CS 121 Homework 5: Fall 2020

Some policies: (See the course policy at `http://madhu.seas.harvard.edu/courses/Fall2020/policy.html` for the full policies.)

- **Collaboration:** You can collaborate with other students that are currently enrolled in this course (or, in the case of homework zero, planning to enroll in this course) in brainstorming and thinking through approaches to solutions but you should write the solutions on your own and cannot share them with other students.
- **Owning your solution:** Always make sure that you "own" your solutions to this other problem sets. That is, you should always first grapple with the problems on your own, and even if you participate in brainstorming sessions, make sure that you completely understand the ideas and details underlying the solution. This is in your interest as it ensures you have a solid understanding of the course material, and will help in the midterms and final. Getting 80% of the problem set questions right on your own will be much better to both your understanding than getting 100% of the questions through gathering hints from others without true understanding.
- **Serious violations:** Sharing questions or solutions with anyone outside this course, including posting on outside websites, is a violation of the honor code policy. Collaborating with anyone except students currently taking this course or using material from past years from this or other courses is a violation of the honor code policy.
- **Submission Format:** The submitted PDF should be typed and in the same format and pagination as ours. Please include the text of the problems and write **Solution X:** before your solution. Please mark in gradescope the pages where the solution to each question appears. Points will be deducted if you submit in a different format.
- **Late Day Policy:** To give students some flexibility to manage your schedule, you are allowed a net total of **eight** late days through the semester, but you may not take more than **two** late days on any single problem set. No exceptions to this policy.

By writing my name here I affirm that I am aware of all policies and abided by them while working on this problem set:

Your name: Todd Morrill

Collaborators: (List here names of anyone you discussed problems or ideas for solutions with)

No. of late days used on previous psets (not including Homework Zero): 1
No. of late days used after including this pset: 1

Questions

Please solve the following problems. Some of these might be harder than the others, so don't despair if they require more time to think or you can't do them all. Just do your best. Also, you should only attempt the bonus questions if you have the time to do so. If you don't have a proof for a certain statement, be upfront about it. You can always explain clearly what you are able to prove and the point at which you were stuck. You can always simply write **"I don't know"** and you will get 15 percent of the credit for the problem. If you are stuck on this problem set, you can use Piazza to send a private message to all staff.

Note on reading the textbook: If you are stuck on some of the problems, try consulting the book to 1) understand the concepts the question is referencing, and 2) review the way similar theorems are proved in the book.

Problem 1:

Problem 1.1 (4 points): Prove that the function $f_1 : \{0,1\}^* \to \{0,1\}$ for which $f_1(x) = 1$ iff $x = 1001111$ is computable.

Solution 1.1

f_1 can be computed by simply checking if $x = 1001111$, which can be done in constant time using a high-level programming language or equivalently using a TM. If $x = 1001111$ the procedure returns 1 (i.e. $f_1(x) = 1$). The procedure returns 0 otherwise (i.e. $f_1(x) = 0$).

Problem 1.2 (6 points): Prove that the function $f_2 : \{0,1\}^* \to \{0,1\}$ for which $f_2(x) = 1$ iff x is an encoding of a Turing Machine that calculates f_1 is uncomputable.

Solution 1.2

The proof is by Rice's theorem.

Nontriviality: There exist two different TM's M, M' such that $f(M) = 1$ and $f(M') = 0$. For example, M could be the TM that computes f_1 as described in **solution 1.1**. M' could be the TM that computes the constant one function.

Semantic Property: Recall that $f_2(M) = 1$ iff M is an encoding of a Turing Machine that calculates f_1. If M and M' are functionally equivalent TMs then $M(x) = M'(x)$ for all $x \in \{0,1\}^*$. This would imply that $f_2(M) = 1$ iff $f_2(M') = 1$.

From this we conclude that f_2 is uncomputable.

Problem 1.3 (5 points): Prove or disprove: There exists a constant C and an uncomputable function $f : \{0,1\}^* \to \{0,1\}$ such that the number of values of x for which $f(x) = 1$ is at most C.

Solution 1.3

Suppose for the sake of contradiction that there was a constant C and an uncomputable function $f : \{0,1\}^* \to \{0,1\}$ such that the number of values of x for which $f(x) = 1$ is at most C.

If this were the case, we could create a finite length array of these values for which $f(x) = 1$ and check if x is equivalent to any of the bit strings in the array (as we did for one particular value of x in **solution 1.1**), which could be done in time $O(C)$. If x matches any of the values in the array, the procedure returns 1. The procedure returns 0 otherwise. But this would imply that f is computable, which contradicts the fact that f was given to be uncomputable.

From this we conclude that there does not exist a constant C and an uncomputable function $f : \{0,1\}^* \to \{0,1\}$ such that the number of values of x for which $f(x) = 1$ is at most C.

Problem 2: For each of the following problems, write either "can be proven uncomputable by Rice's theorem" (i.e. the function is semantic and nontrivial) or "can't be proven uncomputable by Rice's theorem" for 2 points (no justification necessary for this part). Then, write either "computable" or "uncomputable", with brief justification, for a variable number of points. (If you answered "can be proven uncomputable by Rice's theorem", justification is why the function's semantic and nontrivial.)

Problem 2.1 (2+3 points): $f(M) = 1$ iff there is an input x such that $M(x) = 1$.

Solution 2.1

f can be proven uncomputable by Rice's theorem. f is uncomputable.

Nontriviality: There exist two different TM's M, M' such that $f(M) = 1$ and $f(M') = 0$. For example, M could be the TM that computes the constant 1 function. M' could be the TM that computes the constant zero function.

Semantic Property: Recall that $f(M) = 1$ iff there is an input x such that $M(x) = 1$. If M and M' are functionally equivalent TMs then $M(x) = M'(x)$ for all $x \in \{0,1\}^*$. This would imply that $f(M) = 1$ iff $f(M') = 1$.

From this we conclude that f is uncomputable.

Problem 2.2 (2+3 points): $f(M) = 1$ iff any of M's instructions include "move left".

Solution 2.2

f can't be proven uncomputable by Rice's theorem. f is computable.

M encodes its transition table, which is finite by definition. This transition table can be searched for the action "move left" in time $O(n)$ where n is the number of entries in the transition table.

Correctness: If the search procedure finds the action "move left" it returns 1 (i.e. $f(M) = 1$). If the procedure does not find the action "move left", it returns 0 (i.e. $f(M) = 0$).

From this we conclude that f is computable.

Problem 2.3 (2+Bonus 0 points): $f(M) = 1$ iff there exists an input for which M ever moves left.

Solution 2.3

f can't be proven uncomputable by Rice's theorem because f isn't semantic. f is computable.

f can be equated to a DFA that reads input from left to right and if it encounters an action to "move left" it enters into an accept state (i.e. $f(M) = 1$). If M halts before moving left, then the DFA enters into a reject state (i.e. $f(M) = 0$).

Problem 2.4 (2+4 points): $f(M, n) = 1$ iff it is the case that for every input of length n, M runs in time at most 2^n.

Solution 2.4

f can't be proven uncomputable by Rice's theorem because f isn't semantic. f is computable.

Since n is a fixed number, there are a finite number of bit strings $x \in \{0,1\}^n$. We can run the Universal TM on each of these bit strings (i.e. $U(M, x)$) and check if each one halts in at most 2^n steps (adding extra steps for the overhead of running the simulation). If $M(x)$ halts in at most 2^n steps for all $x \in \{0,1\}^n$ then the procedure returns 1 (i.e. $f(M, n) = 1$). Otherwise, the procedure returns 0 (i.e. $f(M, n) = 1$).

Problem 2.5 (2+Bonus (hard) 0 points): $f(M) = 1$ iff it is the case that for every integer n and every input of length n, M runs in time at most 2^n.

Solution 2.5

f can't be proven uncomputable by Rice's theorem because f isn't semantic.

Problem 2.6 (2+3 points): $f(M) = 1$ iff it is the case that for every x, $M(x) = 1$ if and only if $x = x^R$. (x^R denotes the reverse of x.)

Solution 2.6

f can be proven uncomputable by Rice's theorem. f is uncomputable.

Nontriviality: There exist two different TM's M, M' such that $f(M) = 1$ and $f(M') = 0$. For example, M could be the TM that we saw in class that computes the PALINDROME function. M' could be the TM that computes the constant one function.

Semantic Property: Recall that $f(M) = 1$ iff it is the case that for every x, $M(x) = 1$ if and only if $x = x^R$. If M and M' are functionally equivalent TMs then $M(x) = M'(x)$ for all $x \in \{0,1\}^*$. This would imply that $f(M) = 1$ iff $f(M') = 1$.

From this we conclude that f is uncomputable.

Problem 3:

Problem 3.1 (5 points): The "configuration" of a Turing Machine at a time t when it is in the middle of a computation is information that determines its future operation. Specifically, for a machine M with k states and alphabet Σ, its configuration is the triple (q, i, x) where $q \in [k]$ is its current state, $i \in \mathbb{N}$ is the position of the tape head, and $x \in \Sigma^*$ is the string of cells on the tape between $\triangleright$ and the first ϕ. (Assume the TM never writes a ϕ or overwrites any but the leftmost ϕ.) Prove that if, on some input w, a machine M reaches the same configuration twice, i.e. $C_1 = (q_1, i_1, x_1)$ if the configuration at time t_1 and $C_2 = (q_2, i_2, x_2)$ is the configuration at time t_2 and $C_1 = C_2$, then M never halts on input w.

Solution 3.1

The proof is by contradiction. Suppose that a TM M that reaches the same configuration $C_1 = C_2$ at time t_1 and t_2 on input w does halt. When the TM transitions from configuration C_1 to C_2, it is really back in configuration C_1 by the equivalence in C_1 and C_2. Since TM's have a finite transition table and behave deterministically, the machine will again transition from C_1 to C_2, as it did the first time, in some finite number of steps. But this means that the machine will continue looping back to C_1 and in particular, it means that the TM will not halt on input w. But this is a contradiction, since we assumed that M would halt on input w.

From this we conclude that if a machine M reaches the same configuration twice (i.e. $C_1 = C_2$) at two different time steps then M never halts on input w.

Problem 3.2 (15 points): Let SPACE-HOG$(M, x, s) = 1$ if M is a Turing Machine that on input x uses more than s bits of space (i.e., tape head ever reaches the $s + 1$th cell of the tape). Prove that SPACE-HOG is computable.

Solution 3.2

The proof is by the Pigeon-Hole Principle (PHP). The number of configurations of M is $C = k \times s \times |\Sigma|^s$, where k is the number of states that M has, s is the index i such that M uses at most s slots on the tape, and $|\Sigma|^s$ is the number of configurations of the tape on alphabet Σ with length s.

We define a procedure below to compute SPACE-HOG.

```
def compute_spacehog(M, x, s):
    C = k x s x |Sigma|^s
    for _ in range(C):
        Simulate one step of M(x) using the universal TM
        if the index i>s:
            return 1
    return 0
```

Correctness: When SPACE-HOG$(M, x, s) = 1$, compute_spacehog(M, x, s) = 1, because it checks if M's tape has used more then s slots of the tape at every step and returns 1 if this happens. When SPACE-HOG$(M, x, s) = 0$, compute_spacehog(M, x, s) = 0 for one of two possible reasons: 1) $M(x)$ halts before $i > s$ or 2) M has potentially cycled through all of its unique configurations and if it hasn't halted by that point, by the PHP, M will visit a configuration two times. By **solution 3.1** M will not halt because it will visit a configuration twice.

From this we conclude that that SPACE-HOG is computable.

Problem 4 (20 points): Recall that $\text{HALT}(M, x) = 1$ if M is the encoding of a Turing Machine and M halts on x, and $\text{HALT}(M, x) = 0$ otherwise.

For integer C, let $\text{HALT}_C(M, x) = 1$ if and only if both $\text{HALT}(M, x) = 1$ and M is a Turing Machine with at most C states.

Prove that there exists a constant C such that HALT_C is uncomputable.

Solution 4

The proof is by reduction from HALT to HALT_C.

Recall that the Universal TM U can evaluate another TM M on X using a constant number of states, which we denote C_U. In particular, $U(M, x) = M(x)$. Suppose for the sake of contradiction that for all constants C, HALT_C is computable, which means that HALT_{C_U} is computable. With this in hand, we can compute HALT as follows.

```
def compute_halt(M, x):
    return compute_halt_c_u(U, (M, x))
```

Correctness: When $\text{HALT}(M, x) = 1$, compute_halt(M, x) = 1 because compute_halt returns 1 iff compute_halt_c_u(U, (M, x)) = 1, which only returns 1 if $M(x)$ halts. compute_halt_c_u satisfies the condition that U is a TM with at most C_U states when fed the Universal TM. When $\text{HALT}(M, x) = 0$, compute_halt(M, x) = 0 because compute_halt returns 0 iff compute_halt_c_u(U, (M, x)) = 0, which returns 0 if $M(x)$ does not halt.

This implies that we can compute HALT, which we know to be uncomputable. From this we conclude that compute_halt_c_u is uncomputable. But this is a contradiction since we assumed that for all constants C, HALT_C is computable. This shows that there exists a constant C such that HALT_C is uncomputable.

Problem 5 (20 points): Let $f(M) = 1$ if M is a Turing Machine that ever writes the symbol @ on its tape on input the empty string. Let $f(M) = 0$ in all other cases. Show that $f(M)$ is uncomputable. (Hint: reduce from the halting problem.)

Solution 5

The proof is by reduction from HALT. Suppose for the sake of contradiction that we could compute f with a procedure called compute_f. With this procedure in hand, we could compute HALT as follows.

```
def compute_halt(M, x):
    M' = M with all occurrences of @ replaced with !
    def N(y):
        M'(x)
        Write @ on the tape
    return compute_f(N)
```

The line $M' = M$ with all occurrences of @ replaced with ! is required to ensure that compute_f doesn't return 1 in the case where M writes an @ on the tape but doesn't halt. This replacement can be done in $O(n)$ time in the size of the transition table.

Correctness: When $HALT(M, x) = 1$, compute_halt(M, x) = 1. If M'(x), which is equivalent to $M(x)$ up to a change of one character (@ $\rightarrow$!), halts then compute_f(N) = 1 because N writes an @ on the tape when input the empty string. When $HALT(M, x) = 0$, compute_halt(M, x) = 0. If M'(x) does not halt then compute_f(N) = 0 because N never writes an @ on the tape.

This implies that there exists a procedure to compute HALT, which we know to be uncomputable. Our only assumption was that there existed a procedure called compute_f, which must be false and we conclude that $f(M)$ is uncomputable.

Problem 6:

Problem 6.1 (Bonus, 0 points): Give an example of functions f, g, F, G such that $f(n) = O(F(n))$, $g(n) = O(G(n))$ but $f(g(n)) \notin O(F(G(n)))$.

$$f(n) = n \qquad g(n) = \frac{1}{n^2}$$
$$F(n) = n^3 \qquad G(n) = \frac{1}{n}$$

$$f((g(n)) = \frac{1}{n^2} \notin O(F(G(n)) = O(\frac{1}{n^3})$$

Problem 6.2 (10 points): Prove that if $f(n) = O(n^c)$ and $g(n) = O(n^d)$ for $c, d > 0$ then $f(g(n)) = O(n^{cd})$.

Solution 6.2

If $f(n) = O(n^c)$ and $g(n) = O(n^d)$ then there exist constants c_1 and c_2 such that $f(n) \leq c_1 n^c$ and $g(n) \leq c_2 n^d$ for all $n > n_0$ by the definition of Big-O.

Composing f and g, we have $f(g(n)) \leq c_1(c_2 n^d)^c = c_1 \cdot c_2^c \cdot n^{cd} = c_3 \cdot n^{cd}$ for all $n > n_0$, where $c_3 = c_1 \cdot c_2^c$. But this is precisely the definition of Big-O and therefore $f(g(n)) = O(n^{cd})$. From this we conclude that if $f(n) = O(n^c)$ and $g(n) = O(n^d)$ for $c, d > 0$ then $f(g(n)) = O(n^{cd})$.

The contrast between 6.1 and 6.2 may mean that the following problem, which you solved in sections, was harder than you thought

Problem 6.3 (Bonus, 0 points): Prove that if $A : \{0,1\}^* \to \{0,1\}^*$ and $B : \{0,1\}^* \to \{0,1\}^*$ are polynomial time computable functions then their composition $A(B(x))$ is also polynomial time computable.

Solution 6.3

The proof follows from **solution 2.2** from homework 4, which showed that the output from one TM could be fed to another with very little computational overhead, in particular in time $O(T_B)$ where T_B is the time complexity of B. The proof also follows from the results demonstrated in **solution 6.2**, which showed that the composition of two polynomial time functions is still polynomial.