CMSC 28100

Introduction to Complexity Theory

Spring 2024
Instructor: William Hoza

Asymptotic analysis

Big-O, big-Ω, and big-Θ

- Let $T, f: \mathbb{N} \to \mathbb{N}$ be any two functions
- We say that T is $O(f)$ if there exist $C, n_* \in \mathbb{N}$ such that for every $n > n_*$, we have $T(n) \leq C \cdot f(n)$
- We say that T is $\Omega(f)$ if there exist $c \in (0,1)$ and $n_* \in \mathbb{N}$ such that for every $n > n_*$, we have $T(n) \geq c \cdot f(n)$
- We say that T is $\Theta(f)$ if T is both $O(f)$ and $\Omega(f)$ simultaneously

Little-o notation

- Let $T, f: \mathbb{N} \to \mathbb{N}$ be any two functions
- We say that T is $o(f)$ if for every $c \in (0,1)$, there exists $n_* \in \mathbb{N}$ such that for every $n > n_*$, we have $T(n) < c \cdot f(n)$
- Equivalent:

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = 0$$

Little-ω notation

- Let $T, f: \mathbb{N} \to \mathbb{N}$ be any two functions
- We say that T is $\omega(f)$ if for every $C \in \mathbb{N}$, there exists $n_* \in \mathbb{N}$ such that for every $n > n_*$, we have $T(n) > C \cdot f(n)$
- Equivalent:

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty$$

Summary of asymptotic notation

Notation	In words	Analogy
T is $o(f)$	$T(n)$ grows more slowly than $f(n)$	$<$
T is $O(f)$	$T(n)$ is at most $C \cdot f(n)$	$\leq$
T is $\Theta(f)$	$T(n)$ and $f(n)$ grow at the same rate	$=$
T is $\Omega(f)$	$T(n)$ is at least $c \cdot f(n)$	$\geq$
T is $\omega(f)$	$T(n)$ grows more quickly than $f(n)$	$>$

Note: Big-O is not just for time complexity!

- We can use asymptotic notation (big-O, etc.) any time we are trying to understand some kind of "scaling behavior"
- For example, let G be a simple undirected graph with N vertices
  - G has $O(N^2)$ edges
  - If G is connected, then G has $\Omega(N)$ edges
- Admittedly, we are especially interested in time complexity…

Exponential vs. polynomial

- We are especially interested in the distinction between a polynomial time complexity, such as $T(n) = n^2$, and an exponential time complexity, such as $T(n) = 2^n$
- We write $T(n) = \mathrm{poly}(n)$ if there is some k such that $T(n) = O\left(n^k\right)$
- Exponentials grow much faster than polynomials!

Exponential vs. polynomial

Claim: For every constant $k \in \mathbb{N}$, we have $n^k = o(2^n)$

- **Proof:** If $n \geq k$, then

$$2^n = \#\text{ subsets of } \{1, 2, \dots, n\} = \sum_{i=0}^{n} \binom{n}{i} \geq \binom{n}{k} \geq \left(\frac{n}{k}\right)^k$$

- Therefore, for every k, we have $2^n = \Omega\left(n^k\right)$

- Therefore, for every k, we have $2^n = \Omega\left(n^{k+1}\right) = \omega\left(n^k\right)$

Which problems

can be solved

through computation?

The complexity class P

- **Definition:** For any function $T: \mathbb{N} \rightarrow \mathbb{N}$, we let $\mathrm{TIME}(T)$ denote the class of all languages that can be decided in time $O(T)$
- **Definition:** We let P denote the class of all languages that can be decided in time $\mathrm{poly}(n)$, i.e., in polynomial time

$$\mathrm{P} = \bigcup_{k=1}^{\infty} \mathrm{TIME}(n^k)$$

P: Our model of tractability

- Let L be a language
- If $L \in \mathrm{P}$, then we will consider L "tractable"
- If $L \notin \mathrm{P}$, then we will consider L "intractable"

Example: Primality testing

- $\mathrm{PRIMES} = \{\langle N \rangle : N \text{ is a prime number}\}$

Theorem: $\mathrm{PRIMES} \in \mathrm{P}$

- **Proof attempt:** For $M = 2, 3, \ldots, N-1$, check if N/M is an integer.
- That proof is not correct. The algorithm runs in $\mathrm{poly}(N)$ time, but our time budget is only $\mathrm{poly}(n)$ where $n = |\langle N \rangle| \approx \log N$!
- The theorem is true, but the proof is beyond the scope of this course

Example: Decompositions into squares

- We designed an algorithm that decides $\mathrm{DECOMPOSABLE\text{-}INTO\text{-}SQUARES}$
- That algorithm's time complexity is $\Omega(2^n)$
- Can we conclude that $\mathrm{DECOMPOSABLE\text{-}INTO\text{-}SQUARES} \notin \mathrm{P}$?

Theorem: $\mathrm{DECOMPOSABLE\text{-}INTO\text{-}SQUARES} \in \mathrm{P}$

Decomposing into squares in polynomial time

Theorem: $\text{DECOMPOSABLE-INTO-SQUARES} \in \text{P}$

- **Proof:** We'll use an algorithm technique called "dynamic programming"
- Key observation: A nonempty string $w \in \{0,1\}^*$ can be decomposed into squares if and only if it can be written in the form $w = uy$, where u can be decomposed into squares, $|u| < |w|$, and $y \in \text{SQUARES}$

Decomposing into squares in polynomial time

- Let w be the input, $w = w_1 w_2 \dots w_n$, where $w_i \in \{0, 1\}$
- Plan: For each $i \in \{0, 1, \dots, n\}$, we will compute a Boolean value b_i that indicates whether $w_1 w_2 \dots w_i \in \text{DECOMPOSABLE-INTO-SQUARES}$

1) Let $b_0 = \text{True}$
2) For $i = 1$ to n:
   a) If there ex… $= \text{True}$
   b) Otherwise…
3) Accept if b_n is…

What should b_0 be?

A: True

B: False

C: It depends on w

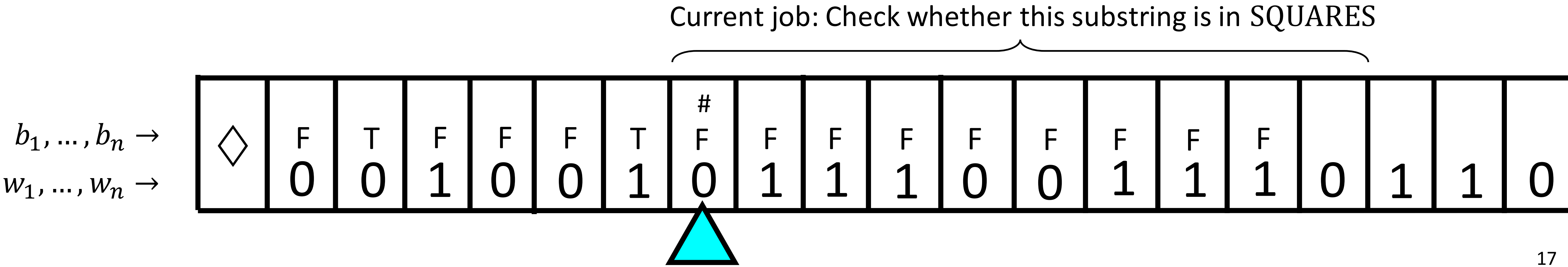
D: It's not well-defined

Respond at PollEv.com/whoza or text "whoza" to 22333

Decomposing into squares in polynomial time

1) Let $b_0 = \mathrm{True}$
2) For $i = 1$ to n:
    a) If there exists $j < i$ such that b_{j-1} is True and $w_j \dots w_i \in \mathrm{SQUARES}$, then set $b_i = \mathrm{True}$
    b) Otherwise, set $b_i = \mathrm{False}$
3) Accept if b_n is True; reject if b_n is False

- TM implementation: Store b_i in w_i's cell, and write # in w_j's cell



Decomposing into squares in polynomial time

1) Let $b_0 = \text{True}$
2) For $i = 1$ to n:
   a) If there exists $j < i$ such that b_{j-1} is True and $w_j \dots w_i \in \text{SQUARES}$, then set $b_i = \text{True}$
   b) Otherwise, set $b_i = \text{False}$
3) Accept if b_n is True; reject if b_n is False

- Outer loop (i) does $O(n)$ iterations; inner loop (j) does $O(n)$ iterations
- We can check whether $w_j \dots w_i \in \text{SQUARES}$ in time $O(n^2)$
- Total time complexity: $O(n^4) = \text{poly}(n)$ ✓

Time complexity: Theory vs. practice

- **OBJECTION:** "In an algorithms course, or in the computing industry, we would say that we can decide $\mathrm{SQUARES}$ in time $O(n)$ rather than $O(n^2)$."

Given an array of bits y:

1) For $i = 1$ to $n/2$: ← $O(n)$ iterations

    a) If $y[i] \neq y[i + n/2]$, reject ← $O(1)$ time per iteration "in practice"

2) Accept

- Is there something wrong with the Turing machine model?

Time complexity: Theory vs. practice

- **RESPONSE:** In this course, we are not concerned about the distinction between $O(n)$ time and $O(n^2)$ time
- The Turing machine model is arguably inappropriate for studying fine-grained time complexity… but that's not our mission
- We are trying to understand the boundary between tractable and intractable. Feasible vs. infeasible.
- Is the algorithm usable, or is it so slow that it's practically worthless?

Is the Turing machine model a good model?

- Switching between two reasonable models of computation can sometimes make the difference between linear time and quadratic time
- Could it ever make the difference between polynomial time and exponential time?
- For example, what happens if we use a multi-tape Turing machine instead of a single-tape Turing machine?

Multi-tape Turing machines, revisited

- Let L be a language, let k be a positive integer, and let $T: \mathbb{N} \to \mathbb{N}$

Theorem: If there is a k-tape Turing machine that decides L with time complexity T, then there is a 1-tape Turing machine that decides L with time complexity $O(T^2 + T \cdot n)$.

- Note: If $T(n) = O\left(n^k\right)$, then $O(T^2 + T \cdot n) = O\left(n^{2k}\right) = \mathrm{poly}(n)$

Efficiently simulating k tapes using one tape

- **Proof sketch:** Let M be the k-tape Turing machine deciding L
- Recall our simulation of M by a one-tape machine…
- To simulate step i, we scan back and forth over $n + i$ cells of the tape
- Therefore, simulating one step of M takes $O(n + T(n))$ steps
- Overall time complexity: $T(n) \cdot O(n + T(n))$

Robustness of P

- Conclusion: We could define P using one-tape Turing machines or using multi-tape Turing machines
- Either way, we get the exact same set of languages

Robustness of P

- Similarly, many other "realistic" models of computation can be simulated by one-tape Turing machines with at most a polynomial slowdown
  - TMs with two-way-infinite tapes
  - TMs with two-dimensional tape
  - TMs that can "teleport" to a specified location on their tape in a single step
- The complexity class P is extremely robust against modifications to the model of computation

Note on standards of rigor

- Going forward, when we analyze specific algorithms, we will not actually prove that they run in polynomial time… we will just assert it
  - In each case, one can rigorously prove the time bound by describing a TM implementation and reasoning about the motions of the heads… but this is tedious
  - Note: We still insist on proofs of correctness, just not efficiency
- You should follow this convention on problem set 5 and beyond
- Nevertheless, the Turing machine model remains extremely valuable for us, because it tells us what an arbitrary poly-time algorithm looks like!