Problem set 4 Complexity

Bayard Walsh

February 2024

1

1.1 a

If f is $\Omega(g)$, we have that there exists $c > 0$ and $n_* \in \mathbf{N}$ such that for every $n > n_*$, we have $f(n) \geq c * g(n)$, by definition.

For contradiction, assume f is $o(g)$. Then, for every $c > 0$, there exists $n_* \in \mathbf{N}$ such that for every $n > n_*$, we have $f(n) < c * g(n)$.

However if f is $\Omega(g)$ there must be some $c > 0$, say c', such that $f(n) \geq c' * g(n)$.

Therefore, for c', $f(n) \not< c' * g(n)$, so not every $c > 0$ gives us $f(n) < c * g(n)$, which implies f is not $o(g)$. Therefore, if f is $\Omega(g)$, f is not $o(g)$.

1.2 b

If f is $o(g)$, then for every $c > 0$, and $n_* \in \mathbf{N}$ such that for every $n > n_*$, we have $f(n) < c * g(n)$.

Therefore, if f is **not** $o(g)$, then there must be some $c > 0$, say c', such that $f(n) \not< c' * g(n)$.

If we have $f(n) \not< c' * g(n)$, then we have $f(n) \geq c' * g(n)$. Therefore we have c', such that $c' > 0$ and $f(n) \geq c' * g(n)$ meaning that some c exists where $f(n) \geq c * g(n)$, so f is $\Omega(g)$ by definition. Therefore, if f is not $o(g)$, f is $\Omega(g)$.

2

2.1 A

Algorithm

Note: we define *generating all prefixes starting at i from string $w \in L$* as all ordered sub strings starting at w^i extending up to w^n, or the substrings

$(w^i), (w^i w^{i+1}), \cdots, (w^i w^{i+1} \cdots w^n)$, with respect to some w where each substring is also $\in L$.
Note: If $c_{i+|w''|}$ is already marked at some position, and it is marked again, do nothing.

Give some arbitrary w where $|w| = n$, follow this procedure:

Create an empty tape T of length n cells.

Generate all prefixes of w at position 1 that are $\in L$.

For every generated prefix w', mark cell $c_{|w'|}$ on tape T.

Iterate through every $i \in 2, 3 \cdots, n$, and apply the following process:

```
    Check if $c_i$ is marked on $T$.
    If $c_i$ is unmarked: continue to $i + 1$.
    If $c_i$ is marked:
        Generate all prefixes of $w$ at position $i$ that are $\in L$.
        If no prefix strings are generated at position $i$: continue to $i + 1$.
        For every prefix string generated at position $i$, $w''$, mark cell $c_{(i+|w''|)}$ on $T$.
```

If cell $c_n \in T$ is marked return True, if not False.

Correctness

Assume L^* accepts some arbitrary w. Therefore there must be some way of constructing $w^1 w^2 \cdots w^m \in L$ such that $w^1 w^2 \cdots w^m = w$. Therefore w^1 must equal some continuous substring starting at index 1 of w and w^2 must equal another substring starting at index $|w^1| + 1$ of w, and this pattern will be repeated with each w^i until reaching w^m, by nature of how w is broken into substrings. Note that our algorithm will find this sequence, because it generates all prefixes of $w \in L$ starting at 1 from string w and marks the length of all reachable indices from 1 based on the lengths of every valid prefix, and then uses those indices to generate more prefixes and so on. Therefore our algorithm will find w^1, and then w^2 (as index $|w^1| + 1$ is marked by our algorithm from checking w^1, and w^2 must be some substring of w starting at $|w^1| + 1$), until eventually finding w^m. w^m must include the end part of w at cell c_n by definition of $w^1 w^2 \cdots w^m = w$, therefore the algorithm will mark the n-th cell of T upon considering w^m, meaning that our algorithm will return True if L^* accepts w.

Next assume L^* rejects some arbitrary w. Therefore there is no possible way of generating a sequence of $w^1 w^2 \cdots w^m$, such that $w^1 w^2 \cdots w^m = w$. Note that our algorithm only returns True if it marks the tape at position n, which will only happen if our algorithm has marked a sequence of previous tape positions corresponding to the starts of substrings generated at indices with substrings that eventually end at n. Because this sequence does not exist (as L^* rejects w, there is no possible way of generating a sequence of $w^1 w^2 \cdots w^m$, such that $w^1 w^2 \cdots w^m = w$), our algorithm will return False if L^* rejects w.

As accepting maps to True and rejecting maps to False, our algorithm is correct.

2.2 B

Runtime

First, we will read input string w onto both of the two tapes of the two tape Turing Machine. Next, we introduce symbol θ such that $\theta \notin \Sigma$ of w, and we will use θ as a way to mark each w_i on input w if that position is reachable from our algorithm. Note that while our algorithm references making another tape T, we can use θ on w_i symbols as a way to simulate T, meaning that checking for T will not involve an extra tape and instead can be done by checking if that θ is on the top tape, which will help us save data storage.

Next, on an input of length n, we will iterate through and apply our string generating algorithm for each cell. Iterating will involve moving the two heads together one step forward to the right on the Turing machine. For a given cell i, we will generate up to n sub strings. There are up to n sub strings for a given i, because our sub strings are all generated as continuous sub strings from w and must start at w_i and end at w_n (and $0 \leq i \leq n$), so for a given i, we only generate sub strings starting at i ranging to $i+1, i+2, \cdots n$ meaning that we only generate a maximum of n sub strings for any given index i. In order to generate sub strings, our first head will stay at i and the next head will move forwards from i until reaching n, and at each step we will check if the string between the two heads is $\in L$. Therefore this process of generating sub strings will be done linearly, with $O(n)$ steps to generate all sub strings for a given index w_i. After the second head has reached n, it will step back to i, and then the two strings will step to $i+1$ together. For every w' sub string generated with this approach, we check if $w' \in L$. As given we have that some Turing Machine M decides $L \in O(n^k)$ on a one-tape Turing Machine. If we can decides $L \in O(n^k)$ on a one-tape Turing Machine, we will use the second head to simulate w' for L, while the other stays at position w' on the tape. Therefore using two heads, we can use the same $O(n^k)$ runtime for a given w', for up to N sub strings for a total of N indices. Therefore, our algorithm has $O(n^{k+2})$ runtime.

3

3.1 a

Algorithm

Note: ordering bits from the least to most significant bits, or $A_n, A_{n-1}, \cdots, A_1$.

First we start with triple $A_1, B_1, 0$, where A_1, B_1 are the first bits in A, B. Next, we calculate the sum of the bits of this triple using *3ADD*. The output of adding 3 bits will never be more than 2 bits long (the max input $1, 1, 1$ outputs 11 in binary), and we will use filler 0s if the output is less (for example *3ADD* $0, 0, 1$ will output 01 instead of 1). Therefore our output from *3ADD* will always be 2 digits long. In our 2 digit output, we will save the first digit as 'carry' and the second digit as the 'output', or $c_1 o_1$. We will concat each o digit to the front of an output string, which will be returned when the algorithm terminates. The carry digit c_1 will be used for $A_2 B_2 c_1$ in the next instance of *3ADD*. We will continue this pattern so that for every $i \in 1, 2, \cdots n$, we will apply *3ADD* to $A_i B_i c_{i-1}$ to generate $c_i o_i$. Note that for $A_n B_n c_{n-1}$ (the final step), we will append both c_i, o_i to the front of the output string, so our final output string is $c_n o_n o_{n-1} \cdots o_1$.

Correctness

I assume that the addition algorithm I was taught in elementary school actually works. This algorithm involves lining up the digits in A, B, so that A is above B, and each digit in A, B is aligned vertically, starting with the first digit. Next, we add the digits in each of the two strings, starting with the ones place and moving to the left. Any time there is a carry digit, carry the 1 and add the 1 to the next digit, represented by c in our algorithm and let the current digit be o in our algorithm. Our algorithm follows this method by applying *3ADD* to $A_i B_i c_{i-1}$ for each $i \in 1, 2, \cdots n$, which adds the corresponding digits at position i while considering the carry from the previous position if there is one. By using leading 0s if there is no carry and starting with 0 as our carry, our algorithm only adds the carry to a set of digits if the previous value had a carry digit. Note that *3ADD*$(A_1, B_1, 0)$ adds A_1, B_1 for the first digits of A, B, and for each successive A_i, B_i, *3ADD*(A_i, B_i, c_{i-1}) is computed from, c_{i-1} the previous, our algorithm follows the same process of carrying extra digits when needed. Therefore, our algorithm follows the elementry school addition algorithm, meaning the algorithm is correct.

Run time

In our algorithm, a given *3ADD* operation has a constant amount of circuit operations relative to the size of the input. Therefore, the function runs in $O(1)$

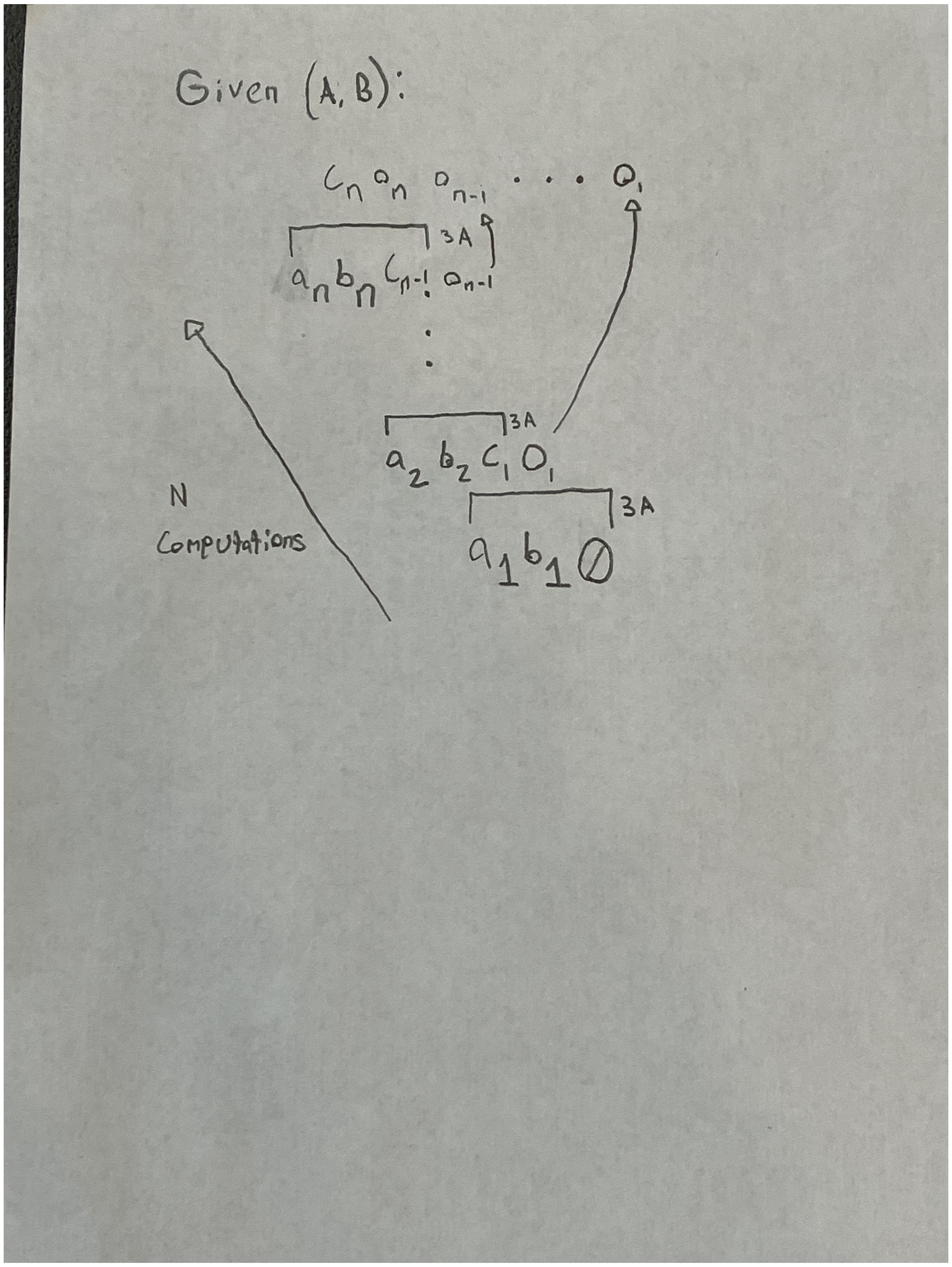

Figure 1: *Diagram for 3 a) algorithm*

time. As our algorithm applies *3ADD* in sequence n times, saving the c from each previous step in computation for the next *3ADD* step, it does $O(1)$ work $O(n)$ times. Therefore the circuit has size $O(n)$. As a circuit exists that solves $ADD_n(A, B)$ with the size $O(n)$, $ADD_n(A, B)$ has a circuit complexity upper bound $O(n)$.

3.2 b

To solve the MAJ circuit, I propose the following approach: first, count the number of 1 digits in a binary string and then use that amount to determine if x is in MAJ. First, start with an initial sum and carry both set to 0, or $sc = 00$. Then for each bit b_i in the binary string (from right to left, starting with the least significant bit): Apply the *3ADD* function with inputs b_i, s, and c, where b_i is the current bit, s is the current sum bit, and c is the current carry bit. Update S with the sum output of the *3ADD* function, adding 1 to the binary string output S whenever $s = 1$, and update c with the carry output of the *3ADD* function for the b_{i+1} instance. Next, after the addition step add leading 0s to the right side of S if $|S| < \log_2 n$, until $|S| = \log_2 n$. Finally OR the 2 most significant (rightmost) bits from S, and return the output. Note that if there are at least $\frac{n}{2}$ ones in x, then the binary string representation that counts the number of xs must have at least one 1 bit in either the most or second most significant bit of a bit string of length $\log_2 n$ with leading zeros, or else the sum of all ones $\in n < \frac{n}{2}$, meaning we return 0. If there is at least one 1 bit in either the most or second most significant bit of the binary representation of the number of ones in x, then we know that x is in MAJ and we return True. The *3ADD* function has runtime $O(1)$, and this operation is ran n times (for each bit in x). We add up to $\log_2 n$ leading 0s, and checking **OR** for the first two bits of S is $O(1)$. Therefore we have $O(n)$ runtime for this algorithm, meaning the circuit complexity is $O(n)$.

4

4.1 a

We want to show that $P \subseteq QP$
We have that $P = \bigcup_{k=1}^{\infty} TIME(n^k)$ and $QP = \bigcup_{k=1}^{\infty} TIME(2^{(\log n)^k})$
Note that $\lim_{n\to\infty} \frac{n^k}{2^{(\log n)^k}} = 0$ for some constant k. Therefore we have that n^k is $o(2^{(\log n)^k})$ by definition of o. Therefore n^k is also $O(2^{(\log n)^k})$, meaning that we have $\bigcup_{k=1}^{\infty} TIME(n^k) \subseteq \bigcup_{k=1}^{\infty} TIME(2^{(\log n)^k})$. Therefore $P \subseteq QP$.

4.2 b

Let us have $T(n) = 2^{(\log n)^2}$, which is time constructable as given.
As solved in a), we have that P is $o(2^{(\log n)^k})$ for any arbitrary constant k. Therefore, P is $o(2^{(\log n)^2})$, for $k = 2$.

By The Weak Time Hierarchy Theorem, we have that $TIME(o(2^{(\log n)^2})) \neq TIME((2^{(\log n)^2})^3)$
$TIME((2^{(\log n)^2})^3) = TIME(2^{(\log n)^2 \cdot 3})$
$\lim_{n \to \infty} \frac{2^{(\log n)^2 \cdot 3}}{2^{(\log n)^{10}}} = 0$. Therefore $2^{(\log n)^2 \cdot 3}$ is in $o(2^{(\log n)^{10}})$, and in $O(2^{(\log n)^{10}})$, and $TIME(2^{(\log n)^2 \cdot 3}) \subseteq TIME(2^{(\log n)^{10}})$
$TIME(2^{(\log n)^{10}}) \subseteq \bigcup_{k=1}^{\infty} TIME(2^{(\log n)^k})$
Therefore, P is $TIME(o(2^{(\log n)^2})) \neq TIME(2^{(\log n)^2 \cdot 3}) \subseteq TIME(2^{(\log n)^{10}}) \subseteq \bigcup_{k=1}^{\infty} TIME(2^{(\log n)^k}) = QP$. Therefore, $P \neq QP$

4.3 c

We want to show that $QP \subseteq EXP$
We have that $QP = \bigcup_{k=1}^{\infty} TIME(2^{(\log n)^k})$ and $EXP = \bigcup_{k=1}^{\infty} TIME(2^{n^k})$
We have that $(\log n)^k = o(2^{(\log n)})$ for some constant k, and that $o(2^{(\log n)}) = o(n)$ through log rules. Therefore $(\log n)^k$ is $o(n)$
Considering that $(\log n)^k$ is $o(n)$, for sufficiently large n, we have $(\log n)^k < n$.
We will take the limit of n, so we make the following substitution: $2^{(\log n)^k} < 2^n$
$\lim_{n \to \infty} \frac{2^n}{2^{n^2}} = 0$, therefore 2^n is $o(2^{n^2})$, and also $O(2^{n^2})$, and $TIME(2^n) \subseteq TIME(2^{n^2})$
$TIME(2^{n^2}) \subseteq \bigcup_{k=1}^{\infty} TIME(2^{n^k})$
$QP = \bigcup_{k=1}^{\infty} TIME(2^{(\log n)^k}) = TIME(2^{o(n)}) \subseteq TIME(2^n) \subseteq TIME(2^{n^2}) \subseteq \bigcup_{k=1}^{\infty} TIME(2^{n^k}) = EXP$
Therefore $QP \subseteq EXP$

4.4 d

Let us have $T(n) = 2^n$, which is time constructable as given.
As solved in c), we have that QP is $o(2^n)$.
By The Weak Time Hierarchy Theorem, we have that $TIME(o(2^n)) \neq TIME((2^n)^3)$,
$TIME((2^n)^3) = TIME(2^{3n})$
$\lim_{n \to \infty} \frac{2^{3n}}{2^{n^2}} = 0$. Therefore 2^{3n} is in $o(2^{n^2})$, and in $O(2^{n^2})$, and $TIME(2^{3n}) \subseteq TIME(2^{n^2})$
$TIME(2^{n^2}) \subseteq \bigcup_{k=1}^{\infty} TIME(2^{n^k})$
$QP = \bigcup_{k=1}^{\infty} TIME(2^{(\log n)^k}) = TIME(2^{o(n)}) \subseteq TIME(2^n) \neq TIME(2^{3n}) \subseteq TIME(2^{n^2}) \subseteq \bigcup_{k=1}^{\infty} TIME(2^{n^k}) = EXP$
Therefore $QP \neq EXP$