CS3052: Computational Complexity

Student ID: 200007413

27 April 2023

1

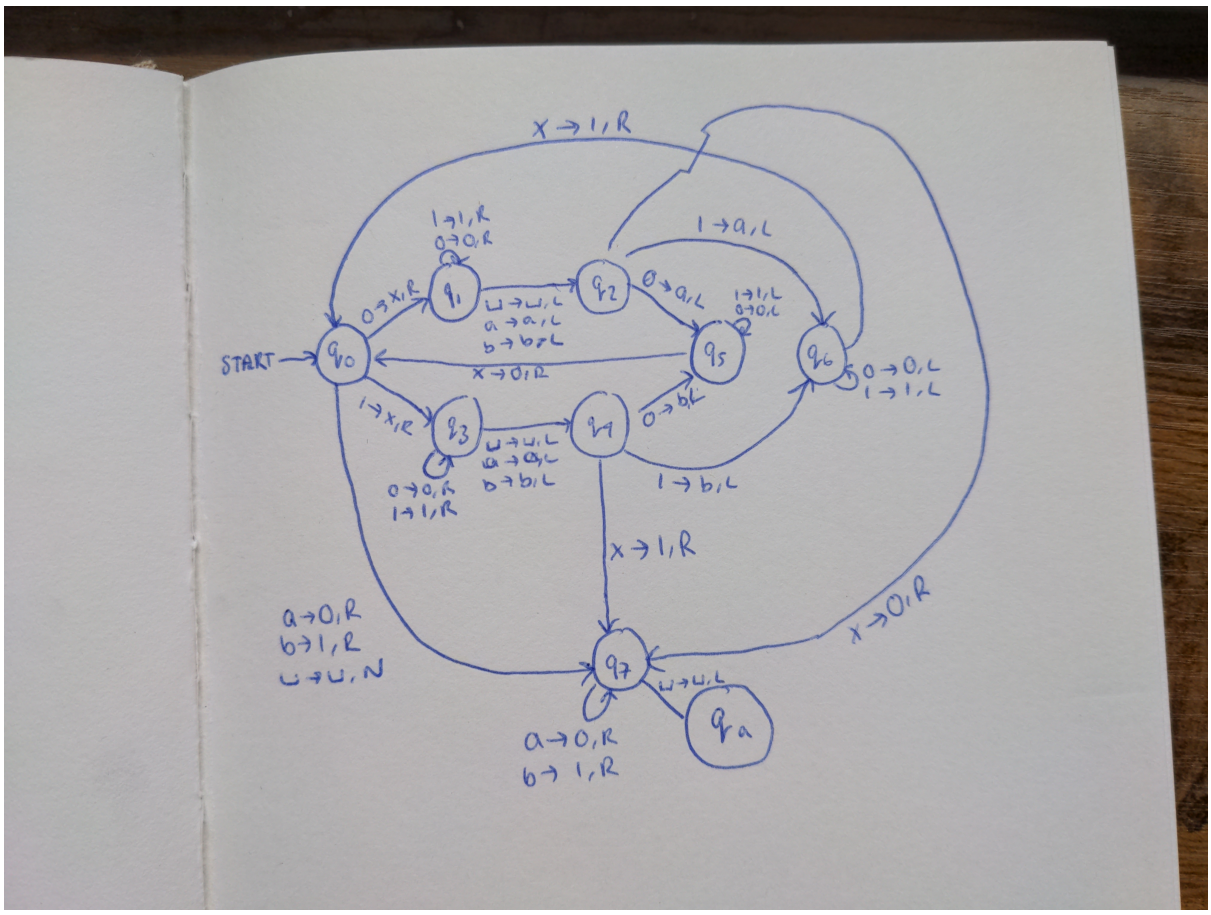

Figure 1: State diagram representing a single-head TM which reverses the

a) Figure 1 Gives a Turing Machine which reverses the order of a binary string (one made up of 0's and 1s.

   It does this by initially looking at the string, and branches off depending on whether a 0 or a 1 is encountered.

   If a 0 is encountered, it goes to state q_1 which skips all 0's and 1's, until it comes to the end of the string (the first space character), where it goes into q_2, and goes back to the final character in the string.

If a 1 is encountered, it goes into state q_3, which also skips all 0's and 1's encountered, till it gets to the end of the string, then goes left to the final character and goes into state q_4.

From here, the state is either q_2 or q_4, where the machine checks for either a 1 or 0. Regardless of the input, q_2 will replace it with an "a", and q_4 will replace it with a "b". However, at either state the input is "0", it goes left and into state q_5. If the input is "1", it goes left and into state q_6.

q_5 and q_6 both skip all 0's and 1's, going left till it reaches the initial x at the start of the string. When reached, state q_5 replaces the x with 0, and q_6 replaces the x with 1, both then go right and transition into q_0 again.

From here, the process repeats, replacing the next character with "x", splitting into states q_1 or q_3, and then going to the end of the string, where either an a or b is found, which are treated the same as the space character.

This leads to the first half of the string then being the reversal of the second half of the string, and the second half the reversal of the first half, but in "a" and "b" characters. Eventually, either "a","b" or a space will be encountered in q_0, meaning the swap is complete, or the "x" will be encountered in state q_2 or q_4 when replacing the last character with the first. The former occurs when the string has an even number of characters, the latter when there is an odd number.

If the length is even and q_0 reaches a space character, it means the string was empty as there is no second half of the string, meaning the swap is done. If "a" or "b" is encountered, it goes into state q_7 where the swapping of the a's into 0's and b's into 1's is done by looping through until a space is reached, where it is done.

If the length is odd, then the x is replaced by the value it was originally (0 if in state q_2 or 1 if q_4. It is then at the start of the second half of the string which is just a's and b's, and so it can enter q_7, doing the same looping steps as described previously.

b) This Turing Machine works by going from the first 0 or 1 character in the string s to the last 0 or 1 character, then back. This repeats, where the first and last 0 or 1 characters converge to the central character, shrinking on the left and right by 1 each time a full loop is complete.

Therefore, this algorithm starts by looping through each character front to back taking $2n$ time. The next loop takes $2n-2$, then $2n-4$ and so on till $2n-a \leq 1$. This will take $\frac{n}{2}$ of these loops, which will take on average n time as you go from $2n$ passes to 0 passes, decreasing each time by 2.

Therefore, we have a time complexity for this part of $n * \frac{n}{2}$ time which is $\frac{n^2}{2}$.

After this, another loop over the second half of the string is required to convert the "a" and "b" characters, taking a single $\frac{n}{2}$ pass.

Therefore, our final time complexity is $\frac{n}{2} + \frac{n^2}{2}$ which can be written $0.5n + 0.5n^2$. As n grows, n^2 dominates both the $0.5n$ and the 0.5 multiplier constant, giving a total asymptotic time complexity of: $\mathcal{O}(n^2)$ time.

2

a) If D_1 and D_2 are in Poly, then it must mean that we can construct machines $T_1(I)$ and $T_2(I)$ which compute D_1 and D_2 respectively with input I in polynomial time.

A problem in poly means it can be ran in n^k time for a positive integer k.

Therefore, we can then construct $T_3(I)$ which computes $T_1(I)$, and sets $v_1 = T_1(I)$, and then computes $T_2(I)$ and sets $v_2 = T_2(I)$. Both of these operations can be done in polynomial time, meaning the time to compute each can be expressed as n_1^k and n_2^k respectively.

It then checks to see if v_1 and v_2 are equal to "yes", returning "yes" if they both are, or "no" otherwise. This can be done in constant time, as the values are always "yes" or "no".

Therefore, the time complexity of this algorithm will be $n_1^k + n_2^k + c$, but the constant is overridden by the polynomial times giving $n_1^k + n_2^k$ time, which can be written $n^{k_1+k_2}$. $k_1 + k_2$ is going to be some non-negative integer k_3, and so the time taken to compute T_3 is n_3^k time which is polynomial.

Computing T_3 is clearly the same as computing BOTHD1D2.

QED.

b) Being in polycheck means that proposed solutions to STRINGCUTTING can be verified in polynomial time.

Given a solution T along with the input, we can verify whether or not the solution is valid by subtracting each value t from L, and the comparing that value to M_1 and then M_2.

The first vale

c) The requirements for NP-completeness is first, that the problem is in polycheck, which has been proved above, as being NP is equivalent to being in polycheck, with the added requirement that the problem is a d decision problem. As problems STRINGNCUTTING either have a subset T meeting the criteria or not, there are only two possible scenario's: either it returns "yes", or it doesn't, and so is a decision problem.

Next, we need to prove that there is a polyreduction to another NP-complete problem. This is because to be NP-complete, for all problems $\mathcal{D} \in$ NP, the problem must be harder than all other decision problems, meaning problems are the hardest of all of the decision problems, as th

d) For the first argument, there is something wrong with the logic because the definition of NP not that the problem is solvable in polynomial time, but instead that solutions to the problem can be verified in polynomial time.

The problem with the second argument is that solving the problem in $\mathcal{O}(n^{\log n})$ time is not polynomial.

We can write $n^{\log n}$ as $(a^{\log a(n)})^{\log n}$

3

a) (i) Here, $\log_c(b) = \log_3(2) =$

(ii)

b) The first implementation has an amortized time complexity of $\mathcal{O}()$.

This is because it always has the same steps for each addition. First, new memory needs to be allocated with the size of the new array. Second, the previous array needs to be copied into the new array. Finally, the old array needs to be deleted from memory.

Therefore, an average of $n/2$ time is taken to look at the array to get it's size.

Then another $n/2$ average time is taken to copy the allocate the array.

Finally, an average of $n/2$ time is taken to copy the array over to the new position.

Therefore, an average of $\frac{3n}{2}$ time is taken.

Therefore, the size of the new array is required, which will take n time, and then allocating takes n time, and then the allocation

The second implementation has an amortized time of $\mathcal{O}(1)$. This is because there is a varying number of steps for each addition, because the new array memory space needs to be claimed, and contents copied over, only when the old array does not have enough space.

The number of doubling operations k that are required increases with n, but as the size doubles, the size exponentially grows. This means that k increases at a decreasing rate as n grows.

As the size is stored, the size does not need to be calculated each time, instead just a constant time size of the next element needs to be calculated and added to the size.

As n grows, there are less and less doubling operations required, meaning the larger n grows, the more operations just requires copying the singular value to the array, which is a constant time operation, which decreases the average time taken for each addition more and more. As n tends towards infinity, the average time taken tends towards the constant time of adding to a not-full array, without the allocation or copying, giving an amortized time complexity of $\mathcal{O}(1)$