

Definition 9.1

- A function $f : \mathcal{N} \rightarrow \mathcal{N}$, where $f(n)$ is at least $O(\log n)$, is called **space constructible** if the function that maps the string 1^n to the binary representation of $f(n)$ is computable in space $O(f(n))$.
- For non-integer function $f(n)$, we mean $\lfloor f(n) \rfloor$.
- If $f(n) = o(n)$, we assume that we have a read-only input tape.
- All commonly occurring functions that are at least $O(\log n)$ are space constructible, including $\log_2 n$, n^2 , and 2^n .

22c:131 Limits of Computation

Hantao Zhang

<http://www.cs.uiowa.edu/~hzhang/c131>

The University of Iowa
Department of Computer Science

Ch.9 INTRACTABILITY

Construction of D

TM D runs in space $O(f(n))$ but is not equivalent to any TM M which uses $o(f(n))$ space.

$D =$ "On input w :

1. Let n be the length of w .
2. Compute $f(n)$ using space constructibility, and mark off this much space. If later stages ever attempt to use more, *reject*.
3. If w is not of form $\langle M \rangle 10^*$ for some TM M , *accept*.
4. Simulate M on w while counting the number of steps used in the simulation. If the count ever exceeds $2^{f(n)}$, *accept*.
5. If M accepts, *reject*. If M rejects, *accept*."

Theorem 9.3

- **Space hierarchy theorem** For any space constructible function $f : \mathcal{N} \rightarrow \mathcal{N}$, a language A exists that is decidable in space $O(f(n))$ but not in space $o(f(n))$.
- **Proof idea:** We need to find a TM D that runs in space $O(f(n))$ but is not equivalent to any TM M which uses $o(f(n))$ space.
- For any TM M which uses $o(f(n))$ space, we construct D such that D and M behave differently on the input w , where $w = \langle M \rangle 01^k$.
- If M loops, D must accept.

Corollary 9.7

- $PSPACE \subset EXPSPACE = \bigcup_k SPACE(2^{n^k})$.
- Problems in $EXPSPACE - PSPACE$ are intractable because they require exponential space.

Corollary 9.4

- For any two functions $f_1, f_2 : \mathcal{N} \rightarrow \mathcal{N}$, $f_1(n) = o(f_2(n))$ and f_2 is space constructible, $SPACE(f_1(n)) \subset SPACE(f_2(n))$.
- **Corollary 9.5** For $c_1 < c_2$, $n^{c_1} = o(n^{c_2})$, and n^{c_2} is space constructible, so $SPACE(n^{c_1}) \subset SPACE(n^{c_2})$.

Theorem 9.10

- **Time hierarchy theorem** For any time constructible function $t : \mathcal{N} \rightarrow \mathcal{N}$, a language A exists that is decidable in $O(t(n))$ but not in $o(t(n)/\log t(n))$ time.
- **Proof idea:** We need to find a TM D that runs in $O(t(n))$ but is not equivalent to any TM M which uses $o(t(n)/\log t(n))$ time.
- For any TM M which uses $o(t(n)/\log t(n))$ time, we construct D such that D and M behave differently on the input w , where $w = \langle M \rangle 01^k$.
- If M loops, D must accept.

Definition 9.8

- A function $t : \mathcal{N} \rightarrow \mathcal{N}$, where $t(n)$ is at least $O(n \log n)$, is called **time constructible** if the function that maps the string 1^n to the binary representation of $t(n)$ is computable in time $O(t(n))$.
- All commonly occurring functions that are at least $O(n \log n)$ are time constructible, including $n \log_2 n$, n^2 , and 2^n .

Corollary 9.11

- For any two functions $t_1, t_2 : \mathcal{N} \rightarrow \mathcal{N}$, $t_1(n) = o(t_2(n)/\log t_2(n))$ and t_2 is time constructible, $\text{TIME}(t_1(n)) \subset \text{TIME}(t_2(n))$.
- **Corollary 9.12** For $c_1 < c_2$, $n^{c_1} = o(n^{c_2}/\log t_2(n))$, and n^{c_2} is time constructible, so $\text{TIME}(n^{c_1}) \subset \text{TIME}(n^{c_2})$.
- **Corollary 9.13** $P \subset \text{EXPTIME}$.

Construction of D

TM D runs in $O(t(n))$ but is not equivalent to any TM M which uses $o(t(n)/\log t(n))$ time.

$D =$ "On input w :

1. Let n be the length of w .
2. Compute $t(n)$ using time constructibility, and store the value $\lceil t(n)/\log t(n) \rceil$ on the tape.
3. If w is not of form $\langle M \rangle 10^*$ for some TM M , *accept*.
4. Simulate M on w while counting the number of steps used in the simulation. If the count ever exceeds $\lceil t(n)/\log t(n) \rceil$, *accept*.
5. If M accepts, *reject*. If M rejects, *accept*."

Exponentiation Operation

- Let $\uparrow: \text{REX} \times \mathcal{N} \rightarrow \text{REX}$ be the exponentiation operation, $R \uparrow k = R^k$.
- Generalized regular expressions contains exponentiation operation.
- $\text{EQ}_{\text{REX}\uparrow} = \{ \langle Q, R \rangle \mid Q \text{ and } R \text{ are equivalent generalized regular expressions} \}$.

Regular Expressions

- Three base cases:
 - \emptyset is a regular expression denoting the language \emptyset ;
 - ϵ is a regular expression denoting the language $\{\epsilon\}$;
 - For any $a \in \Sigma$, a is a regular expression denoting the language $\{a\}$;
- Three recursive cases: If r_1 and r_2 are regular expressions denoting languages L_1 and L_2 , respectively, then
 - **Union:** $r_1 \cup r_2$ denotes $L_1 \cup L_2$;
 - **Concatenation:** $r_1 r_2$ denotes $L_1 L_2$;
 - **Star:** r_1^* denotes L_1^* .

Two NFAs are inequivalent?

NTM N can answer this question in linear space.

$N =$ "On input $\langle N_1, N_2 \rangle$, where N_1 and N_2 are NFAs: expressions"

1. Place a marker on each of the start states of N_1 and N_2 .
2. Repeat Step 3 $2^{q_1+q_2}$ times, where q_1 and q_2 are the numbers of states in N_1 and N_2 .
3. Nondeterministically select an input symbol and change the positions of the markers on the states of N_1 and N_2 to simulate reading a symbol.
4. If at any point in Step 3, a marker was placed on an accept state of one of NFAs and not on any accept state of the other NFA, *accept*. Otherwise, *reject*."

EXPSpace-Completeness

Definition 9.14 A language B is *EXPSpace*-complete if it satisfies two conditions:

1. $B \in \text{EXPSpace}$
2. Every $A \in \text{EXPSpace}$ is polynomially time reducible to B .

Theorem 9.15 $EQ_{\text{REX}\uparrow}$ is *EXPSpace*-complete.

Recall $EQ_{\text{REX}\uparrow} = \{ \langle Q, R \rangle \mid Q \text{ and } R \text{ are equivalent generalized regular expressions} \}$.

$EQ_{\text{REX}\uparrow}$ is in EXPSpace

$E =$ "On input $\langle R_1, R_2 \rangle$, where R_1 and R_2 are generalized regular expressions"

1. Convert R_1 and R_2 to equivalent regular expressions B_1 and B_2 that use repetition instead of exponentiation.
2. Convert B_1 and B_2 to equivalent NFAs N_1 and N_2 , using the conversion procedure given in the proof of Lemma 1.55
3. Use the deterministic version of algorithm N to determine whether N_1 and N_2 are equivalent."