and the total number of steps in all circles up to the  $j^{\text{th}}$  circle is  $\sum_{i=1}^j 8i$  which is  $8 \sum_{i=1}^j i = 8(j + 1/2)j \approx 4j^2$ . So, reaching a square  $(x, y)$  where  $\max(x, y) \leq k$  requires at most  $4k^2$  steps.

If the MDTM moves out from the center with each step, it can reach square  $(-t(n), 0)$  in  $t(n)$  steps. This maps to a square within  $4(t(n)^2)$  squares of the starting square on the 1DTM. So, at worst, simulating each step of the MDTM with the 1DTM involves  $4(t(n)^2)$  steps. The number of steps is  $t(n)$ , so the total number of steps of the 1DTM is bounded by  $4(t(n)^3)$ . This is a polynomial bound. A tighter bound could be found by showing that the maximum number of steps required to simulate one move is actually much less than traversing the entire length of the tape, but looking at the mapping more closely, and by noting that the maximum active length is not the maximum length calculated above until the end of the simulation. This would get a tighter bound, but not a tight bound (which I believe would be extraordinarily difficulty to do, and should be worth much more than 50 bonus points!).

**Problem 5: 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 that we can simulate a nondeterministic TM in exponential time on 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) which shrinks the input 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 6: 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  $\{\text{ACCAGAATACC}, \text{TCCAGAATAA}, \text{TACCCGTGATCCA}\}$ , 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:

$L_{GA} = \{\langle \{x_1, x_2, \dots, x_n\}, m \rangle\}$  where each  $x_i$  is a string and there is a string  $X$  of length  $m$  that includes all  $x_i$  strings as substrings

Note that we can use the decision problem to find the minimum possible value of  $m$  for a set of strings by trying each value of  $m$  from  $1, 2, \dots$  until the string is accepted. This is guaranteed to be terminated when  $m$  reaches  $\sum_i x_i$  since we can always just concatenate the strings to form  $X$ .

To prove  $L_{GA}$  is in NP, we show that there is a polynomial time verifier for  $L_{GA}$ . The verifier is the string  $X$ . It is easy to test if  $X$  is correct in polynomial time: (1) Check the length of  $X$  is less than or equal to  $k$ ; (2) For each string  $x_i$ , check that  $X$  contains  $x_i$  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  $n$  strings.

- 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  $L_{GA}$ . In theory, 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*, and leave the reduction from TSP for you to consider (possibly on the final).

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 *HAMPATH* input and convert it to a set of strings and  $k$  value for the  $L_{GA}$  problem. For details on the reduction, see Class 26.

- c. Explain how the human genome was sequenced even though it involves solving an NP-Complete 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 26.

**Problem 7: Complexity Congress.** Write a short (no more than one page) essay explaining the  $P = NP$  question in a way that (1) would be understandable to a typical congressperson, and (2) that makes it clear why it is a compelling and interesting problem. Especially good answers will also convince the prospective congressperson why it matters to them. Note that unlike some fifth graders, most congresspeople (with the possible exceptions of Representatives Vern Ehlers, Bill Foster, Rush Holt, Jerry McNerney, and John Olver) should not be expected to already understand what Turing Machines, algorithms, complexity classes, polynomials, and nondeterminism are, among other things.

**Answer:** The most disappointing thing about this question is that several of your answers said that NP meant “non-polynomial” time. This is very wrong, even for a congressperson. Even if  $P$  is not equal to NP, the class NP still includes everything in P. The “N” in NP stands for “nondeterministic”. Good answers to this question used an example problem that is of obvious importance to a congressperson (for example, the genome assembly problem from the previous question would be a good choice since congress has been spending billions of the taxpayers dollars trying to solve this, it seems reasonable for them to understand what the problem is and why it is hard). Then, explain how if you could always guess right, it would be easy to solve the problem quickly, but we don’t know of anyway to solve it quickly with realistic computers. The article you read over Spring Break includes many excellent examples explaining NP-Completeness and its importance in an accessible way, so you might want to read that over again for more ideas on this question.