

Lecture 1

*Lecturer: Peeter Laud**Scribe(s): Ilya Kuzovkin***Introduction**

During this lecture we looked at the proof of the halting theorem, which shows what is halting problem about. After that we prove two theorems (for deterministic and non-deterministic cases) which say that for every two problems f and g there is problem p , which takes more time than f , but less than g . Proved existence of a problem which is not NP-complete (in case if $P \neq NP$). Talked about NP-intermediate problems and looked into some philosophical notes about P complexity.

The Halting Problem

Theorem 1 *If we define language*

$$HALT = \{\langle \alpha, x \rangle \mid M_\alpha \text{ stops on input } x\}$$

then this language is not accepted by any Turing Machine.

Proof Assume there is TM M_{HALT} which accepts language HALT. Let there be another Turing Machine M' which takes x as input and invokes $M_{HALT}(\langle x, x \rangle)$ (Run machine encoded by x with input x).

We define M' behaviour as follows: if M_{HALT} accepts, then M' will work indefinitely, if M_{HALT} rejects then M' will return 1.

Now let us say β is encoding of machine M' and we run $M'(\beta)$. We have two cases:

$$\begin{aligned} M'(\beta) \text{ will stop} &\Leftrightarrow M_{HALT}(\langle \beta, \beta \rangle) \text{ rejects} \Leftrightarrow \\ &\Leftrightarrow M_\beta(\beta) \text{ will not stop} \equiv M'(\beta) \text{ will not stop} \end{aligned}$$

Contradiction!

And second case

$$\begin{aligned} M'(\beta) \text{ will not stop} &\Leftrightarrow M_{HALT}(\langle \beta, \beta \rangle) \text{ accepts} \Leftrightarrow \\ &\Leftrightarrow M_\beta(\beta) \text{ will stop} \equiv M'(\beta) \text{ will stop} \end{aligned}$$

Contradiction! ■

Deterministic Time Hierarchy Theorem

Theorem 2 Let f and g be two time-constructible functions, such that $f(n) > n$ and $\lambda n.f(n) \log f(n) \in o(g)$. Then $DTIME(f) \subsetneq DTIME(g)$

In other words there always is a function p which takes more time than f , but less than g .

Proof We introduce new function h such that it is more complex than f , but less complex than g :

- $h \in \omega(f)$
- $h(n) \log h(n) \in O(g)$

We define language D

$$D = \{\alpha \in \{0,1\}^* \mid \text{accepts } \alpha \text{ in } \leq h(|\alpha|) \text{ steps}\}^c$$

Note that c here means *complementary* – D consist of α which do not satisfy the condition in the brackets.

We pick language $L \in DTIME(f)$, and let machine M accept that language in time $c \cdot f$
We pick α such that $M_\alpha = M$ and $\frac{h(|\alpha|)}{f(|\alpha|)} > c$.

$$\begin{aligned} & \text{If } \alpha \in L \Rightarrow \\ & \Rightarrow M_\alpha(\alpha) \text{ accepts with } \leq c \cdot f(|\alpha|) \text{ steps } \Rightarrow \\ & \Rightarrow M_\alpha(\alpha) \text{ accepts with } \leq c \cdot h(|\alpha|) \text{ steps } \Rightarrow \\ & \Rightarrow \text{from the definition of } D \text{ we can see that } \alpha \notin D \Rightarrow \end{aligned}$$

$$L \neq D$$

At the same time $D \in DTIME(g)$ because, as it follows from the definition of D , we can accept or reject α in time $\leq DTIME(h) \leq DTIME(g)$ ■

Non-deterministic Time Hierarchy Theorem

Theorem 3 Let f and g be two time-constructible functions, such that $f(n) > n$ and $\lambda n.f(n+1) \in o(g)$. Then $NTIME(f) \subsetneq NTIME(g)$

This theorem states the same fact as the previous one, but for non-deterministic time. We cannot construct the proof in the same way as the previous one because there we had, that then machine rejects in accept on the complementary set of α . When dealing with non-deterministic machines we cannot say that – if one branch rejects, the α can be still accepted in some other branch.

Proof We introduce two new functions h and h' so that $f(n+1) \in o(h')$, $h' \in o(h)$, $h \in o(g)$, or we can say that in terms of time complexity $f \leq h' \leq h \leq g$.

Also we define a new function φ as follows:

$$\varphi(1) = 2$$

$$\varphi(i+1) = 2^{h(\varphi(i))}$$

Let $\tilde{\varphi}(n) = \max\{i \mid \varphi(i) \leq n\}$ (provide the argument i , with which φ value is closest to n from the left).

Function φ is used to split the set of lexicographically ordered bit strings into sets, such that each next set exponentially larger then previous.

Next we define language D :

$$D = \left\{ 1^n \left| \begin{array}{l} i := \tilde{\varphi}(n) - 1 \\ n \neq \varphi(i+1) \text{ and } M_i \text{ accepts } 1^{n+1} \text{ in time } h'(n) \\ \text{OR} \\ n = \varphi(i+1) \text{ and } M_i \text{ rejects } 1^{\varphi(i)+1} \text{ in time } g(\varphi(i)+1) \end{array} \right. \right.$$

This can be represented with following figure:

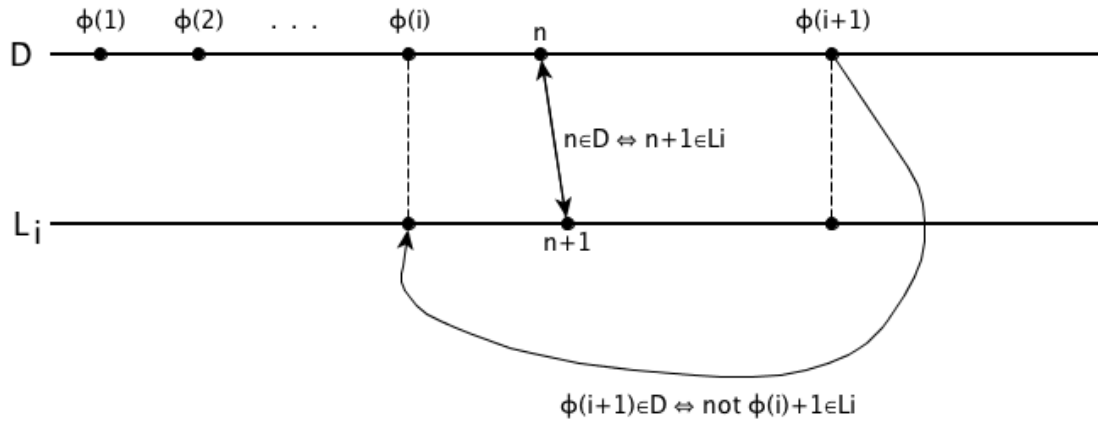


Figure 1: Representation of definition of D

We have machine M_i which accepts language L_i in h' steps.

The rule is that element $n \in D$ iff $n+1 \in L_i$, where i is argument of φ .

And one additional rule says that $\varphi(i+1) \in D$ iff $\varphi(i)+1 \notin L_i$

We want to show that $D \notin \text{NTIME}(f)$ and $D \in \text{NTIME}(g)$

$D \in \text{NTIME}(g)$

First we compute $\varphi(1), \varphi(2), \dots$ until n in order to find $\tilde{\varphi}(n)$. We can do it in such naive way since it will take only logarithmic time.

If $n \neq \varphi(i+1)$ when just simulate M_i and see if $n+1 \in L_i$. This is doable in h' time.

The situation is more complicated when $n \neq \varphi(i+1)$. In this case we have to search through all possible computational paths of M_i to make sure it rejects on every path (only then we can accept n to D). Here comes in play that every $\varphi(i+1)$ is exponentially larger than $\varphi(i)$. This gives us time to compute all $O(2^{g(\varphi(i)+1)})$ paths of M_i . Since both cases are doable in time less than g (by definition of D) we can say that $D \in \text{NTIME}(g)$.

$D \notin \text{NTIME}(f)$

Let language $L \in \text{NTIME}(f)$. L is accepted by machine M_i . Now we assume that $L = D$.

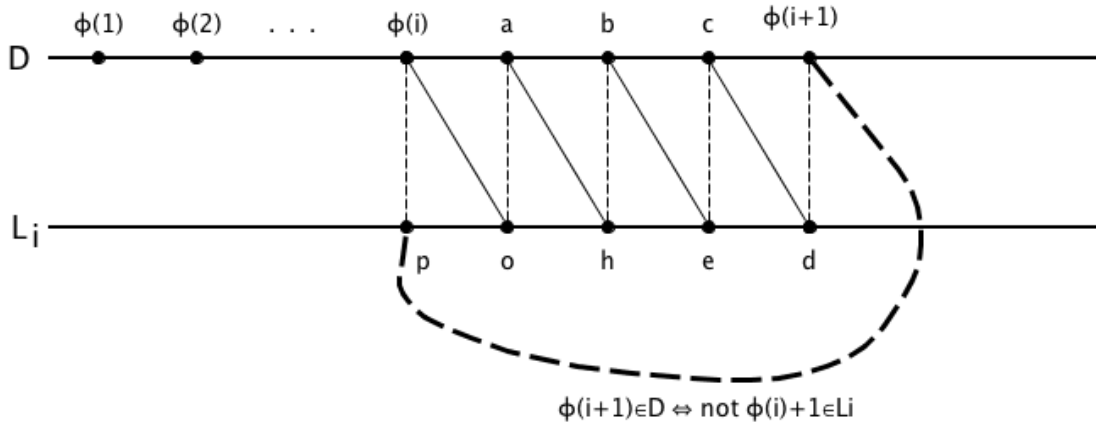


Figure 2: Illustration of contradiction

Now, when we assumed that $L = D$ we have three rules:

1. $n \in D$ iff $n + 1 \in L_i$ (by definition) – solid line on the Figure 2
2. $\varphi(i + 1) \in D$ iff $\varphi(i) + 1 \notin L_i$ (by definition) – bold dashed line on the Figure 2
3. $n \in D$ iff $n + 1 \in L_i$ (new rule comes from $L = D$) – dashed line on the Figure 2

Now if we apply these rules one by one we see:

If $p \in L_i \Rightarrow$ (rule 3) $\varphi(i) \in D \Rightarrow$ (rule 1) $o \in L_i \Rightarrow$ (rule 3) $a \in D \Rightarrow$ (rule 1) $\dots \Rightarrow$ (rule 3) $\varphi(i + 1) \in D \Rightarrow$ (rule 2) $p \notin L_i \Rightarrow$ Contradiction!

In the same way the second case

If $p \notin L_i \Rightarrow$ (rule 3) $\varphi(i) \notin D \Rightarrow$ (rule 1) $o \notin L_i \Rightarrow$ (rule 3) $a \notin D \Rightarrow$ (rule 1) $\dots \Rightarrow$ (rule 3) $\varphi(i + 1) \notin D \Rightarrow$ (rule 2) $p \in L_i \Rightarrow$ Contradiction!

We have showed $D \notin \text{NTIME}(f)$.

Now we know that $D \in \text{NTIME}(g)$ and $D \notin \text{NTIME}(f)$, which means $\text{NTIME}(f) \subsetneq \text{NTIME}(g)$. ■

Existence of not NP-complete problems

Theorem (Lander) 4 *If $P \neq NP$ then there exists a language $A \in NP \setminus P$ that is not NP-complete.*

Proof We will prove by constructing such language A. First we define a few things.

- M_1, M_2, \dots will be polynomial-time DTM such that language L_i is accepted by machine M_i
- f_1, f_2, \dots will be polynomial-time computable functions such that M_i computes f_i in time $O(n^i)$

We define A as

$$A = \{x \in \{0, 1\}^* \mid x \in \text{SAT and } g(|x|) \text{ is even}\}$$

Function $g(n)$ is defined as follows:

$$g(0) = 2$$

$$g(1) = 2$$

For $n \geq 2$ we do following recursive iterations:

1. take u to be largest such that $g(u)$ was computed
2. $k = g(u)$
3. $i = \lfloor \frac{k}{2} \rfloor$
4. for $j \in \{0, 1, 2, \dots\}$
 - (a) If k is even check $B_j \in L_i$ **XOR** $B_j \in A$
 - (b) If k is odd check $B_j \in \text{SAT}$ **XOR** $f_j(B_j) \in A$
 - (c) Now look at **XOR** result – if it is *true* return $k + 1$, if *false* return k

Note: since function itself does not have any boundaries and will work indefinitely, we will use a counter in the parallel process, which will stop the execution of the computation after n time units (for example seconds) have passed.

We will go through three claims which together show that A is in NP, but is not in P and is not in NP-complete.

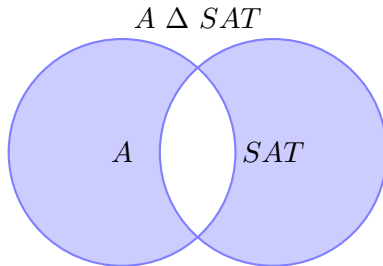
A \in NP

To accept or reject we have to check if x is in SAT. This is NP problem, so A belongs to NP by definition.

$A \notin P$

Assume $A \in P$. In that case there is language $L_i = A$. Let us consider smallest such i . In that case $g(n)$ will never be more than $2i$.

If during the iterations $g(n)$ will be most of the time $2i$ it will mean that **XOR** after some point will become always *false* \Rightarrow it is stuck in the "k is even" branch (step 4.a) \Rightarrow the result in "k is odd" will give us only finite set for SAT:



And because $A \in P$ and SAT is finite we can compute SAT in polynomial time \Rightarrow Contradiction!

On the other hand if $g(n)$ has not reached $2i$ and most of the time equals $2i + 1$ it will mean that iterations are stuck in the "k is odd" branch (step 4.b) \Rightarrow **XOR** gives *false* \Rightarrow There is $B \in SAT$ and $f(B) \in A \Rightarrow SAT \leq_m^p A \Rightarrow$ since $A \in P$ then due to reducibility SAT is also in P \Rightarrow Contradiction!

SAT is not reducible to A

If we show that SAT is not reducible to A we will show that A is not in NP-complete.

Assume SAT is reducible to A. Then there should be such f_i that $f_i(SAT) = A$.

We in the same way as in the previous section: $g(n)$ never grows past $2i + 1$.

If $g(n) = 2i + 1$ most of the time, then it is stuck in the branch 4.b and A will be finite $\Rightarrow A \in P \Rightarrow SAT \in P \Rightarrow$ Contradiction!

And another case if $g(n) = 2i$ most of the time \Rightarrow stuck in 4.a branch \Rightarrow there is $L_i = A \Rightarrow$ since $L_i \in P$ then also $A \in P \Rightarrow SAT \in P \Rightarrow$ Contradiction! ■

NP-intermediate problems

Problems which are in NP, but are not in P or NP-complete are called NP-intermediate. There is no proof of existence of such (otherwise it would mean that $P \neq NP$), but there are several problems which are considered to be a good candidates to be NP-intermediate:

- Finding whether two graphs are isomorphic
- Integer factorisation
- Discrete logarithm problem

Note about polynomial complexity

In practice it turned out to hold that if there is a P solution for the problem with a large polynomial, then there is also a solution with small polynomial. Theoretically there exist problems with polynomial algorithm with large c , but for practical problems $p^{\text{something}}$ seems to be reducible to p^{small} pretty often.