MTAT.07.004 — Complexity Theory October 13th-14th, 2011

Lecture 6

Lecturer: Peeter Laud *Scribe(s): Riivo Talviste*

1 Logarithmic memory

Turing machines working in logarithmic space become interesting when we are dealing with problems with really large input. For example, when the input is the connectivity graph of the internet.

It is important to remember that we only count the cells on the working tapes towards the space (memory) that a given Turing machine uses. The input tape and the (read-only) output tape are excluded.

2 Log-space reductions

Similarly to the polynomial-time reductions, we have log-space reductions.

Definition 1 Language L is *log-space* reducible to language L' (denote $L \leq_{\mathrm{m}}^{\mathrm{L}} L'$) if there exists a function $f : \{0,1\}^* \to \{0,1\}^*$, such that

- $f(x)$ is computable in space $O(\log |x|)$
- $x \in L$ iff $f(x) \in L'$.

$\diamondsuit$

From the previous lectures we know that polynomial-time reductions are transitive. If we have languages L, L' and L'', we can state that

$$L \leq_{\mathrm{m}}^{\mathrm{P}} L' \wedge L' \leq_{\mathrm{m}}^{\mathrm{P}} L'' \Rightarrow L \leq_{\mathrm{m}}^{\mathrm{P}} L''.$$

This is trivial, as if we have a reduction function f for $L \leq_{\mathrm{m}}^{\mathrm{P}} L'$ and a reduction function g for $L' \leq_{\mathrm{m}}^{\mathrm{P}} L''$, we can just use their composition $h = g \circ f$ as a reduction function for $L \leq_{\mathrm{m}}^{\mathrm{P}} \mathsf{L}''$. Since both f and g work in polynomial-time, h also works in polynomial-time.

Theorem 1 *$\leq_{\mathrm{m}}^{\mathrm{L}}$ is transitive. If $L \leq_{\mathrm{m}}^{\mathrm{L}} L'$ and $L' \in \mathsf{L}$ then $L \in \mathsf{L}$.*

Proof We have languages L, L' and L'' and want to state that

$$L \leq_{\mathrm{m}}^{\mathrm{L}} L' \wedge L' \leq_{\mathrm{m}}^{\mathrm{L}} L'' \Rightarrow L \leq_{\mathrm{m}}^{\mathrm{L}} L''.$$

Unfortunately, showing this is not as trivial as was in the case of polynomial-time. If we have a reduction function f for $L \leq_{\mathrm{m}}^{\mathrm{L}} L'$ and a reduction function g for $L' \leq_{\mathrm{m}}^{\mathrm{L}} L''$, we can again use their composition $h = g \circ f$ as a reduction function for $L \leq_{\mathrm{m}}^{\mathrm{P}} L''$, but not naively by first computing the output of f and using it as input for g, because the output of f might not fit into the logarithmic memory of h. Instead, we need a smarter way of computing h.

Recall from the previous lecture, that we have two ways to define Turing machines that compute a function:

- The machine has an extra *output tape*. It is write-only and the head can only move to the right.
- Languages L_i and L'_i must be in $\mathsf{DSPACE}(g)$ for all i, where
  - $L_i = \{x \,|\, |f(x)| \geq i\}$
  - $L'_i = \{x \,|\, x \in L_i \wedge i\text{-th bit of } f(x) \text{ is } 1\}$.
- The recognition of languages L_i and L'_i must be *uniform*. I.e., there must exist a TM M working in space g, such that $x \in L_i$ iff $M(i,x)$ accepts. Also, there must exist a TM M' working in space g, such that $x \in L'_i$ iff $M(i,x)$ accepts.

Since we can freely choose between the two definitions, let us assume that the Turing machine computing f uses the latter variant. That is, we have Turing machine:

- $M_f(i,x) = \mathsf{true} \iff |f(x)| \geq i$, and
- $M'_f(i,x) = \mathsf{true} \iff |f(x)| \geq i \wedge i\text{-th bit of } f(x) \text{ is } 1$.

The computation model for g and h does really not matter, so let us just say that these are computed by Turing machines M_g and M_h respectively.

Since $h = g \circ f$, we can use the input tape of M_h as input tape for M_f and M'_f (that is, for computing f) and the output tape of M_g as output tape of M_h. The only tricky part is involving the output of computing f and the input of M_g. Again, since h works in log-space, we cannot just precompute $f(x)$ and feed this as input for M_g. Instead, we calculate $f(x)$ bit-by-bit. So M_h just runs M_g and whenever M_g needs to read the i-th bit from its input tape, we just run $M'_f(i,x)$ and return its value (converted from boolean to bit).

Remark We can also choose to model the Turing machine computing f using the first definition (ie. with write-only output tape), denoted by $\bar{M}_f$. Then, if M_g wants to read the first input bit, it just runs $\bar{M}_f$ until it outputs the necessary bit and then puts the computation of f on hold (pauses $\bar{M}_f$) and carries on its own computation of g. For the next input bit M_g can just resume $\bar{M}_f$ until it outputs the next bit and pause it again. However, whenever M_g wants to move left on its input tape, it has to restart $\bar{M}_f$ from the beginning and memorize how many bits it has to let it compute before taking over. The latter requires an extra "counter" tape in M_g, but it will only take logarithmic amount of memory.

■

3 NL-completeness

Definition 2 A language L is NL-*hard* if for any $L' \in \mathsf{NL}$ we have $L' \leq^{\mathrm{L}}_{\mathrm{m}} L$. Language L is NL-complete if it is NL-hard and belongs to NL. ♢

Theorem 2 PATH *is* NL-*complete.*

Proof From the previous lecture we already know that $\mathsf{PATH} \in \mathsf{NL}$. So we only have to show that PATH is NL-hard, that is, for any $L \in \mathsf{NL}$ we have $L \leq_{\mathrm{m}}^{\mathrm{L}} \mathsf{PATH}$.

Let M be a non-deterministic Turing machine that accepts $L \in \mathsf{NL}$ in $O(\log n)$ space. Let us define function f for the reduction $L \leq_{\mathrm{m}}^{\mathrm{L}} \mathsf{PATH}$:

$$f : \{0,1\}^* \to (G, C_s, C_t) : x \in L \iff f(x) \in \mathsf{PATH},$$

where $G \in \{0,1\}^*$ is a graph representation of all the configurations of M, given by its adjacency matrix and C_s is the initial configuration of M and C_t is a configuration with an accepting state.

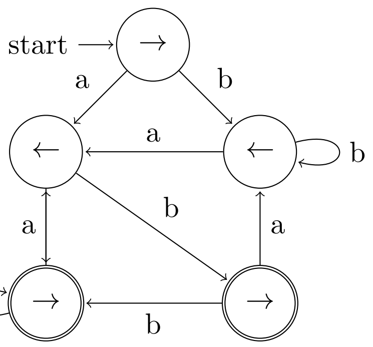
The function f is able to compute the adjacency matrix of M, because for every two configurations C and C', f can check in $O(\log n)$ space whether C' is one of the at most two (because we are using binary) configurations that follow C. Given G and two configurations C_s, C_t of size $O(\log n)$, we can now non-deterministically decide in space $O(\log n)$ whether there exists a path $C_s \to C_t$. ■

4 $\mathsf{DSPACE}(0)$ and finite automata

The idea of $\mathsf{DSPACE}(0)$ is simple. If we are allowed to use only a constant number of memory cells (space) then we can easily just hold the same information in the state of the Turing machine and get rid of the working tapes altogether.

This way $\mathsf{DSPACE}(0)$ is equal to *two-way finite automata* — finite automata where in each node there is an arrow showing in which direction the head reading the input moves. In this case, if the input reading head goes over the edge on the right side (so it has read all the input) we accept the input if the automata is currently in an accepting state and reject it otherwise. However, if the reading head goes over the edge on the left side or the machine goes into an infinite loop, we will reject the input.

For example, consider the following two-way finite automata, which on the input of `aaabba` goes into cycle.



Claim 3 *Two-way finite automata have the same power as (one-way) finite automata.*

Sketch of Proof (Informal) A Turing machine corresponding to a one-way finite automata is a TM where the head on the input tape is only allowed to move to the right. We show that a one-way finite automata can be used to simulate a two-way finite automata.

Let us have a two-way finite automaton M with states $Q = \{q_0, q_1, \ldots, q_n\}$. We will construct a "regular" finite automaton M', where the head on input tape is only able to move right. As long as M moves only to the right on its input tape, the automaton M' acts identically. However, suppose that M is in some state q_i reading the k-th position of the input string and wants to move its input reading head to the left. There are two possibilities:

- either there exists some state q_j where the reading head is back in the position k;
- or the head on the input tape never comes back.

In the former case, M' can just jump right from q_i to q_j and skip all the states inbetween, in the latter case however, M' goes to a special rejecting state $\perp$. So for simulating M, the machine M' must store, together with the current state of M, also a mapping $f : Q \to Q \cup \{\perp\}$ that describes the behaviour of M on left moves. ■

Theorem 4 *A one-way finite automaton may need exponentially more states than a two-way finite automaton with the same functionality.*

To illustrate this, let us assume that for some n we have a language

$$L_n = \{ww | w \in \{0,1\}^n\},$$

that is a language of $2n$-bit inputs, where two identical n-bit words are concatenated together. A two-way automaton could read the 1-st bit, then move to the $n+1$-st bit, compare it to the 1-st, move back and read the 2-nd bit, move to the $n+2$-nd bit, compare it to the 2-nd bit and so on. This TM works in $O(n^2)$ time and must store at most one bit of information about the input string at any time. This movement and storage may be encoded in $O(n^2)$ states. However, a one-way automaton that can only move its head to the right on the input tape, has to remember all n bits by the time it reaches $n+1$-th bit on its input tape. Hence, this automaton would need 2^n states.

4.1 Least amount of usable memory

Theorem 5 *If $s \in o(\lambda n. \log\log n)$ then $\mathsf{DSPACE}(s) = \mathsf{DSPACE}(0)$.*

In other words, there are no other space complexity classes between the constant space $\mathsf{DSPACE}(0)$ and $\mathsf{DSPACE}(\lambda n. \log\log n)$. For example, the class $\mathsf{DSPACE}(\lambda n. \log\log\log n)$ does not exist.

Sketch of Proof (Informal) Let M be a Turing Machine. Let it be a two-tape machine (has only a single work tape) and let it always move the read-write head of its work tape to the leftmost position before finishing the work.

Let $f(n)$ be the maximum amount of memory that a Turing machine M uses given an input $x \in \{0,1\}^{\leq n}$. Let $f^{-1}(m) = \min\{n | f(n) \geq m\}$.

Fix m. Let the input $x \in \{0,1\}^{f^{-1}(m)}$ be such that $M(x)$ uses m space (memory cells).

Let the *memory configuration* of M consist of

- the data on its working tape;

- the position of the head on its working tape and
- the state of M itself.

Let C_k is the set of all possible memory configurations where the head on the working tape is in position k.

Consider the following input:

$$\triangleright x_1 x_2 \dots x_r \dots x_s \dots x_{f^{-1}(m)} \square\square \dots \tag{1}$$

In positions r and s the sets of all possible memory configurations are C_r and C_s respectively. Now, for some x_r, x_s, where $x_r = x_s \wedge C_r = C_s$, we could leave out the input symbols between x_{r+1} and x_s and get the following input:

$$\triangleright x_1 x_2 \dots x_r x_{s+1} x_{s_2} \dots x_{f^{-1}(m)} \square\square \dots \tag{2}$$

It is easy to see that on both of these inputs M finishes with the same configuration and with the head of the input tape in the same position. We can say that M does not notice the input between x_{r+1} and x_s. Hence M uses the same amount of memory on (2) as it does on (1). This contradicts the choice of x as the input string of least length where M uses at least m memory cells. Hence $x_r = x_s$ implies $C_r \neq C_s$.

The machine M has a total of $|\Gamma|^m \cdot m \cdot |Q|$ different memory configurations, where Γ is its alphabet and Q is the set of M's states. The size of the set of all possible memory configurations of M is given by $2^{|\Gamma|^m \cdot m \cdot |Q|}$. When we take into account that we have four different symbols $(0, 1, \triangleright$ and $\square)$, we can say that

$$\begin{aligned} f^{-1}(m) &\leq 4 \cdot 2^{|\Gamma|^m \cdot m \cdot |Q|} \\ m &\leq f(\underbrace{4 \cdot 2^{|\Gamma|^m \cdot m \cdot |Q|}}_{n}) \end{aligned}$$

from what we can see that

$$\begin{aligned} \frac{n}{4} &\leq 2^{|\Gamma|^m \cdot m \cdot |Q|} \\ \log \frac{n}{4} &\leq |\Gamma|^m \cdot m \cdot |Q| \\ \log\log \frac{n}{4} &\leq m \cdot \log|\Gamma| \cdot \log m \cdot \log|Q| = C \cdot m \end{aligned}$$

So we have that

$$\frac{1}{C} \log\log n \leq m \leq f(n),$$

where we can see that m is lower bounded by $O(\log\log n)$. ■

Exercise. Given

$$L = \{\#\mathsf{bit}(1)\#\mathsf{bit}(2)\#\cdots\mathsf{bit}(n)\#\,|\,n \in \mathbb{N}\},$$

show that $L \in \mathsf{DSPACE}(\lambda n.\log\log n)\backslash\mathsf{DSPACE}(0)$.

Notice that $x \in \{0, 1, \#\}^*$ and $|x| = O(n \log n)$. First, we want to show that $L \in \mathsf{DSPACE}(\lambda n.\log\log n)$. The idea is simple, for every pair of two subsequent numbers $\#bit(x)\#bit(y)\#$ we check whether $y - x = 1$. We can do that in $O(\log\log n)$ space, as for every such pair, we can just memorize the current bit and the position number we are comparing.

Secondly, we have to show that $L \notin \mathsf{DSPACE}(0)$. We showed that $\mathsf{DSPACE}(0)$ is equal to finite automata. From the formal languages course, we should know that $\mathsf{DSPACE}(0)$ is thus the class of regular languages, where the following lemma holds.

Lemma 6 (Pumping lemma)

$$L \in \mathsf{DSPACE}(0) \iff \exists n \forall x \in L, |x| \geq n,$$

such that the following holds:

$$\exists u, v, w : x = uvw \wedge 1 \leq |v| \leq n \wedge \forall_{0 \leq i \leq \infty} i : uv^i w \in L.$$

In this lemma, n is larger than the number of nodes (states) in the corresponding finite automaton. So v is the part of the input that is between being in a given state and returning to the same after some steps, i.e. v is the cyclic part of the input. Clearly, the language L in the exercise does not contain any cyclic parts, so $L \notin \mathsf{DSPACE}(0)$.

5 Complexity classes of complementary languages

Let $\mathcal{C}$ be a complexity class. The class $\mathsf{co}\mathcal{C}$ is

$$\mathsf{co}\mathcal{C} = \{L^{\mathrm{c}}\,|\,L \in \mathcal{C}\}\ .$$

That is, $\mathsf{co}\mathcal{C}$ is the class of all languages whose complement L^{c} is in $\mathcal{C}$.

For deterministic Turing machines, $\mathcal{C} = \mathsf{co}\mathcal{C}$, as we can just switch the machine's output ("accept" or "reject"). So for any f we have

$$\mathsf{DTIME}(f) = \mathsf{coDTIME}(f) \text{ and } \mathsf{DSPACE}(f) = \mathsf{coDSPACE}(f).$$

For example,

$$\mathsf{coP} = \mathsf{P}. \quad \mathsf{coPSPACE} = \mathsf{PSPACE}. \quad \mathsf{coL} = \mathsf{L}.$$

However, NP and coNP are thought to be different. For example, we know that $\mathsf{SAT} \in \mathsf{NP}$, so $\mathsf{coSAT} \in \mathsf{coNP}$, where coSAT is the set of *unsatisfiable* boolean formulas.

$\mathsf{P} = \mathsf{NP}$ would imply $\mathsf{NP} = \mathsf{coNP}$. Opposite implication is not known.

5.1 Functions computable by NTMs

Thus far we have only talked about computing functions in the case of using deterministic Turing machines. For non-deterministic TMs, computing functions is quite similar.

Definition 3 An NTM M computes the function $f : \{0,1\}^* \rightarrow \{0,1\}^*$, if on input x

- each computation path of M ends by
  - M outputting $f(x)$, or
  - M giving up (outputting "*don't know*")
- at least one computation path of M ends by outputting $f(x)$.

$\diamondsuit$

We only have to stress that if more than one computation path is able to find the function value, they all must output the same value, i.e. the function value $f(x)$ must be unique.

5.2 Number of reachable vertices

For the following part, we need the next exercise and theorem.

Exercise. Given graph G with n vertices and a vertex s. How many vertices can be reached from s?

Theorem 7 (Immerman-Szelepscényi) *This can be computed by a NTM in space* $O(\log n)$.

Proof Let $S(k)$ be the set of vertices reachable from s in at most k steps. The following algorithm can be used to compute $S(n-1)$. To get a better overview, we give the full algorithm in many iterations, where every new iteration is more detailed than the previous one (with the changed portion of the algotithm highlighted).

```
$|S(0)| := 1$
for $k := 1$ to $n-1$ do
    compute $|S(k)|$ from $|S(k-1)|$
```

```
$|S(0)| := 1$
for $k := 1$ to $n-1$ do
    $|S(k)| := 0$
    for $u \in V(G)$ do
        $b := (u \overset{?}{\in} S(k))$
        if $b$ then $|S(k)| := |S(k)| + 1$
```

```
$|S(0)| := 1$
for $k := 1$ to $n-1$ do
    $|S(k)| := 0$
    for $u \in V(G)$ do
        $m := 0$    $b :=$ false
        for $v \in V(G)$ do
            $b' := (v \overset{?}{\in} S(k-1))$        -- nondeterministic procedure
            if $b'$ then
                $m := m + 1$
                if $(v, u) \in E(G)$ then $b :=$ true
        if $m < |S(k-1)|$ then give up
        if $b$ then $|S(k)| := |S(k)| + 1$
```

Here the variable m is used to make sure, that we only count the path where we have made the "right" non-deterministic decisions, meaning that at the end, m should equal $|S(k-1)|$.

```
$|S(0)| := 1$
for $k := 1$ to $n-1$ do
    $|S(k)| := 0$
    for $u \in V(G)$ do
        $m := 0$    $b :=$ false
        for $v \in V(G)$ do
            $w_0 := s$    $b' :=$ true
            for $p := 1$ to $k-1$ do
                choose $w_p \in V(G)$
                $b' := b' \wedge (w_{p-1}, w_p) \in E(G)$
            if $b' \wedge (w_{k-1} = v)$ then
                $m := m + 1$
                if $(v, u) \in E(G)$ then $b :=$ true
        if $m < |S(k-1)|$ then give up
        if $b$ then $|S(k)| := |S(k)| + 1$
```

The highlighted part now constructs $S(k-1)$, but it does not hold it in memory, it just finds $|S(k-1)|$. It is easy to see that the whole algorithm uses only a constant number of variables, each taking logarithmic amount of memory. Hence, the algorithm works in $O(\log n)$ space. ■

5.3 NSPACE(f) and coNSPACE(f)

Previously, we showed that for deterministic Turing machines, language complexity classes and the complexity classes of corresponding complementary languages are equal. Now we show that $\mathsf{NSPACE}(f) = \mathsf{coNSPACE}(f)$ also holds.

Theorem 8 *If f is a space-computable function, and $f \in \Omega(\lambda n. \log n)$, then* $\mathsf{NSPACE}(f) = \mathsf{coNSPACE}(f)$.

Sketch of Proof (Informal) Let NTM M working in space f decide $L \subseteq \{0,1\}^*$. We need to construct a machine M' that decides L^c.

The computation graph of a NTM M working in space f has at most $c^{f(n)}$ vertices for some c. M' uses previous algorithm to compute $|S(c^{f(n)})|$. According to the Immerman-Szelepscényi theorem, it takes $\log c^{f(n)} = \log c \cdot f(n) = c' \cdot f(n)$ space. Then M' checks all configurations of M for being included in $S(c^{f(n)})$:

- If accepting configuration found, then M' rejects.
- If algorithm gives up, then M' rejects.
- If no accepting configuration found, M' accepts.

■

Notice, that by this theorem, $\mathsf{NL} = \mathsf{coNL}$ also holds.

6 Oracle Turing Machines

An *Oracle TM* (either det. or non-det.) M is a TM with

- A designated tape — the *query tape*
- Three designated states q_{query}, q_{yes}, q_{no}.

An *oracle* $\mathcal{O}$ defines a language, so $\mathcal{O} \subseteq \{0,1\}^*$.
Whenever M running together with $\mathcal{O}$ (denoted $M^{\mathcal{O}}$) goes into state q_{query},

- the contents of query tape is interpreted as a bit-string x;
- M goes to state q_{yes} if $x \in \mathcal{O}$. Otherwise M goes to state q_{no}.
- This takes a single step.

So Turing machines with oracle access are more powerful, as they can decide in a single step whether $x \in \mathcal{O}$ for any x. An oracle $\mathcal{O}$ gives us *relativized* complexity classes $\mathsf{P}^{\mathcal{O}}$, $\mathsf{NP}^{\mathcal{O}}$, etc.

6.1 Limits of diagonalization

Recall, that the diagonalization proofs used the facts that (a) there is an efficient mapping between bit-strings and TMs; and (b) efficient universal TMs exist. However, the proofs did not really consider the internal workings of the TMs M_i from the enumeration of all TMs. All these proofs would go through also for oracle TMs.

Can a similar proof decide $\mathsf{P} \overset{?}{=} \mathsf{NP}$. As the following theorem states, the answer is *no*.

Theorem 9 *There exist $A, B \subseteq \{0,1\}^*$, such that $\mathsf{P}^A = \mathsf{NP}^A$ and $\mathsf{P}^B \neq \mathsf{NP}^B$.*

Proof First, let us show that $\exists A : \mathsf{P}^A = \mathsf{NP}^A$. For A, consider this language:

$$\mathsf{EXPCOM} = \{\langle M, x, 1^n\rangle \,|\, \text{DTM } M \text{ accepts } x \text{ in } \leq 2^n \text{ steps}\} \ .$$

This is a complete language for *exponential-time* computation.

A computation in $\mathsf{NP}^{\mathsf{EXPCOM}}$ on input of length n would

- *non-deterministically* choose a certificate of length $\leq p(n)$ (where $p()$ stand for polynomial);
- (make up to $p(n)$ steps), solve up to $p(n)$ problems, each requiring up to $2^{p(n)}$ steps.

A deterministic algorithm doing the same things would need at most

$$2^{p(n)} \cdot p(n) \cdot 2^{p(n)} = 2^{2 \cdot p(n) + \log p(n)} \text{ steps.}$$

This still fits in EXPCOM. Thus $P^{\mathsf{EXPCOM}} = \mathsf{NP}^{\mathsf{EXPCOM}}$. A different example for the language A would be any PSPACE-complete language.

Secondly, let us show that $\exists B : \mathsf{P}^B \neq \mathsf{NP}^B$. For any $B \subseteq \{0,1\}^*$ let

$$U_B = \{1^n \,|\, \exists x : |x| = n \wedge x \in B\} \ .$$

So U_B is a language consisting of all the different lenghts (represented as unary strings) of all the elements in B.

For any B we have $U_B \in \mathsf{NP}^B$, because a TM M^B deciding U_B could just non-deterministically choose an input with the right length and use the oracle B to decide if this input belongs to B:

```
$M^B(1^n)$ :
    **choose** $x \in \{0,1\}^n$
    $B(x)$
```

Now, we will construct a language B, such that $U_B \notin \mathsf{P}^B$.

Let $M_1, M_2, \ldots$ be the enumeration of oracle DTM-s and let t be a superpolynomial function, such that $\forall n : t(n) < 2^n$. We will construct B in stages, where stage i ensures, that M_i^B does not decide U_B in $t(n)$ time. We define

$$B = \{x \in \{0,1\}^* \,|\, \exists i : \varphi_i(x) = \mathsf{yes}\},$$

where $\varphi_i (i = 0, 1, 2, \ldots)$ is a partial function from $\{0,1\}^*$ to $\{\mathsf{yes}, \mathsf{no}\}$, such that

- φ_0 is always undefined.
- Each φ_i is defined only on a finite subset of $\{0,1\}^*$.
- If $\varphi_i(x)$ is defined, then $\varphi_{i+1}(x) = \varphi_i(x)$.
- For each $x \in \{0,1\}^*$ there exists i, such that $\varphi_i(x)$ is defined.

Constructing φ_{i+1} With φ_i, we have declared for a finite number of strings whether they are in B or not. Now choose n large enough so that it exceeds the length of any such string, for example let

$$n = (\max_{\varphi_i(x) \text{ is defined}} |x|) + 1 \ .$$

Now run $M_{i+1}^{(\cdot)}(1^n)$ for $t(n)$ steps. If M_{i+1} queries for x, then

- If $\varphi_i(x)$ is defined, then answer $\varphi_i(x)$.
- If $\varphi_i(x)$ is not defined, then answer no.
  - Set $\varphi_{i+1} = \varphi_{i+1}[x \mapsto \mathsf{no}]$.

In other words, for all queried strings whose status has already been determined, we answer honestly. However, for all other strings we declare that they are not in B. Note, that until this point, we have not declared that B has any string with length n.

Now, if $M_{i+1}^{(\cdot)}$ stops in $t(n)$ steps, then

- If $M_{i+1}^{(\cdot)}$ accepts, we say that all strings with length n are not in B, ensuring that $1^n \notin U_B$.
- If $M_{i+1}^{(\cdot)}$ rejects then pick $x \in \{0,1\}^n$ that $M_{i+1}^{(\cdot)}$ did not query (such x exists, because $M_{i+1}^{(\cdot)}$ did not have time to query all 2^n possibilities) and declare it to be in B, thus ensuring that $1^n \in U_B$.

In either case, the answer of $M_{i+1}^{(\cdot)}$ is incorrect. Thus $U_B \notin \mathsf{P}^B$. ■