



Time Complexity



Agenda for this week

- Mon Apr 25 (today)
 - Time Complexity, Part I
- Wed Apr 27
 - Time Complexity, Part II
- Fri Apr 29
 - Wrap-up and Review for final



Warm-up question (4/25/22)

What is your favorite ice-cream topping?

Send me your response by Canvas email.



Overview

- So far in the course we saw two parts
 - Part 1: Automata and Languages
 - Regular Languages
 - Context-Free Languages
 - Part 2: Computability Theory
 - The Church-Turing Thesis
 - Decidability
 - Reducibility
- In this lecture and the next we get just a quick intro to Complexity Theory (Part 3 in the book)
 - Part 1 and Part 2 is what this course is mainly about



Complexity Theory

- Investigation of resources required for solving computational problems:
 - Time
 - Memory
 - Or other resources
- We get an intro to just Time Complexity
 - How time is measured
 - Classify problems according to amount of time needed to solve



Measuring complexity

Example. Consider the language

$$A = \{0^k 1^k \mid k \geq 0\}$$

How much time does a single-tape TM need to decide A ?

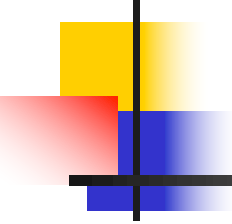
$M_1 =$ “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
 3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.”



Time complexity

Let M be a deterministic Turing machine that halts on all inputs. The *running time* or *time complexity* of M is the function $f: \mathcal{N} \rightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n . If $f(n)$ is the running time of M , we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.



Asymptotic Analysis: Big-O notation

Let f and g be functions $f, g: \mathcal{N} \rightarrow \mathcal{R}^+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,

$$f(n) \leq c g(n).$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an *upper bound* for $f(n)$, or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that we are suppressing constant factors.



Example

Let $f_1(n)$ be the function $5n^3 + 2n^2 + 22n + 6$. Then, selecting the highest order term $5n^3$ and disregarding its coefficient 5 gives $f_1(n) = O(n^3)$.

Let's verify that this result satisfies the formal definition. We do so by letting c be 6 and n_0 be 10. Then, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for every $n \geq 10$.

In addition, $f_1(n) = O(n^4)$ because n^4 is larger than n^3 and so is still an asymptotic upper bound on f_1 .

However, $f_1(n)$ is not $O(n^2)$. Regardless of the values we assign to c and n_0 , the definition remains unsatisfied in this case. ■



Small-o notation

Let f and g be functions $f, g: \mathcal{N} \rightarrow \mathcal{R}^+$. Say that $f(n) = o(g(n))$ if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < c g(n)$ for all $n \geq n_0$.



Examples

1. $\sqrt{n} = o(n)$.
2. $n = o(n \log \log n)$.
3. $n \log \log n = o(n \log n)$.
4. $n \log n = o(n^2)$.
5. $n^2 = o(n^3)$.



Analyzing Algorithms: revisiting M_1

M_1 = “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.”

Stage 1: $O(n)$

Stage 2 and 3: $(n/2)O(n) = O(n^2)$

Stage 4: $O(n)$

Total time: $O(n) + O(n^2) + O(n) = O(n^2)$



Classifying languages according to time requirements

Let $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the *time complexity class*, $\text{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

Recall the language $A = \{0^k 1^k \mid k \geq 0\}$. The preceding analysis shows that $A \in \text{TIME}(n^2)$ because M_1 decides A in time $O(n^2)$ and $\text{TIME}(n^2)$ contains all languages that can be decided in $O(n^2)$ time.

Is there a machine that decides A asymptotically faster?



Faster Algorithm for A

$M_2 =$ “On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*.”

Stages 1 and 5: $O(n)$

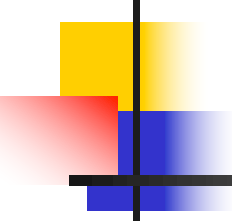
Stage 2, 3 and 4: $(1+\log n) O(n) = O(n \log n)$

Total: $O(n) + O(n \log n) = O(n \log n)$



Where then does A belong?

- M_1 implied that A is in $\text{TIME}(n^2)$
- M_2 implied that A is in $\text{TIME}(n \log n)$
- The second result cannot be further improved on a single-tape machine
- We can decide the language A in $O(n)$ time if the TM has a second-tape
- Hence the time complexity of A on a single-tape TM is $O(n \log n)$, and on a two-tape machine is $O(n)$.
- Note that the complexity of A depends on the model of computation selected.



Complexity theory vs computability theory

- The discussion in the previous slide highlights an important difference between complexity theory and computability theory
- In **computability theory**
 - The Church-Turing thesis implies that all reasonable models of computation are equivalent – in that they decide the same class of languages
- In **complexity theory**
 - The choice of model affects the time complexity of languages
- In **complexity theory**, we classify computational problems according to their time complexity. But with which model do we measure time?
- Fortunately, time requirements don't differ greatly for typical deterministic models.
- So, if we “build our tents large enough”, we can classify problems neatly in a few classes



Complexity relationships among models

- We consider three models to examine how the choice of computational model can affect the time complexity of languages:
 - Single-tape Turing Machine
 - Multiple-tape Turing Machine
 - Nondeterministic Turing Machine



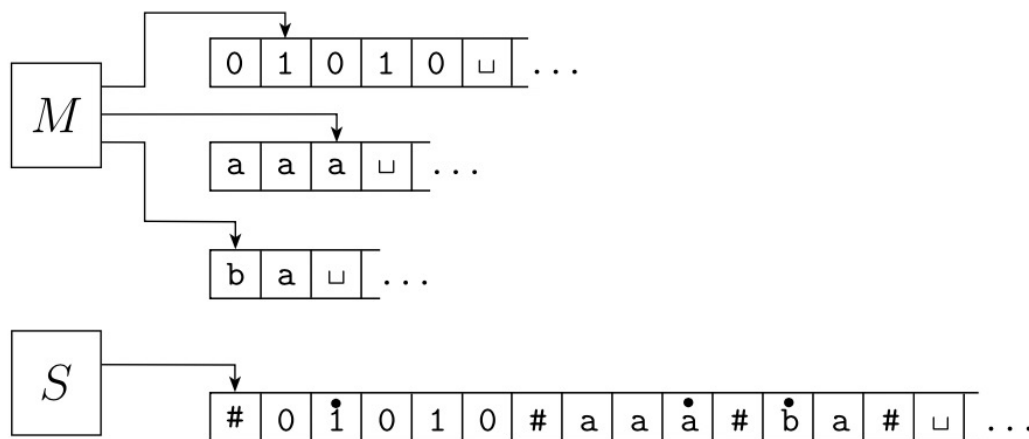
Single-tape vs multi-tape

Theorem:

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

PROOF IDEA The idea behind the proof of this theorem is quite simple. Recall that in Theorem 3.13, we showed how to convert any multitape TM into a single-tape TM that simulates it. Now we analyze that simulation to determine how much additional time it requires. We show that simulating each step of the multitape machine uses at most $O(t(n))$ steps on the single-tape machine. Hence the total time used is $O(t^2(n))$ steps.

Simulation of multi-tape on single tape TM (old slide from lecture on TMs)



Simulation Idea:

- Say M has k tapes
- S simulates the effect of k tapes by storing their information on its single tape
- **Delineate tapes:** S uses the new symbol $\#$ as a delimiter to separate contents of the different tapes
- **Track tape heads:** S writes a dot above a tape symbol to mark the place where the head on that tape would be