Foundations of Computing

Lecture 19

Arkady Yerukhimovich

April 1, 2025

Computability Theory

- Studies what problems can be computed – i.e., decided

Computability Theory

- Studies what problems can be computed – i.e., decided
- Tells us that some problems are undecidable – HALT, A_{TM}, etc.

Computability Theory

- Studies what problems can be computed – i.e., decided
- Tells us that some problems are undecidable – HALT, A_{TM}, etc.
- Independent of model of computation

Computability Theory

- Studies what problems can be computed – i.e., decided
- Tells us that some problems are undecidable – HALT, A_{TM}, etc.
- Independent of model of computation
  - TM = 2-tape TM = Nondeterministic TM = algorithm

Computability Theory

- Studies what problems can be computed – i.e., decided
- Tells us that some problems are undecidable – HALT, A_{TM}, etc.
- Independent of model of computation
  - TM $=$ 2-tape TM $=$ Nondeterministic TM $=$ algorithm

Computability Theory

- Studies what problems can be computed – i.e., decided
- Tells us that some problems are undecidable – HALT, A_{TM}, etc.
- Independent of model of computation
  - TM $=$ 2-tape TM $=$ Nondeterministic TM $=$ algorithm

Question

Suppose we want to solve a problem in real life, is knowing that it is decidable enough?

Complexity Theory

- In the real world, we need to know what problems can be solved EFFICIENTLY

Complexity Theory

- In the real world, we need to know what problems can be solved EFFICIENTLY
- That is, we need to bound the algorithm to decide L

Complexity Theory

- In the real world, we need to know what problems can be solved EFFICIENTLY
- That is, we need to bound the algorithm to decide L
  - Bounded time
  - Bounded memory / space
  - ...

Complexity Theory

- In the real world, we need to know what problems can be solved EFFICIENTLY
- That is, we need to bound the algorithm to decide L
  - Bounded time
  - Bounded memory / space
  - ...

Complexity

The study of decidability under bounded models of computation

Outline

- To measure runtime of an algorithm, we need to count the number of steps it takes

1. $f(n) = 5n^2 \log n + 10n + 6$

2. $f(n) = \begin{cases} 6n & \text{for } n < 1000 \\ 3n^3 & n \geq 1000 \end{cases}$

3. $f(n) = \begin{cases} 2^n & n < 10000000 \\ 5n & n \geq 100000000 \end{cases}$

Asymptotic Notation – Big-O

- To measure runtime of an algorithm, we need to count the number of steps it takes
- Often messy to compute exactly

Asymptotic Notation – Big-O

- To measure runtime of an algorithm, we need to count the number of steps it takes
- Often messy to compute exactly
- Instead, we want to an (approximate) upper bound as input size grows

Asymptotic Notation – Big-O

- To measure runtime of an algorithm, we need to count the number of steps it takes
- Often messy to compute exactly
- Instead, we want to an (approximate) upper bound as input size grows

Example

$$f(n) = 5n^3 + 3n^2 + 10n + 8$$

Asymptotic Notation – Big-O

- To measure runtime of an algorithm, we need to count the number of steps it takes
- Often messy to compute exactly
- Instead, we want to an (approximate) upper bound as input size grows

Example

$$f(n) = 5n^3 + 3n^2 + 10n + 8$$

- Leading term is $5n^3$

Asymptotic Notation – Big-O

- To measure runtime of an algorithm, we need to count the number of steps it takes
- Often messy to compute exactly
- Instead, we want to an (approximate) upper bound as input size grows

Example

$$f(n) = 5n^3 + 3n^2 + 10n + 8$$

- Leading term is $5n^3$
- Dropping the constant 5, we say f is asymptotically at most n^3

Asymptotic Notation – Big-O

- To measure runtime of an algorithm, we need to count the number of steps it takes
- Often messy to compute exactly
- Instead, we want to an (approximate) upper bound as input size grows

Example

$$f(n) = 5n^3 + 3n^2 + 10n + 8$$

- Leading term is $5n^3$
- Dropping the constant 5, we say f is asymptotically at most n^3
- We write $f = O(n^3)$

Asymptotic Notation – Big-O

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$, we say that $f(n) = O(g(n))$ if

- There exist positive integers c, n_0 s.t. for all $n \geq n_0$

$$f(n) \leq cg(n)$$

Asymptotic Notation – Big-O

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$, we say that $f(n) = O(g(n))$ if

- There exist positive integers c, n_0 s.t. for all $n \geq n_0$

$$f(n) \leq cg(n)$$

- We say that $g(n)$ is an upper bound on $f(n)$

Asymptotic Notation – Big-O

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$, we say that $f(n) = O(g(n))$ if

- There exist positive integers c, n_0 s.t. for all $n \geq n_0$

$$f(n) \leq cg(n)$$

- We say that $g(n)$ is an upper bound on $f(n)$
- Big-O notation will be very useful for analyzing runtime of algorithms

Asymptotic Notation – Big-O

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$, we say that $f(n) = O(g(n))$ if

- There exist positive integers c, n_0 s.t. for all $n \geq n_0$

$$f(n) \leq cg(n)$$

- We say that $g(n)$ is an upper bound on $f(n)$
- Big-O notation will be very useful for analyzing runtime of algorithms

Example

$$f(n) = 5n^3 + 3n^2 + 10n + 8$$

Asymptotic Notation – Big-O

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$, we say that $f(n) = O(g(n))$ if

- There exist positive integers c, n_0 s.t. for all $n \geq n_0$

$$f(n) \leq cg(n)$$

- We say that $g(n)$ is an upper bound on $f(n)$
- Big-O notation will be very useful for analyzing runtime of algorithms

Example

$$f(n) = 5n^3 + 3n^2 + 10n + 8$$

- $f(n) = O(n^3)$

Asymptotic Notation – Big-O

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$, we say that $f(n) = O(g(n))$ if

- There exist positive integers c, n_0 s.t. for all $n \geq n_0$

$$f(n) \leq cg(n)$$

- We say that $g(n)$ is an upper bound on $f(n)$
- Big-O notation will be very useful for analyzing runtime of algorithms

Example

$$f(n) = 5n^3 + 3n^2 + 10n + 8$$

- $f(n) = O(n^3)$
- For every $n \geq 6$, $f(n) \leq 6n^3$

Asymptotic Notation – Big-O

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$, we say that $f(n) = O(g(n))$ if

- There exist positive integers c, n_0 s.t. for all $n \geq n_0$

$$f(n) \leq cg(n)$$

- We say that $g(n)$ is an upper bound on $f(n)$
- Big-O notation will be very useful for analyzing runtime of algorithms

Example

$$f(n) = 5n^3 + 3n^2 + 10n + 8$$

- $f(n) = O(n^3)$
- For every $n \geq 6$, $f(n) \leq 6n^3$
- I.e., $n_0 = 6, c = 6$

Asymptotic Notation – Big-O

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$, we say that $f(n) = O(g(n))$ if

- There exist positive integers c, n_0 s.t. for all $n \geq n_0$

$$f(n) \leq cg(n)$$

- We say that $g(n)$ is an upper bound on $f(n)$
- Big-O notation will be very useful for analyzing runtime of algorithms

Example

$$f(n) = 5n^3 + 3n^2 + 10n + 8$$

- $f(n) = O(n^3)$
- For every $n \geq 6$, $f(n) \leq 6n^3$
- I.e., $n_0 = 6, c = 6$
- Note that $f(n) = O(n^4)$

Time Complexity

- Roughly, time complexity is the number of “steps” a TM must take to decide a language L

Time Complexity

- Roughly, time complexity is the number of “steps” a TM must take to decide a language L
- Of course, this depends on the input – some inputs are easier than others

Time Complexity

- Roughly, time complexity is the number of "steps" a TM must take to decide a language L
- Of course, this depends on the input – some inputs are easier than others

Worst-Case Complexity

The time complexity of L is the maximum number of steps taken by a TM M to decide whether $x \in L$ for any x.

- Runtime measured as a function of $|x|$

Time Complexity

- Roughly, time complexity is the number of "steps" a TM must take to decide a language L
- Of course, this depends on the input – some inputs are easier than others

Worst-Case Complexity

The time complexity of L is the maximum number of steps taken by a TM M to decide whether $x \in L$ for any x.

- Runtime measured as a function of $|x|$

Time Complexity Classes

Let $t : \mathbb{N} \to \mathbb{N}$. Define time complexity class $TIME(t(n))$ as

$$TIME(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time TM}\}$$

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:
$M_1 =$ On input string w

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:

$M_1 =$ On input string w

1. Scan the tape and reject if 0 found after a 1

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:

$M_1 =$ On input string w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
    - Scan the tape, crossing off one 0 and one 1

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:

$M_1 =$ On input string w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape, crossing off one 0 and one 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:
M_1 = On input string w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape, crossing off one 0 and one 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:

$M_1 =$ On input string w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape, crossing off one 0 and one 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

- Step 1 takes $O(n)$ steps

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:

M_1 = On input string w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape, crossing off one 0 and one 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

- Step 1 takes $O(n)$ steps
- Step 2 runs at most $n/2$ times, each time requiring $O(n)$ steps

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:

$M_1 =$ On input string w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape, crossing off one 0 and one 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

- Step 1 takes $O(n)$ steps
- Step 2 runs at most $n/2$ times, each time requiring $O(n)$ steps
- Step 3 takes $O(n)$ steps

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:

M_1 = On input string w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape, crossing off one 0 and one 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

- Step 1 takes $O(n)$ steps
- Step 2 runs at most $n/2$ times, each time requiring $O(n)$ steps
- Step 3 takes $O(n)$ steps
- Total: $O(n) + (n/2) \cdot O(n) + O(n) = O(n^2)$

An Example

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_1:

M_1 = On input string w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape, crossing off one 0 and one 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

- Step 1 takes $O(n)$ steps
- Step 2 runs at most $n/2$ times, each time requiring $O(n)$ steps
- Step 3 takes $O(n)$ steps
- Total: $O(n) + (n/2) \cdot O(n) + O(n) = O(n^2)$
- $L_1 \in TIME(n^2)$

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

$M_2 =$ On input w

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

$M_2 =$ On input w

1. Scan the tape and reject if 0 found after a 1

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

$M_2 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape and see if $\#0's + \#1's$ is odd, if so reject

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

$M_2 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
    - Scan the tape and see if $\#0's + \#1's$ is odd, if so reject
    - Scan the tape again, crossing off every other 0 and every other 1

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

$M_2 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
    - Scan the tape and see if $\#0's + \#1's$ is odd, if so reject
    - Scan the tape again, crossing off every other 0 and every other 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

$M_2 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape and see if $\#0's + \#1's$ is odd, if so reject
   - Scan the tape again, crossing off every other 0 and every other 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

$M_2 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape and see if $\#0's + \#1's$ is odd, if so reject
   - Scan the tape again, crossing off every other 0 and every other 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

- Step 1 takes $O(n)$ steps

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

$M_2 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape and see if $\#0's + \#1's$ is odd, if so reject
   - Scan the tape again, crossing off every other 0 and every other 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

- Step 1 takes $O(n)$ steps
- Step 2 runs at most $\log_2(n)$ times, taking $O(n)$ steps each time

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

$M_2 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape and see if $\#0's + \#1's$ is odd, if so reject
   - Scan the tape again, crossing off every other 0 and every other 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

- Step 1 takes $O(n)$ steps
- Step 2 runs at most $\log_2(n)$ times, taking $O(n)$ steps each time
- Step 3 takes $O(n)$ steps

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:
$M_2 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape and see if $\#0's + \#1's$ is odd, if so reject
   - Scan the tape again, crossing off every other 0 and every other 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

- Step 1 takes $O(n)$ steps
- Step 2 runs at most $\log_2(n)$ times, taking $O(n)$ steps each time
- Step 3 takes $O(n)$ steps
- Total: $O(n) + \log_2 n \cdot -O(n) + O(n) = O(n \log n)$

Can We Do Better?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following TM M_2:

$M_2 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. While both 0s and 1s remain on the tape
   - Scan the tape and see if $\#0's + \#1's$ is odd, if so reject
   - Scan the tape again, crossing off every other 0 and every other 1
3. If only 0s or only 1s left on the tape, reject. If no symbols left on the tape, accept.

Counting number of steps on $|w| = n$:

- Step 1 takes $O(n)$ steps
- Step 2 runs at most $\log_2(n)$ times, taking $O(n)$ steps each time
- Step 3 takes $O(n)$ steps
- Total: $O(n) + \log_2 n \cdot O(n) + O(n) = O(n \log n)$
- $L_1 \in TIME(n \log n)$

Can We Do Even Better?

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$
- What about on a 2-tape TM?

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$
- What about on a 2-tape TM?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$
- What about on a 2-tape TM?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following 2-tape TM M_3:

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$
- What about on a 2-tape TM?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following 2-tape TM M_3:
$M_3 =$ On input w

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$
- What about on a 2-tape TM?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following 2-tape TM M_3:

$M_3 =$ On input w

1. Scan the tape and reject if 0 found after a 1

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$
- What about on a 2-tape TM?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following 2-tape TM M_3:

$M_3 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. Scan the 0s until the first 1 copying all 0s to tape 2

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$
- What about on a 2-tape TM?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following 2-tape TM M_3:

$M_3 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. Scan the 0s until the first 1 copying all 0s to tape 2
3. Scan across all 1s on tape 1.
   - For each 1 on tape 1, cross off a 0 on tape 2

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$
- What about on a 2-tape TM?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following 2-tape TM M_3:

$M_3 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. Scan the 0s until the first 1 copying all 0s to tape 2
3. Scan across all 1s on tape 1.
    - For each 1 on tape 1, cross off a 0 on tape 2
    - If all 0s are crossed off before all 1s are done, reject

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$
- What about on a 2-tape TM?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following 2-tape TM M_3:

$M_3 =$ On input w

1. Scan the tape and reject if 0 found after a 1
2. Scan the 0s until the first 1 copying all 0s to tape 2
3. Scan across all 1s on tape 1.
   - For each 1 on tape 1, cross off a 0 on tape 2
   - If all 0s are crossed off before all 1s are done, reject
4. If any 0s remain, reject. If no symbols remain, accept

Can We Do Even Better?

- On a 1-tape TM cannot do better than $O(n \log n)$
- What about on a 2-tape TM?

$L_1 = \{0^k 1^k \mid k \geq 0\}$

L_1 can be decided by the following 2-tape TM M_3:

M_3 = On input w

1. Scan the tape and reject if 0 found after a 1
2. Scan the 0s until the first 1 copying all 0s to tape 2
3. Scan across all 1s on tape 1.
   - For each 1 on tape 1, cross off a 0 on tape 2
   - If all 0s are crossed off before all 1s are done, reject
4. If any 0s remain, reject. If no symbols remain, accept

$O(n)$

Important

Time complexity depends on the exact model of computation

Dependence on Model of Computation

Theorem

For any function $t(n) \geq n$, every multi-tape TM (with $O(1)$ tapes) running in time $t(n)$ has an equivalent 1-tape TM running in time $O(t^2(n))$.

Polynomial Time

Efficient Computation

We define computation to be efficient if it runs in time bounded by some polynomial of the input size n

Polynomial Time

Efficient Computation

We define computation to be efficient if it runs in time bounded by some polynomial of the input size n

Why polynomial:

Polynomial Time

Efficient Computation

We define computation to be efficient if it runs in time bounded by some polynomial of the input size n

Why polynomial:

- Polynomials grow much slower than exponentials:

Polynomial Time

Efficient Computation

We define computation to be efficient if it runs in time bounded by some polynomial of the input size n

Why polynomial:

- Polynomials grow much slower than exponentials:
  - $f(n) = n^3$: If $n = 1000$, $f(n) = 1,000,000,000$ – large, but not unreasonable for today's PCs
  - $f(n) = 2^n$: If $n = 1000$, $f(n) >$ number of atoms in the universe

Polynomial Time

Efficient Computation

We define computation to be efficient if it runs in time bounded by some polynomial of the input size n

Why polynomial:

- Polynomials grow much slower than exponentials:
  - $f(n) = n^3$: If $n = 1000$, $f(n) = 1,000,000,000$ – large, but not unreasonable for today's PCs
  - $f(n) = 2^n$: If $n = 1000$, $f(n) >$ number of atoms in the universe
- All "reasonable" deterministic computation models are polynomially equivalent

Polynomial Time

Efficient Computation

We define computation to be efficient if it runs in time bounded by some polynomial of the input size n

Why polynomial:

- Polynomials grow much slower than exponentials:
  - $f(n) = n^3$: If $n = 1000$, $f(n) = 1,000,000,000$ – large, but not unreasonable for today's PCs
  - $f(n) = 2^n$: If $n = 1000$, $f(n) >$ number of atoms in the universe
- All "reasonable" deterministic computation models are polynomially equivalent
- Convenient closure properties:

Polynomial Time

Efficient Computation

We define computation to be efficient if it runs in time bounded by some polynomial of the input size n

Why polynomial:

- Polynomials grow much slower than exponentials:
  - $f(n) = n^3$: If $n = 1000$, $f(n) = 1,000,000,000$ – large, but not unreasonable for today's PCs
  - $f(n) = 2^n$: If $n = 1000$, $f(n) >$ number of atoms in the universe
- All "reasonable" deterministic computation models are polynomially equivalent
- Convenient closure properties:
  - $\text{poly}(n) + \text{poly}(n) = \text{poly}(n)$

Polynomial Time

Efficient Computation

We define computation to be efficient if it runs in time bounded by some polynomial of the input size n

Why polynomial:

- Polynomials grow much slower than exponentials:
  - $f(n) = n^3$: If $n = 1000$, $f(n) = 1,000,000,000$ – large, but not unreasonable for today's PCs
  - $f(n) = 2^n$: If $n = 1000$, $f(n) >$ number of atoms in the universe
- All "reasonable" deterministic computation models are polynomially equivalent
- Convenient closure properties:
  - $\text{poly}(n) + \text{poly}(n) = \text{poly}(n)$
  - $\text{poly}(n) \cdot \text{poly}(n) = \text{poly}(n)$ (up to $O(1)$ multiplications)

Outline

1. Polynomial Time
2. The Complexity Class $\mathcal{P}$

Complexity Class $\mathcal{P}$

Definition

$\mathcal{P}$ is the class of languages decidable in polynomial time on a 1-tape deterministic TM.

Complexity Class $\mathcal{P}$

Definition

$\mathcal{P}$ is the class of languages decidable in polynomial time on a 1-tape deterministic TM.

$$\mathcal{P} = \bigcup_k TIME(n^k)$$

Complexity Class $\mathcal{P}$

Definition

$\mathcal{P}$ is the class of languages decidable in polynomial time on a 1-tape deterministic TM.

$$\mathcal{P} = \bigcup_k TIME(n^k)$$

- $\mathcal{P}$ corresponds to the class of "efficiently-solvable" problems

Complexity Class $\mathcal{P}$

Definition

$\mathcal{P}$ is the class of languages decidable in polynomial time on a 1-tape deterministic TM.

$$\mathcal{P} = \bigcup_k TIME(n^k)$$

- $\mathcal{P}$ corresponds to the class of "efficiently-solvable" problems
- $\mathcal{P}$ is invariant for all models of computation polynomially-equivalent to 1-tape TM

Complexity Class $\mathcal{P}$

Definition

$\mathcal{P}$ is the class of languages decidable in polynomial time on a 1-tape deterministic TM.

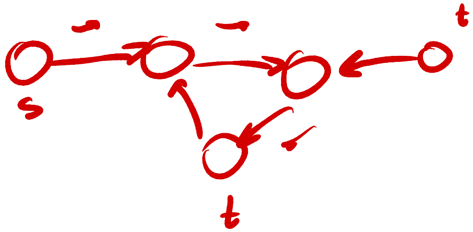
$$\mathcal{P} = \bigcup_k TIME(n^k)$$

- $\mathcal{P}$ corresponds to the class of "efficiently-solvable" problems
- $\mathcal{P}$ is invariant for all models of computation polynomially-equivalent to 1-tape TM
- $\mathcal{P}$ has nice closure properties

Problems in $\mathcal{P}$

PATH problem

$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}$



Problems in $\mathcal{P}$

RELPRIME problem

$$RELPRIME = \{\langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime integers}\}$$

Problems in $\mathcal{P}$

RELPRIME problem

$$RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime integers}\}$$

Definition: Greatest Common Divisor (GCD)
For $a, b \in \mathbb{Z}$, $gcd(a, b) = c$ s.t. c is the largest integer so that $c|a$ and $c|b$

Problems in $\mathcal{P}$

RELPRIME problem

$$RELPRIME = \{\langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime integers}\}$$

Definition: Greatest Common Divisor (GCD)
For $a, b \in \mathbb{Z}$, $gcd(a, b) = c$ s.t. c is the largest integer so that $c|a$ and $c|b$

Euclidean Algorithm:
$GCD(a, b)$:

Problems in $\mathcal{P}$

RELPRIME problem

$RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime integers}\}$

Definition: Greatest Common Divisor (GCD)
For $a, b \in \mathbb{Z}$, $gcd(a, b) = c$ s.t. c is the largest integer so that $c|a$ and $c|b$

Euclidean Algorithm:
$GCD(a, b)$:

1. If $b|a$, return b

Problems in $\mathcal{P}$

RELPRIME problem

$$RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime integers}\}$$

Definition: Greatest Common Divisor (GCD)
For $a, b \in \mathbb{Z}$, $gcd(a, b) = c$ s.t. c is the largest integer so that $c|a$ and $c|b$

Euclidean Algorithm:
$GCD(a, b)$:

1. If $b|a$, return b
2. Else, return $GCD(b, [a \bmod b])$

Problems in $\mathcal{P}$

RELPRIME problem

$$RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime integers}\}$$

Definition: Greatest Common Divisor (GCD)
For $a, b \in \mathbb{Z}$, $gcd(a, b) = c$ s.t. c is the largest integer so that $c|a$ and $c|b$

Euclidean Algorithm:
$GCD(a, b)$:

1. If $b|a$, return b
2. Else, return $GCD(b, [a \bmod b])$

$O(\log a)$ recursion

$O(|a|)$

Decision Problems vs. Search Problems

- We have defined all languages as decision problems (i.e., is $x \in L$?)

Decision Problems vs. Search Problems

- We have defined all languages as decision problems (i.e., is $x \in L$?)
- We often more naturally think of computation as search problems (i.e., find a path from s to t)

Decision Problems vs. Search Problems

- We have defined all languages as decision problems (i.e., is $x \in L$?)
- We often more naturally think of computation as search problems (i.e., find a path from s to t)
- For some complexity classes, but not all, the two are equivalent – we will talk about this more later

Next Class

- Nondeterministic computation and the class $\mathcal{NP}$

$$P \stackrel{?}{=} NP$$