CS 332 – Spring 2021, Assignment 7

Colaborators: None

Answer 1

(a) The input for f should be $\langle M, w\rangle$, where M is a TM and w is a string. The output should be the pair $\langle M'\rangle$, where M' is a TM.

(b) $T =$ "On input $\langle M, w\rangle$:

Construct M_1 as follows:

On input x:

1. If $x \neq w$: if $|x|\%2 == 1$ accept, else reject.
2. Else: run M on input w, if it accepts, accept, else reject.

Return M_1."

This way, the constructed TM M_1 will always accept/reject as OL_{TM}, yet when it gets to input w (which needs to be tested in order for OL_{TM} to be true), it'll solve the $\overline{A_{TM}}$. Furthermore, by returning a TM instead of an accept/reject decision, it works as a mapping function from $\overline{A_{TM}}$ to OL_{TM}.

(c) Assuming that there is a TM recognizer R for OL_{TM}, then using the reduction above, we can create another TM S:

$S =$ "On input $\langle M, w\rangle$:

1. Run T on input $\langle M, w\rangle$, and get M_1.
2. Run R on input M_1 and get the accept/reject decision d, if it loops loop.
3. If $d =$ "accept": If $|w|\%2 == 1$, then reject, else accept.
3. If $d =$ "rejected": If $|w|\%2 == 1$, then accept, else reject."

The TM above would recognize the $\overline{A_{TM}}$. Since we know that $\overline{A_{TM}}$ is unrecognizable, then no recognizer R can exist, which means that OL_{TM} is not Turing-recognizable.

(d) If we make use of the $\overline{A_{TM}}$, we can create a reduction from $\overline{A_{TM}}$ to $\overline{OL_{TM}}$:

$T' =$ "On input $\langle M, w\rangle$:

Construct and return M_1' as follows:

On input x:

1. If $x \neq w$: if $|x|\%2 == 0$ accept, else reject.
2. Else: run M on input w, if it accepts, accept, else reject."

From here, assuming that there is a recognizer for $\overline{OL_{TM}}$, R', then we can construct a TM that recognizes $\overline{A_{TM}}$:

$S' =$ "On input $\langle M, w\rangle$:

1. Run T' on input $\langle M, w\rangle$, and get M_1'.
2. Run R' on input M_1' and get the accept/reject decision d, if it loops loop.
3. If $d =$ "accept": If $|w|\%2 == 0$, then reject, else accept.
3. If $d =$ "rejected": If $|w|\%2 == 0$, then accept, else reject."

This TM would recognize $\overline{A_{TM}}$, since we know that this is not possible, and we also know that our TM T' works, then it must be that R' does not exist, which means that $\overline{OL_{TM}}$ is not recognizable.

Answer 2

(a) There exists a $c > 0$ and an n_0 so that $n^2(\log_7 x + x) \leq cn^3$ for every $n \geq n_0$. These values are $c = 2$ and $n_0 = 0$. Thus $n^2(\log_7 x + x) = O(n^3)$

(b) There exists an n_0 so that, for every c $3n \leq cn^2$. This value is $n_0 = 3$, which is the point at which $3n \leq n^2$, and since $cn^2 \geq n^2$, for every $c < 0$, then $3n \leq cn^2$ for every c when $n \geq 3$.

(c) Since the little-o notation can mean that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then we can prove that $3^{\sqrt{n}} = 2^{o(n)}$, however we must first simplify this formula:

$3^{\sqrt{n}} = 2^{o(n)} \equiv \log_2(3^{\sqrt{n}}) = o(n)$

With that said, we can now apply the limits:

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2(3^{\sqrt{n}})}{n} = \lim_{n\to\infty} \frac{\log_2(3^{\sqrt{n}})}{n} = \lim_{n\to\infty} \frac{3^{\sqrt{n}}}{2^n} = \lim_{n\to\infty} \frac{\frac{1}{3^n}}{2^n} =$

$= \lim_{n\to\infty} \frac{1}{3^n \times 2^n} = 0$

Thus, $3^{\sqrt{n}} = 2^{o(n)}$.

Answer 3

(a) True

(b) False

(c) False

(d) False

(e) True

(f) True

(g) True

(k) True

(l) False

(m) True

(n) True

(o) True

(p) False

(q) True

Answer 4

(a) On input w, pick the first character, if it's a 0, remove it, and go through the tape until you find a 1, then remove that and return to the start. If it was a 1, do the same thing, delete it and go until you find a 0, delete it, and return. If at any point you cannot find a 1 or 0, then reject, else run until you have an empty tape, and accept.

Since there is no writing, and the TM deletes every number it encounters, there is no chance of looping, which means that it'll end up having to make a decision.

For space, $A \in SPACE(n)$, since it won't use any space outside of its input size, never writing anything else, or anywhere else. As for time, the machine will inherently stop at least once at each space (hence the n), and for each one, it'll recur whatever is left of the tape, but since it's deleting as it goes, the tape will grown increasingly smaller, meaning that it won't actually reach the n^2 time, rather it'll be $A \in TIME(n \times \log n)$.

(b) Queue up all the vertices of the Graph, and then for each, using another queue of all the other vertexes except the previous one, go through that queue, and for each of these, make a third queue with all the vertices except the prior two, and test for triangles in each of these combinations. If at any point you find a triangle return false, else run until you've dealt with every vertex in the queue, and return true.

This works on the basis that we deal at least once with every vertex, and for each of these, we check the combinations with every other possible duo, and only return true after all these are checked.

This algorithm will run in polynomial time, as for every vertex (n), it'll create a list of size $n - 1$ to check, and for each of these, it'll create another list of size $n - 2$ to check. This makes the algorithm have a runtime of $n \times (n - 1) \times (n - 2)$, which is polynomial, thus making it so that $TF \in P$.

(c) The numbers of the Fibonacci sequence increase exponentially in size, with the 100th number taking up 70 digits for its binary representation, while the 200th takes 139, and the 300th takes 209. Although TM's can represent numbers this large, and can do operations on binary in polynomial time (like addition), the TM would eventually reach a point where it'd have to travel quite a long time to do this math, and since it takes traveling back and forth a lot to do addition, addition in numbers this large is unfeasible in polynomial time simply by the amount of numbers it would have to list before it reached the conclusion, as it takes up to 2^n digits to represent the numbers until the nth natural number.

(d) This algorithm would use a two-tape TM, on the first it would have its input, and the second it would start empty. Then, going cell by cell on the first tape, for each cell that has a 1, it would write the unary representation of the F_n for that number, then when it moved to the next cell, it would write a on the second tape, and then go back three 's to find the F_{n-2}, and for each 1 in between the it move to, and the second one, it would add those 1's again after the first (from the end), once it does this, it adds another , advances on tape 1 (clearing the 1 behind itself), checks for a 1, and rinse and repeat.

At the end, once it has gone through and cleared every 1 from tape 1, it would have the resulting F_n in tape 2, and then the TM could, for every 1 in the second tape, add 1 to the binary number in tape 1, since it can do this in polynomial time, then it this part would run in polynomial time.

For the first part, it would also run in polynomial time since it would only go through a small section of tape 2 for each 1 in tape 1. Hence it would run in polynomial time, since it only takes n spaces to represent the n natural number, as well as all the ones before, in unary (a significant improvement from the 2^n value from before.

This works based on the basic idea that the fibonacci number is equal to the previous number plus the number before. And as seen, runs in polynomial time.