COMPUTATIONAL COMPLEXITY HW-1

SIDDHANT CHAUDHARY, AMIK RAJ BEHERA
BMC201953 , BMC201908

Problem 1. Prove that any finite set of strings belongs to **DTIME**(n).

Solution. Suppose L is any finite set of strings. The basic idea of our polynomial time TM will be to enumerate each string of L and write it on its tape, and check whether the given input is equal to one of those strings. So, make a TM M that does the following.

(1) On input x, enumerate each string of L and write each string on the tape.
(2) Check whether x is equal to one of the strings written on the tape. Accept if x is equal to one of the strings, otherwise reject.

Step (1) takes constant time, because enumerating a (fixed) number of strings on the tape takes constant time. Step (2) takes time $O(n)$ (where $n = |x|$), because we just need to scan the input to check equality with one of the strings. So, this machine is a DTM that runs in time $O(n)$, and hence $L \in$ **DTIME**(n). ■

Problem 2. Prove that if there is a polynomial time algorithm that converts a CNF formula to a DNF formula preserving the satisfiability, then **P** $=$ **NP**.

Solution. First, suppose we have a DNF formula ϕ. We claim that ϕ is satisfiable if and only if each clause of ϕ *does not* contain both x and $\neg x$, where x is some variable. To show this, observe that ϕ is satisfiable if and only if atleast one clause of ϕ is satisfiable (because ϕ is an OR of ANDs). Now, take any clause C of ϕ. We have

$$C = k_1 \wedge k_2 \wedge \cdots \wedge k_m$$

where each k_i is a literal and $m \in \mathbb{N}$ is an integer. Now, by assigning each literal $k_1, ..., k_m$ a value of 1, we can satisfy C, and this works if and only if C does not contain both x and $\neg x$ for some variable x. This proves our claim.

So, to check whether a formula ϕ in DNF is satisfiable or not, we only need to check the existence of x and $\neg x$ inside a clause for some variable x. This can clearly be done in linear time. This means that DNF is in **P**. But, this means that CNF-SAT is in **P**, and hence this implies that **P** $=$ **NP**, because CNF-SAT is **NP**-complete. This completes our proof. ■

Problem 3. Show that if **P** $=$ **NP**, then every language $A \in$ **P** such that $A \neq \phi$ and $A \neq \Sigma^*$ is **NP**-complete. Explain why ϕ and Σ^* can never be **NP**-complete.

Solution. Suppose **P** $=$ **NP**, and let $A \in$ **P** such that $A \neq \phi$ and $A \neq \Sigma^*$. Clearly, we see that $A \in$ **NP**. Now, let $x, y \in \Sigma^*$ be such that $x \in A$ and $y \notin A$, and fix these x, y. Let $B \in$ **NP**, which means that $B \in$ **P**. Define a map $f : \Sigma^* \to \Sigma^*$ as follows.

$$f(s) = \begin{cases} x & , \text{ if } s \in B \\ y & , \text{ otherwise} \end{cases}$$

Date: February 15 2021.

Clearly, this is a Karp-reduction from B to A. f is also polynomial time computable, and the TM to compute f works as follows:

(1) On input s, use the polynomial time TM for B to check whether $s \in B$ or $s \notin B$. If $s \in B$, output the word x on the output tape. If $s \notin B$, output the word y on the output tape.

Clearly, the running time of the above TM is polynomial, and hence f is polynomially computable. So, it follows that $B \leq_P A$, and hence A is **NP**-complete.

Now, we show that ϕ and Σ^* can *never* be **NP**-complete. Consider the language ϕ, and the proof for Σ^* is similar. Suppose ϕ is **NP**-complete. This would mean that all problems $A \in$ **NP** are polynomial Karp-reducible to ϕ, i.e there is some polynomially computable function $f : \Sigma^* \to \Sigma^*$ such that

$$x \in A \iff f(x) \in \phi \quad , \quad \forall x \in \Sigma^*$$

However, this implies that $A = \phi$. However, we know that there are non-empty languages that are in **NP**, for instance, we can take any finite language. A similar reason shows why Σ^* cannot be **NP**-complete. This completes the proof. ■

Problem 4. Let $S = \{\psi \,|\, \psi$ is Satisfiable 3CNF formula$\}$. Suppose we have a deterministic poly-time Turing machine M_S for deciding S. Describe a deterministic poly-time Turing machine M that given a 3CNF formula ϕ can write the satisfying assignment for ϕ on its output tape (using M_S).

Solution. Let $\phi = \phi(x_1, \dots, x_n)$. The following algorithm describes a deterministic poly-time Turing machine M which will write the satisfying assignment for ϕ (if ϕ is a satisfying assignment) on it's output tape:

Algorithm 1 WRITE-SATISFYING-ASSIGNMENT (ϕ)

```
Run M_S on φ
if φ is a satisfying assignment then
  for i ∈ [1...n] do
    Assign x_i = 1 and then run M_s on φ
    if M_S returns true then
      Write 1 on i^th cell of the output tape and assign x_i = 1 permanently
    else
      Write 0 on i^th cell of the output tape and assign x_i = 0 permanently
    end if
  end for
else
  No satisfying assignment
end if
```

The above algorithm will terminate because there are exactly n iterations of the *for loop*, and in each iteration, M_S is being called, which is a terminating algorithm. It is deterministic because M_S is deterministic. Clearly, this is a poly-time algorithm since it invokes M_S exactly n times, and M_S itself is a poly-time algorithm. ■

Problem 5. A language L is said to be a *unary language* if $L \subseteq \{1\}^*$. Prove that if all unary languages in **NP** are also in **P**, then **EXP** $=$ **NEXP**.

Proof. Suppose all unary languages in **NP** are also in **P**. Now, let $L \in$ **NEXP**, and let N be an $O(2^{n^c})$ time NDTM deciding L, where c is some positive integer. Without loss of generality, suppose on input x, the machine N halts in $2^{|x|^c}$ steps. Now, consider the language

$$L' := \{1^{x10^y} \mid x \in L,\, y \geq 2^{|x|^c}\}$$

(here $1^{x10^y} = 1^m$, where m is the integer whose binary representation is $x10^y$) and clearly, L' is a unary language. Observe that

$$(*) \qquad x \in L \iff 1^{x10^{2^{|x|^c}}} \in L'$$

and we will be using this equivalence below.

We now show that L' is in **NP**. To see this, consider the following algorithm for L':

(1) On input 1^m, first check whether $m = x10^y$ for some $x \in \{0,1\}^*$ and some y. If not, then simply reject.
(2) If m is of the form $m = x10^y$, then run the NDTM N on x for atmost y steps. If N halts within this time, then return N's answer, otherwise reject.

Clearly, this algorithm is a polynomial time algorithm, because step (2) runs for atmost y steps, and y is part of our input. Hence we see that $L' \in$ **NP**. But by our assumption, we know that $L' \in$ **P**, i.e there is some DTM M deciding L' in polynomial time.

So, we can give a $O(2^{n^c})$ time DTM M' for the language L. The algorithm is as follows.

(1) On input x, generate the string $x10^{2^{|x|^c}}$. This takes time $O(2^{|x|^c})$. Write it somewhere on M's tape.
(2) Now, run the machine M on the input $1^{x10^{2^{|x|^c}}}$, and return M's answer.

Observe that step (2) runs in polynomial time on the length $|x|+1+2^{|x|^c}$, which is exponential in $|x|$. Hence, it follows that M' is a $O(2^{n^c})$ time DTM, and the equivalence in $(*)$ shows that M' accepts the language L. So we have shown that **NEXP** $\subseteq$ **EXP**, and hence this shows that **EXP** $=$ **NEXP**, and this completes our proof. ∎

Problem 6. Prove or disprove: A language L is **NP**-complete iff L^c is **coNP**-complete.

Solution. We first show that if L is **NP**-complete, then L^c is **coNP**-complete as follows:

(1) $L \in$ **NP** $\Rightarrow L^c \in$ **coNP**.
(2) For all $A \in$ **NP**, let f_A be the reduction of A to L.
(3)

$$x \in A \Leftrightarrow f(x) \in L$$
$$\Rightarrow x \notin A \Leftrightarrow f(x) \notin L$$
$$\Rightarrow x \in A^c \Leftrightarrow f(x) \in L^c$$

Thus for all $A^c \in$ **coNP**, $A^c \leq L^c$. Hence L^c is **coNP**-complete.

Now we show that if L^c is **coNP**-complete, then L is **NP**-complete as follows:

(1) $L^c \in$ **coNP** $\Rightarrow L \in$ **NP**.

(2) For all $A^c \in$ **coNP**, let f_{A^c} be the reduction of A^c to L^c.
(3)

$$x \in A^c \Leftrightarrow f(x) \in L^c$$

$$\Rightarrow x \notin A^c \Leftrightarrow f(x) \notin L^c$$

$$\Rightarrow x \in A \Leftrightarrow f(x) \in L$$

Thus for all $A \in$ **NP**, $A \leq L$. Hence L is **NP**-complete. ■

Problem 7. Prove or disprove the following statements:

(1) If $L_1, L_2 \in$ **NP** , then $L_1 \cup L_2 \in$ **NP** and $L_1 \cap L_2 \in$ **NP**.

Solution. Let $B_1(x, w)$ be the verifier which checks whether x is a yes-instance of L_1 with certificate w, and similarly let $B_1(x, w)$ be the verifier which checks whether x is a yes-instance of L_1 with certificate w. Now we define a verifier $B_3(x, w)$ for $L_1 \cup L_2$ as follows:

$$B_3(x, w) = B_1(x, w) \vee B_2(x, w).$$

Note that B_3 is in **P** because both B_1 and B_2 are in **P**. Since $B_3(x, w)$ is a yes-instance if and only if atleast one of the $B_1(x, w)$ or $B_2(x, w)$ is a yes-instance. In other words, x is a yes-instance of $L_1 \cup L_2$ if and only if it is a yes-instance of atleast one of L_1 and L_2.

Similarly, we define a verifier $B_3(x, w)$ for $L_1 \cap L_2$ as follows:

$$B_3(x, w) = B_1(x, w) \wedge B_2(x, w).$$

Note that B_3 is in **P** because both B_1 and B_2 are in **P**. Since $B_3(x, w)$ is a yes-instance if and only if both of the $B_1(x, w)$ or $B_2(x, w)$ are yes-instances. In other words, x is a yes-instance of $L_1 \cap L_2$ if and only if it is the yes-instance for both L_1 and L_2. Hence we have showed that both $L_1 \cup L_2$ and $L_1 \cap L_2$ are in **NP**. ■

(2) Let L be an **NP**-complete problem. If $L \in$ **NP** and $L^c \in$ **NP**, then **NP** = **co-NP**.

Solution. Since $L^c \in$ **NP**, this means that $L \in$ **coNP** and therefore, $L \in$ **NP** $\cap$ **coNP**. Similarly, $L^c \in$ **NP** $\cap$ **coNP**. From **Problem 6**, we get that L^c is **coNP**-complete. This means that for all $A \in$ **coNP**, there is a poly-time reduction from A to L^c, but $L^c \in$ **NP**. Thus for all $A \in$ **coNP**, we have $A \in$ **NP**. Therefore, **coNP** $\subseteq$ **NP**. Similarly, we can show that **NP** $\subseteq$ **coNP**. Hence **NP** = **coNP**. ■

(3) **P** $\subseteq$ **NP** $\cap$ **coNP**

Solution. We already know that **P** $\subseteq$ **NP**. Now let $L \in$ **P**. Then we can decide whether $x \notin L$ simply by running x on L and in poly-time we will get to know whether $x \notin L$. Thus $L \in$ **coNP**. This leads to **P** $\subseteq$ **coNP**. Hence we get that **P** $\subseteq$ **NP** $\cap$ **coNP**. ■

Problem 8.1. For any function $f(n) \geq \log n$, show that

$$\textbf{NSPACE}(f(n)) = \textbf{coNSPACE}(f(n))$$

Proof. Let M be a non-deterministic Turing Machine which solves problems in **NSPACE**$(f(n))$ is $O(f(n))$ space. Let $G = (V, E)$ be the configuration graph of M on input w, with $|V| = 2^{O(f(n))}$. To show that **NSPACE**$(f(n))$ = **coNSPACE**$(f(n))$, we have to show that if $L \in$ **NSPACE**$(f(n))$, then $L^c \in$ **NSPACE**$(f(n))$. $L \in$ **NSPACE**$(f(n))$, means there exists a path from $s \in V$ to $t \in V$, where s is the initial configuration and t is the accepting configuration. On the other hand, $L^c \in$ **NSPACE**$(f(n))$ means: it can be verified that there exists no path from s to t in $O(f(n))$ space. Let $count_k$ denotes the number of vertices which have a path from s of length at most k. Note that $count_1 =$ deg$(s) + 1$. We can compute $count_{i+1}$ from $count_i$ by guessing a vertex u, and guessing a path from s to u of length at most i. If there exists a path from s to u of length at most i, we count all the neighbours of u as vertices who have a path from s of length at most $i+1$. This sub-routine can be done using log$(2^{O(f(n))}) = O(f(n))$ space, because we only need to keep track of two vertices at one time and then we can re-use the space (the same idea which we used in proving **NL** = **coNL**). Now we define an algorithm to determine whether a vertex v has a path from s of length at most k:

Algorithm 2 REACH-IN-k(G, s, v, k)

```
for each u ∈ V − v do
  bool = Guess whether there is a path from s to u of length at most k
  countcheck = countcheck + bool {bool = 0 when false and 1 when true}
  if bool = true then
    Guess a path from s to u of length at most k
    if the path doesn't reach u then
      return reject
    end if
  end if
end for
if countcheck = count then
  return reject
end if
if countcheck = count - 1 then
  Guess a path from s to v
  if a path of length at most k exists then
    return accept
  end if
else
  return reject
end if
```

Again, note that this algorithm can decide whether v is reachable from s in at most k steps by using $O(f(n))$ space. Now we can check whether there exists any path from s to t as follows:

Algorithm 3 REACHABILITY-OF-t

```
for i in range |V| do
    Bool = REACH-IN-k(G, s, v, i)
    if Bool = true then
        reject
    else
        accept
    end if
end for
```

Thus we have showed that we $L^c \in$ **NSPACE**$(f(n))$, which implies that $L \in$ **coNSPACE**$(f(n))$. Thus **NSPACE**$(f(n))$ = **coNSPACE**$(f(n))$. ∎

Problem 8.2. Is $\mathbf{EXP}^{\mathbf{EXP}} = \mathbf{EXP}$? Justify your answer.

Solution. No, the given inequality is not true, and we now show this. First, observe that

$$2^{n^c} \log n = o\left(2^{2^{n^c}}\right)$$

Now, we know that **EXP** is the class of all those problems that are solvable in $O(2^{n^c})$ time for some constant c. By the **Time-Hierarchy Theorem**, there is some problem in **DTIME** $\left(2^{2^{n^c}}\right)$ that is not in **EXP**. We will now show that **DTIME** $\left(2^{2^{n^c}}\right) \subseteq$ $\mathbf{EXP}^{\mathbf{EXP}}$, and this will show that $\mathbf{EXP}^{\mathbf{EXP}} \neq \mathbf{EXP}$.

So, take any problem $L \in$ **DTIME** $\left(2^{2^{n^c}}\right)$ for some constant c, and let M be an $O\left(2^{2^{n^c}}\right)$ time DTM deciding L. Define the language

$$L' := \{x10^{2^{|x|^c}} \mid x \in L\}$$

Now, observe that $L' \in$ **DTIME**(2^{n^c}), because the following algorithm decides the language L':

(1) On input y, check if y is of the form $y = x10^{2^{|x|^c}}$. If not, then simply reject.
(2) If y is of the form $y = x10^{2^{|x|^c}}$, then run the machine M on the input x. Return M's answer.

Observe that the above algorithm runs in time $2^{|y|^c}$, because $|x| = O(\log|y|)$. So, we have reduced the problem L to a problem in **EXP**, namely L'. So, it follows that $L \in \mathbf{EXP}^{\mathbf{EXP}}$, because we can pad the input exponentially, and call the oracle for L' to decide the language L. This completes the proof. ∎

Problem 9. Σ_2SAT is the following decision problem: Given a CNF formula ϕ, decide whether $\psi = \exists x\, \forall y\, \phi(x, y) = 1$ is true. Show that if **P=NP**, then Σ_2SAT $\in$ **P**.

Solution. Firstly, we will show that Σ_2SAT $\in \Sigma_2^p$. In other words, we will show that

ϕ is in Σ_2SAT $\Leftrightarrow \exists\, x,\ \forall\, y : (\phi, x, y)$ is a yes-instance of B, where B is a decision problem in **P** regarding (ϕ, x, y), and where $\lvert x\rvert + \lvert y\rvert + \lvert\phi\rvert =$ poly$(\lvert\phi\rvert)$.

Clearly $|x|+|y|+|\phi| =$ poly$(|\phi|)$ because x and y are the inputs to ϕ. B is a verifier which takes argument (ϕ, x, y) such that $|x| + |y| =$ no. of variables in ϕ. On the argument (ϕ, x, y), B verifies whether $\phi(x, y)$ is satisfiable or not, which takes poly-time in regards to $|\phi|$. This implies that B runs in poly-time in regards to $|x| + |y| + |\phi|$. Therefore, B is in **P**. Thus we have showed that Σ_2SAT $\in \Sigma_2^p$.

If **P** = **NP**, then it was proven in the class that polynomial hierarchy collapses to **P**, and hence Σ_2^p = **P**, which means that Σ_2SAT is in **P**. ■

Problem 10. Recall the definition of a log-space transducer. A log-space transducer M, which is a Turing Machine, is said to compute a log-space computable function $f : \Sigma^* \to \Sigma^*$ if on running M on input $w \in \Sigma^*$, it writes $f(w)$ on the output tape.

Let M_1 and M_2 be two log-space transducers computing the log-space computable functions f_1 and f_2. Show that there exists a log-space transducer M that computes the function $f : \Sigma^* \to \Sigma^*$ such that $\forall w \in \Sigma^*$, we have $f(w) = f_1(f_2(w))$. In other words, show that the composition of two log-space computable functions is log-space computable.

Solution. First, we make the following observation: there is some polynomial q such that for all $w \in \Sigma^*$, we have

$$|f_2(w)| = q(|w|)$$

i.e the length of $f_2(w)$ is some polynomial of the length of w. This is true because f_2 is a polynomial-time machine; if we have an input w, then there are $|w|$ possibilities for the position of the input head, and there are $O(2^{\log|w|}) = O(|w|)$ possible work tape configurations (because M_2 uses only logarithmic space). So, the total number of configurations is $O(|w|^2)$, which is polynomial in $|w|$. The machine M_2 cannot repeat any configurations, and hence M_2 runs in polynomial time. This immediately implies that the length of $f_2(w)$ is polynomial in the length of w, and that proves our claim.

Now, our log-space transducer for the composition $f_1 \circ f_2$ will work as follows: suppose our input is $w \in \Sigma^*$. We use the transducer M_2 to generate the string $f_2(w)$. Then, we pass the input $f_2(w)$ to the transducer M_1 to generate the string $f_1(f_2(w))$. But there is a catch: since we are only allowed $O(\log|w|)$ space, we *cannot* store the string $f_2(w)$. However, note that M_1 does not need all of $f_2(w)$ to operate, it only needs one bit at a time.

So, we make a tranducer M which works as follows.

(1) On input w, run the transducer M_1 on the string $f_2(w)$; keep track of M_1's input tape head, and each time the transducer M_1 needs the i^{th} bit of $f_2(w)$, run the transducer M_2 on w and generate the i^{th} bit of $f_2(w)$ and pass it to M_1, ignoring the rest of the bits generated. The output will be the output generated by M_1 on the string $f_2(w)$.

We now show that M uses only logarithmic space. To see this, observe that the only space that we need to account for is the space that M_1 uses on the string $f_2(w)$, since we are generated $f_2(w)$ bit-by-bit. We know that on input $f_2(w)$, M_1 uses $O(\log|f_2(w)|)$ space. But, as in the previous paragraph, we know that $|f_2(w)| = q(|w|)$ for some polynomial q, and this means that

$$O(\log|f_2(w)|) = O(\log(q(|w|))) = O(\log|w|)$$

and hence the overall space used is logarithmic. So, it follows that the composition of two log-space computable functions is log-space computable, and this completes our proof. ■