Theory of Computation - 20585

Maman 15

Yehonatan Simian
206584021

Question 1 (10%)

Prove: $\lfloor \sqrt{n} \rfloor$ is space constructible.

PROOF First, let's define the input representation. We assume that the input n is given in binary notation, where the length of the input is $|n|$, the number of bits required to represent n.

In order to prove this theorem, we shall define a Turing machine M that uses a variation of the *binary search algorithm* in order to compute $\lfloor \sqrt{n} \rfloor$ using a space bounded by a polynomial in the input length $|n|$. The algorithm is rather simple:

```
M = " On input ⟨n⟩ where n is a natural number:
    1. Initialize two variables, $lo \leftarrow 0$ and $hi \leftarrow n$.
    2. While $lo$ is not greater than $hi$:
    3.     Let $mid \leftarrow \frac{lo+hi}{2}$.
    4.     If $mid \times mid$ is greater than $n$, then $hi \leftarrow (mid - 1)$.
    5.     Otherwise $lo \leftarrow (mid - 1)$.
    6. Return $hi$. "
```

The idea behind this algorithm is to find the largest integer whose square is less than or equal to n. By adjusting the search range based on the comparison of $mid \times mid$ with n, we gradually narrow down the search space until we find the largest integer whose square is less than or equal to n.

The binary search algorithm requires $\log(n)$ iterations to converge, as each iteration halves the search range. Within each iteration, we perform basic arithmetic operations (multiplications and comparisons) on integers which can be implemented in logarithmic space, hence the space complexity of this algorithm is logarithmic in the input size. Since $\log(n)$ is a polynomial in the input size $|n|$, we can conclude that $\lfloor \sqrt{n} \rfloor$ is space constructible function. ■

Question 2 (14%)

Section 1

Replacing ”Simulate M on w” with ”Simulate M on $\langle M\rangle$” **won’t** keep the proof valid.

PROOF A quote from Theorem 9.3’s proof idea: “D guarantees that A is different from any language that is decidable in $o(f(n))$ space”. This quote is the key idea to our proof; When simulating M on $\langle M\rangle$ instead of w, M and D might behave the same on every input. It is possible that they both reject $\langle M\rangle$ and accept w, and that M needs more time to reject $\langle M\rangle$ than the time D can run $\langle M\rangle$. In that case, D will *reject* $\langle M\rangle$ as a timeout exception, but it will *accept* w because M rejects $\langle M\rangle$, in contrast to the theorem’s assumption. ■

Section 2

Replacing ”Simulate M on w” with ”Simulate M on 10^k” **won’t** keep the proof valid.

PROOF This proof’s idea is the same as Section 1’s. When simulating M on 10^k instead of w, M and D might behave the same on every input. It is possible that they both reject 10^k and accept w. In that case, M will *reject* 10^k, hence D will *accept* w, in contrast to the theorem’s assumption. ■

Question 3 (12%)

Explain how can a double-taped Turing machine, that runs in a linear time complexity, convert an unary number 1^n (given on the first tape which is read-only) to its binary representation (will be written on the second tape which is read-write tape).

PROOF IDEA Our Turing machine will initialize a variable to 0 and increment its value (by 1) n times. The idea itself is rather simple, but a non-trivial mathematical proof is required to prove that this algorithm runs in linear time and not quadratic time complexity.

PROOF First of all, let's construct the Turing machine itself:

```
M = " On input ⟨n⟩ where n is a natural number in unary representation:
    1. Initialize variable $a \leftarrow 0$ and write it on the second tape.
    2. While there is a 1 symbol to the right of the first tape's head:
    3.     Move the first tape's head right.
    4.     Calculate $a \leftarrow a + 1$ and write the result on the second tape (override $a$).
    5. Return $a$. "
```

At worst case, the algorithm takes $O(n^2)$ time complexity - which means replacing *every symbol* on the second tape for each symbol read on the first tape. I will show now why this case never happens, and $O(n)$ is the actual time complexity even at worst case.

- Half of the executions of step #3 are done when a's binary representation ends with 0. Thus, only one symbol was replaced in a's binary representation.
- Quarter of the executions of step #3 are done when a's binary representation ends with 01. Thus, only two symbols were replaced in a's binary representation.
- Eighth of the executions of step #3 are done when a's binary representation ends with 011. Thus, only three symbols were replaced in a's binary representation.

This pattern leads to $\sum_{i=1}^{n}\left(\frac{1}{2^i}\cdot n\cdot i\right)$ being the amount of steps needed for the entire algorithm, where $\frac{1}{2^i}$ is the portion of executions of step #3 as shown in the list above, n is the total number of executions of step #3, and i is the number of steps done in each execution portion. We can pull n out of the sum and find that the number of steps is $n\cdot\sum_{i=1}^{n}\left(\frac{1}{2^i}\cdot i\right) = n\cdot 2 = 2n$.

NOTE The sum $\sum_{i=1}^{n}\left(\frac{1}{2^i}\cdot i\right)$ *converges to* 2 *as taught in calculus lessons. Since this is not a calculus course, I'll skip the proof.*

To summarize, I've explained with details how can a double-taped Turing machine convert an unary number to its binary representation using a linear time complexity, so I'm officially allowed to put that little square symbol at the end of this sentence. ■

Question 4 (10%)

Show that every $n \in \mathbb{N}^+$ has an undirected graph $G = (V, E)$ such that:

- $|V| = 2n$
- $\exists$ Minimal vertex cover $U \subseteq V$ such that $|U| = n$
- The algorithm A will find a minimal vertex cover of size $2n$.

PROOF Let $G = (V, E)$ where $V = s_1, s_2 \ldots s_n, t_1, t_2 \ldots t_n$ and $E = \{\{s_1, t_1\}, \{s_2, t_2\} \ldots \{s_n, t_n\}\}$. Let $S = s_1, s_2 \ldots s_n, T = t_1, t_2 \ldots t_n$. Notice that S, T are two disjoint and independent sets, and every edge connects a vertex in S to one in T, hence and by definition it is a *bipartite graph*. Let $U = S$.

Surely $|V| = 2n$, and U is a minimal vertex cover of size n because $\forall \{s_i, t_i\} \in E$, $s_i \in U$. All that is left to prove is that A will find a cover of size $2n$. Let's remind ourselves A's behaviour:

```
A = " On input ⟨G⟩ where G is an undirected graph:
    1. Repeat the following until all edges in G touch a marked edge:
    2.     Find an edge in G untouched by any marked edge.
    3.     Mark that edge.
    4. Output all nodes that are endpoints of marked edges. "
```

Since G is a bipartite graph, each edge that will be chosen on step #2 will add exactly two nodes that will be outputted on step #4, i.e. $\forall 1 \leq i \leq n$ the edge $\{s_i, t_i\}$ will be chosen exactly once, adding the two nodes s_i, t_i to the nodes outputted on step #4 as for they haven't been touched before by any other edge.

We've shown that all of G's nodes will be outputted by A, meaning that A will always find a minimal vertex cover of size $2n$. ∎

Question 5 (26%)

Prove: if P=NP, then exists a polynomial time algorithm that gets as input a monotonic formula φ and a natural number k, and outputs a satisfying assignment of φ where k atoms assigned with 1, if such assignment exists.

PROOF IDEA In each step of the algorithm, we'll use the Turing machine from Maman 13 Question 8 in order to know if there's an assignment that satisfies the requirements. If there is such assignment, then we'll take one atom from the formula, assign it with 0 or 1, remove it from the formula, and recursively call our algorithm in order to find the rest of the assignment. Since we know that there's an assignment that satisfies the requirements, one of that atom's assignments (either 0 or 1) must be in the entire formula's assignment.

Solution Let M_8 be the Turing machine from Maman 13 Question 8.

```
$M$ = " On input $\langle \varphi, k \rangle$ where $\varphi$ is a monotonic formula, and $k \in \mathbb{N}$:
     1. Run $M_8$ on $\langle \varphi, k \rangle$. If rejected, print "no" and terminate the execution.
     2. If $\varphi$ is empty:
     3.    If $k = 0$, return an empty assignment.
     4.    Else, print "no" and terminate the execution.
     5. Let atom $x \in \varphi$. Non-deterministicly choose an assignment, either $x \leftarrow 0$ or $x \leftarrow 1$.
     6. Create a new formula $\varphi'$ that does consist of $\varphi$ without $x$ in the following way:
     7.    If $x$ was assigned with 0:
             -   Each conjunction $c = (x \wedge y \wedge \cdots \wedge z)$ will be removed from $\varphi'$.
             -   Each disjunction $d = (x \vee y \vee \cdots \vee z)$ will become $d' = (y \vee \cdots \vee z)$ in $\varphi'$.
     8.    Run $M$ on $\langle \varphi', k \rangle$, and return the received assignment with $x \leftarrow 0$.
     9.    Else, if $x$ was assigned with 1:
             -   Each conjunction $c = (x \wedge y \wedge \cdots \wedge z)$ will become $d' = (y \vee \cdots \vee z)$ in $\varphi'$.
             -   Each disjunction $d = (x \vee y \vee \cdots \vee z)$ will be removed from $\varphi'$.
    10.    Run $M$ on $\langle \varphi', k-1 \rangle$, and return the received assignment with $x \leftarrow 1$. "
```

NOTE In steps #8 and #10 we assert that an assignment must be received, as explained in the proof idea. This assertion is "legal" due to the check that has been made on step #1, and without it the entire algorithm wouldn't be correct.

PROOF The correctness of the algorithm come directly from the rules of logic, assignments, conjunctions and disjunctions, so I'll spare this part because it is not a logic class. Let's focus on the time complexity of this algorithm, by analyzing each step's time complexity:

- Step #1 runs in a polynomial time, as part of the proof of Maman 13 Question 8, and the assumption that P=NP.
- Steps #2 to #4 run in constant time, because they do not rely on the input's length.

Question 5 continued on next page...

- Step #5 runs in constant time as well, due to the assumption that P=NP.
- Steps #6, #7, and #9 all run in $O(|\varphi|)$, i.e. polynomial in the input's length, because $|\varphi'| \leq |\varphi|$ due to the removal of x from the formula.
- Steps #8, #10 are just a recursive call to M, hence run in polynomial time. Note that only **one** of those steps (either #8 or #10) will be called at each execution of M, and that's why the time complexity of this algorithm is polynomial and not exponential (as if it was if both steps #8 and #10 would be called one by one).

We've shown that each step of the algorithm runs in polynomial time, hence the entire algorithm runs in polynomial time complexity. We found an algorithm that satisfies the question's requirements, therefore such algorithm exists. ■

Question 6 (20%)

Let languages A, B where $B \in \text{BPP}$. Let M a polynomial time Turing machine with input w, such that:

- If $w \in A$, then at the finish of M's execution, $\frac{3}{4}$ of M's paths write on the tape a word $w_1 \in B$, and $\frac{1}{4}$ of M's paths write on the tape a word $w_1 \notin B$.
- If $w \notin A$, then at the finish of M's execution, $\frac{2}{3}$ of M's paths write on the tape a word $w_1 \in B$, and $\frac{1}{3}$ of M's paths write on the tape a word $w_1 \notin B$.

Prove: $A \in \text{BPP}$.

Solution Let's start with a polynomial time Turing machine that accepts A:

```
$M_A$ = " On input $\langle w \rangle$ where $w$ is a string:
    1. Run $M$ on $w$. Run $M_B$ on the result. If $M_B$ rejected, reject.
    2. Run $M$ on $w$. Run $M_B$ on the result. If $M_B$ rejected, reject. Otherwise, accept. "
```

NOTE $B \in BPP$, hence we assume the existence of a polynomial Turing machine M_B that accepts B with an error probability of $\frac{1}{3}$, so we can use it to determine whether a string is in B.

PROOF According to page 369 in the book, ***M recognizes language A with error probability ϵ if***

1. $w \in A$ implies $\Pr[M \text{ accepts } w] \geq 1 - \epsilon$, and
2. $w \notin A$ implies $\Pr[M \text{ rejects } w] \geq 1 - \epsilon$.

Let's focus on $\Pr[M_A \text{ accepts } w]$. In order for M_A to accept w, M must accept w twice. Since each run of M on w in independent with the other run, the probability that M_A accepts w is $\Pr[M_A \text{ accepts } w] = \Pr[M_B \text{ accepts } w_M]^2$, where w_M is the result on M's run on w.

Let's continue, this time we'll focus on $\Pr[M_B \text{ accepts } w_M]$. Since we know the probabilities to the outcomes of M in terms of acceptance by M_B, we can deduce that:

1. $w \in A$ implies
$$\begin{aligned}\Pr[M_B \text{ accepts } w_M] &= \Pr[w_M \in B] \cdot (1-\epsilon) + \Pr[w_M \notin B] \cdot \epsilon \\ &= \frac{3}{4} \cdot (1-\epsilon) + \frac{1}{4} \cdot \epsilon \\ &= \frac{3}{4} - \frac{\epsilon}{2}\end{aligned}$$
2. $w \notin A$ implies
$$\begin{aligned}\Pr[M_B \text{ accept } w_M] &= \Pr[w_M \in B] \cdot (1-\epsilon) + \Pr[w_M \notin B] \cdot \epsilon \\ &= \frac{2}{3} \cdot (1-\epsilon) + \frac{1}{3} \cdot \epsilon \\ &= \frac{2}{3} - \frac{\epsilon}{3}\end{aligned}$$

After calculating those probabilities, it's time for $\Pr[M_A \text{ accepts } w]$ again. Look and learn!

Question 6 continued on next page. . .

1. $w \in A$ implies $\Pr\left[M_A \text{ accepts } w\right] = \Pr\left[M_B \text{ accepts } w_M\right]^2$
$$= \left(\frac{3}{4} - \frac{\epsilon}{2}\right)^2$$
$$= \frac{9}{16} - \frac{3}{4} \cdot \epsilon + \frac{1}{4} \cdot \epsilon^2$$

2. $w \notin A$ implies $\Pr\left[M_A \text{ rejects } w\right] = 1 - \Pr\left[M_B \text{ accepts } w_M\right]^2$
$$= 1 - \left(\frac{2}{3} - \frac{\epsilon}{3}\right)^2$$
$$= \frac{5}{9} + \frac{4}{9} \cdot \epsilon - \frac{1}{9} \cdot \epsilon^2$$

Now, let's see what values of ϵ yield a result that is greater than $1 - \epsilon$:

$$\begin{aligned}
&\frac{5}{9} + \frac{4}{9} \cdot \epsilon - \frac{1}{9} \cdot \epsilon^2 \geq 1 - \epsilon \\
\iff &\frac{1}{9} \cdot \epsilon^2 - \frac{13}{9} \cdot \epsilon + \frac{4}{9} \geq 0 \\
\iff &\epsilon^2 - 13\epsilon + 4 \geq 0 \\
\iff &\epsilon \geq \frac{13 - \sqrt{13^2 - 16}}{2} \approx 0.315
\end{aligned}$$

This means that every choice of $0.315 \leq \epsilon < 0.5$ should satisfy the requirement for M_A to recognize A. Let's choose $\epsilon = \frac{1}{3}$, then M_A recognizes A by definition.

One thing I forgot to mention - M_A runs in polynomial time because it only runs M and M_B, who both run in polynomial times according to their definitions. We've found a probabilistic polynomial time Turing machine with an error probability of $\frac{1}{3}$ that recognizes A, therefore $A \in \text{BPP}$. ■

Question 7 (8%)

This question is about the algorithm *PRIME* from the book.
Prove: if $t < p \in \mathbb{N}$ and they are not coprime, then t is p's *compositeness witness*.

PROOF First of all, let's remember the definition of *witness*: "a_i is a ***(compositeness) witness*** if the algorithm rejects at either stage 4 or 7, using a_i." This means the algorithm must fail on either:

4. Compute $a_i^{p-1} \mod p$ and *reject* if different from 1.

7. If some element of $a_i^{s\cdot 2^0}, a_i^{s\cdot 2^1}, \ldots, a_i^{s\cdot 2^h}$ modulo p is not 1, find the last element that is not 1 and *reject* if that element is not -1.

Back to our proof. p and t's are not coprime, hence p and every power of t are not coprime as well. Let's assume that $t^{p-1} \mod p = 1$. Then $\exists k \in \mathbb{N}$ such that $t^{p-1} = kp + 1$, hence $t^{p-1} - kp = 1$. Since the coefficients on the right-hand-side are t and p, this entire side is divided by their gcd, which is greater than 1 according to them being coprime. Hence the left-hand-side must be divided by that gcd, but this side consist of the number 1, therefore that gcd is 1, in contradiction to them being coprime.

We found a contradiction, hence $t^{p-1} \mod p \neq 1$, therefore stage 4 of the algorithm will *reject*, thus t is p's witness. ∎