# Complexity

A decidable problem is

- computationally solvable in principle,
- but not necessarily in practice.

Problem is resource consumption:

- time
- space

# Example

Consider

$$A = \{0^n 1^n | n \geq 0\}$$

Clearly this language is decidable.

**Question:** How much time does a single-tape TM need to decide it?

## Example

$M_1$: On input $w$ where $w$ is a string,

- Scan across tape and *reject* if 0 is found to the right of a 1.
- Repeat the following if both 0s and 1s appear on tape
  - scan across tape, crossing of single 0 and single 1
- If 0s still remain after all the 1s have been crossed out, or vice-versa, *reject*. Otherwise, if the tape is empty, *accept*.

## Question

So how much time does $M_1$ need?

Number of steps may depend on several parameters.

Example: if input is a graph, could depend on number of

- nodes
- edges
- maximum degree
- all, some, or none of the above!

## Question

Our Gordian knot solution:

**Definition:** Complexity is measured as function of input string length.

- worst case: longest running time on input of given length
- average case: average running time on given length

Actually, here we consider worst case.

## Definition

Let $M$ be a deterministic TM that halts on all inputs.

The *running time* of $M$ is a function

$$f : \mathcal{N} \longrightarrow \mathcal{N}$$

where $f(n)$ is the maximum running time of $M$ on input of length $n$.

Terminology

- $M$ runs in time $f(n)$
- $M$ is an $f(n)$-time TM

## Running Time

The exact running time of most algorithms is quite complex.

Better to "estimate" it.

Informally, we want to focus on "important" parts only.

Example:

- $6n^3 + 2n^2 + 20n + 45$ has four terms.
- $6n^3$ more import
- $n^3$ most important

## Asymptotic Notation

Consider functions,

$$f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$$

We say that

$$f(n) = O(g(n))$$

if there exist positive integers $c$ and $n_0$ such that

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

for $n \geq n_0$.

## Confused?

Basic idea: ignore constant factor differences.

- $617n^3 + 277x^2 + 720x + 7x = O(n^3)$.
- $2 = O(1)$
- $\sin(x) = O(1)$.

## Reality Check

Consider

$$f_1(n) = 5n^3 + 2n + 22n + 6$$

We claim that

$$f_1(n) = O(n^3)$$

Let $c = 6$ and $n_0 = 10$. Then

$$5n^3 + 2n + 22n + 6 \leq 6n^3$$

for every $n \geq 10$.

## Reality Check (Part Two)

Recall:

$$f_1(n) = 5n^3 + 2n + 22n + 6$$

- we have seen that $f_1(n) = O(n^3)$.
- also that $f_1(n) = O(n^4)$.
- but $f_1(n)$ is not $O(n^2)$, because no value for $c$ or $n_0$ works!

## Logarithms

The big-O interacts with logarithms in a particular way.

High-school identity:

$$\log_b n = \frac{\log_2 n}{\log_2 b}$$

- changing $b$ changes only constant factor
- When we say $f(n) = O(\log n)$, the base is unimportant

## Important Notation

Sometimes we will see

$$f(n) = O(n^2) + O(n).$$

Each occurrence of $O$ symbol is distinct constant.

But $O(n^2)$ term dominates $O(n)$ term, equivalent to $f(n) = O(n^2)$.

## Important Notation

Exponents are even more fun.

What does this mean?

$$f(n) = 2^{O(n)}$$

It means an upper bound of $2^{cn}$ for some constant $c$

## Important Notion

What does this mean?

$$f(n) = 2^{O(\log n)}$$

Identities

$$\begin{aligned} n &= 2^{\log_2 n} \\ n^c &= 2^{c \log_2 n} \end{aligned}$$

it follows that $2^{O(\log n)}$ means an upper bound of $n^c$ for some constant $c$

## Related Important Notions

- A bound of $n^c$, where $c > 0$, is called **polynomial**.
- A bound of $2^{(n^\delta)}$, where $\delta > 0$, is called **exponential**.

## Back to Business

$M_1$: On input string $w$,

**1.** Scan across tape and *reject* if 0 is found to the right of a 1.

**2.** Repeat the following if both 0s and 1s appear on tape

- scan across tape, crossing of single 0 and single 1

**3.** If no 0s or 1s remain, *accept*, otherwise *reject*.

## Analysis

Consider stages separately.

**1.** Scan across tape and *reject* if 0 is found to the right of a 1.

- Scanning requires $n$ steps.
- Repositioning head requires $n$ steps.
- Total is $2n = O(n)$ steps.

## Analysis

2. Repeat the following if both 0s and 1s appear on tape
   - scan across tape, crossing off single 0 and single 1
   - Each scan requires $O(n)$ steps
   - because each scan crosses off two symbols, at most $n/2$ scans can occur.
   - Total is $(n/2)O(n) = O(n^2)$ steps.

## Analysis

**3.** If 0s still remain after all 1s have been crossed out, or vice-versa, *reject*. Otherwise, if the tape is empty, *accept*.

- Single scan requires $O(n)$ steps.
- Total is $O(n)$ steps.

## Analysis

Total cost for stages

1. $O(n)$
2. $O(n^2)$
3. $O(n)$

which is $O(n^2)$

## Definition

Let

$$t : \mathcal{N} \longrightarrow \mathcal{N}$$

be a function.

**Definition:**

$$\begin{aligned}\text{TIME}(t(n)) \;=\; \{L|L \text{ is a language} \\ \text{decided by an } O(t(n))\text{-time TM}\}\end{aligned}$$

We have seen that

- $A = \{0^n 1^n | n \geq 0\}$,
- $A \in \mathsf{TIME}(n^2)$.

Can we do better?

## Home Improvement

$M_2$: On input string $w$,

**1.** Scan across tape and *reject* if 0 is found to the right of a 1.

**2.** Repeat the following if both 0s and 1s appear on tape

  **2.1** scan across tape, checking whether total number of 0s and 1s is even or odd. If odd, *reject*.

  **2.2** Scan across tape, crossing off every other 0 (starting with the first), and every other 1 (starting with the first).

**3.** If no 0s or 1s remain, *accept*, otherwise *reject*.

## Analysis

Check that $M_2$ halts.

- on each scan in step 2.2,
  - The total number of 0s is cut in half, and
  - remainder discarded.
  - Same for 1s.
- example: start with 13 0s
  - first pass: 6
  - then 3
  - then 1
  - then 0

## Analysis

Check that $M_2$ is correct.

- consider parity of 0s and 1s in 2.1
- example: start with 13 0s and 13 1s
  - odd, odd (13)
  - even, even (6)
  - odd, odd (3)
  - odd, odd (1)
- result is 1011, reverse of binary representation of 13
- each pass checks one binary digit

## Running Time

$M_2$: On input string $w$,
1. Scan across tape and *reject* if 0 is found to the right of a 1.
2. Repeat the following if both 0s and 1s appear on tape
   - 2.1 scan across tape, checking whether total number of 0s and 1s is even or odd. If odd, *reject*.
   - 2.2 Scan across tape, crossing off every other 0 (starting with the first), and every other 1 (starting with the first).
3. If no 0s or 1s remain, *accept*, otherwise *reject*.

- Each stage takes $O(n)$ time.
- stage 1 and 3: once
- stage 2.2 eliminates half of 0s and 1s: $1 + \log_2 n$ times
- total for 2 is $(1 + \log_2 n)O(n) = O(n \log n)$.
- grand total: $O(n) + O(n \log n) = O(n \log n)$.

## A Two Tape TM

$M_3$: on input string $w$

**1.** Scan across tape and *reject* if 0 is found to the right of a 1.

**2.** Scan across 0s to first 1, copying 0s to tape 2.

**3.** Scan across 1s on tape 1 until the end. For each 1, cross off a 0. If no 0s left, *reject*.

**4** If any 0s left, *reject*, otherwise *accept*. first), and every other 1 (starting with the first).

**Question:** What is the running time?

## Complexity

Deciding $\{0^n 1^n\}$:

- single-tape $M_1$: $O(n^2)$.
- single-tape $M_2$: $O(n \log n)$ (fastest possible!)
- two-tape $M_3$: $O(n)$.

Complexity and computability differ in one important way.

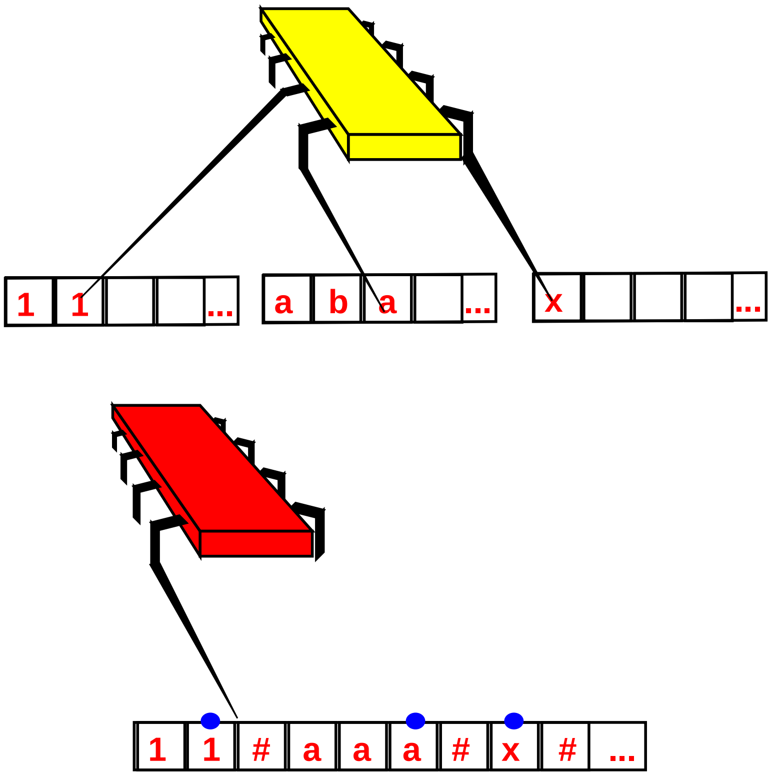
- Computability: all reasonable models equivalent (Church-Turing)
- Complexity: choice of model affects time complexity.

**Question:** How does model affect complexity?

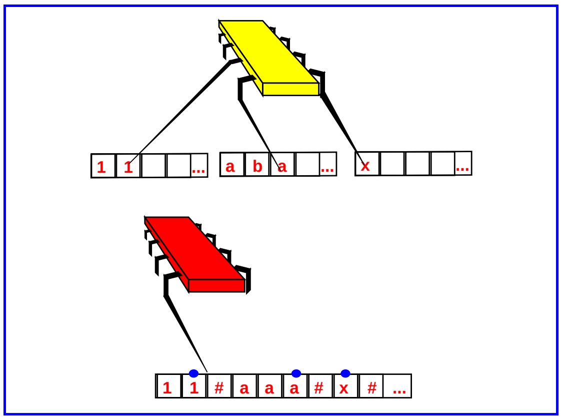
## Models and Complexity

Let $t(n)$ be a function where $t(n) \geq n$.

Any $t(n)$-time multitape TM has an equivalent $O(t^2(n))$-time single TM.



## Review:Simulation




On input $w = w_1 \cdots w_n$, $S$:

- puts on its tape:

$$\# \overset{\bullet}{w}_1\ w_2 \cdot w_n \# \overset{\bullet}{\sqcup} \# \overset{\bullet}{\sqcup} \# \cdots \#$$

- scans its tape from first # to $k + 1$-st # to read symbols under "virtual" heads.
- rescans to write new symbols and move heads
- if $S$ tries to move virtual head onto #, then $M$ takes "tape fault" and re-arranges tape.

## Complexity

For each step of $M$, $S$ performs

- two scans
- up to $k$ rightward shifts
- total of $O(t(n))$ time

total:

- initial tape arrangement $O(n)$ time
- simulates each of $O(t(n))$ steps using $O(t(n))$ steps
- for a total of $O(t(n)) \times O(t(n)) = O(t^2(n))$ steps
- grand total: $O(n) + O(t^2(n))$ steps

Assumption that $t(n) > n$ reasonable *Why?*.

## Non-Deterministic TMs

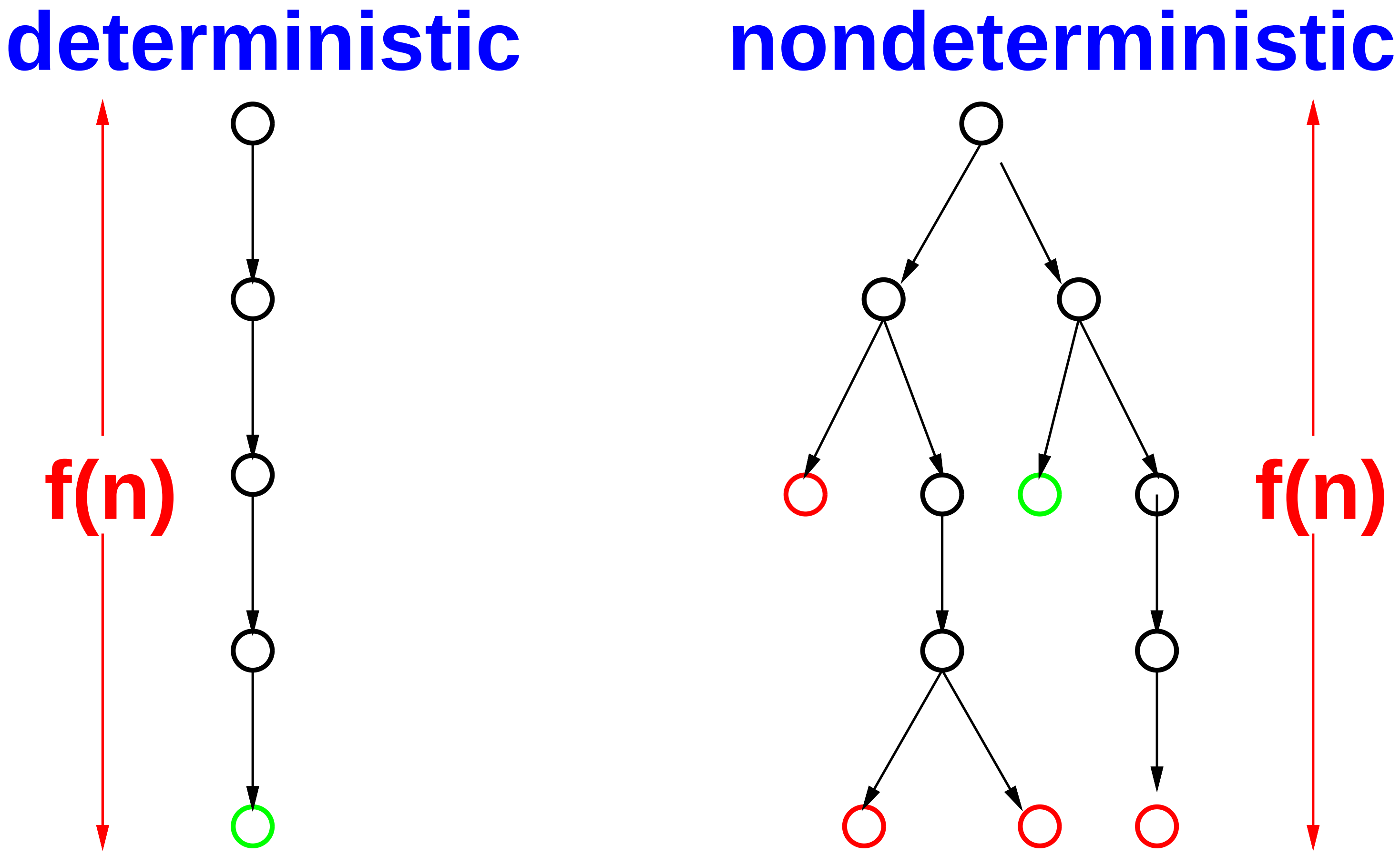
The *running time* of a non-deterministic TM $N$ is a function

$$f : \mathcal{N} \longrightarrow \mathcal{N}$$

where $f(n)$ is

- the maximum number of steps that $N$ uses
- on any branch of the computation
- on any input of size $n$.

## Non-Deterministic TMs



## Models and Complexity

Let $t(n)$ be a function where $t(n) \geq n$.

Any $t(n)$-time non-deterministic TM has an equivalent $2^{O(t(n))}$-time deterministic single-tape TM.

Note contrast with multi-tape result!

# Simulation

Let $N$ be a non-deterministic TM running in $t(m)$ time.

Simulation basic idea:

- $D$ tries all possible branches
- If $D$ finds any accepting state, it accepts.
- If all branches reject, $D$ rejects.
- If all branches reject or loop, $D$ loops.

## Simulation

$N$'s computation is a tree.

- root is starting configuration
- each node has finite fanout $\leq b$ *Why?*
- each branch has length $\leq t(n)$
- total number of leaves at most $b^{t(n)}$
- total number of nodes $O(b^{t(n)})$
- time to travel root to node is $O(t(n))$

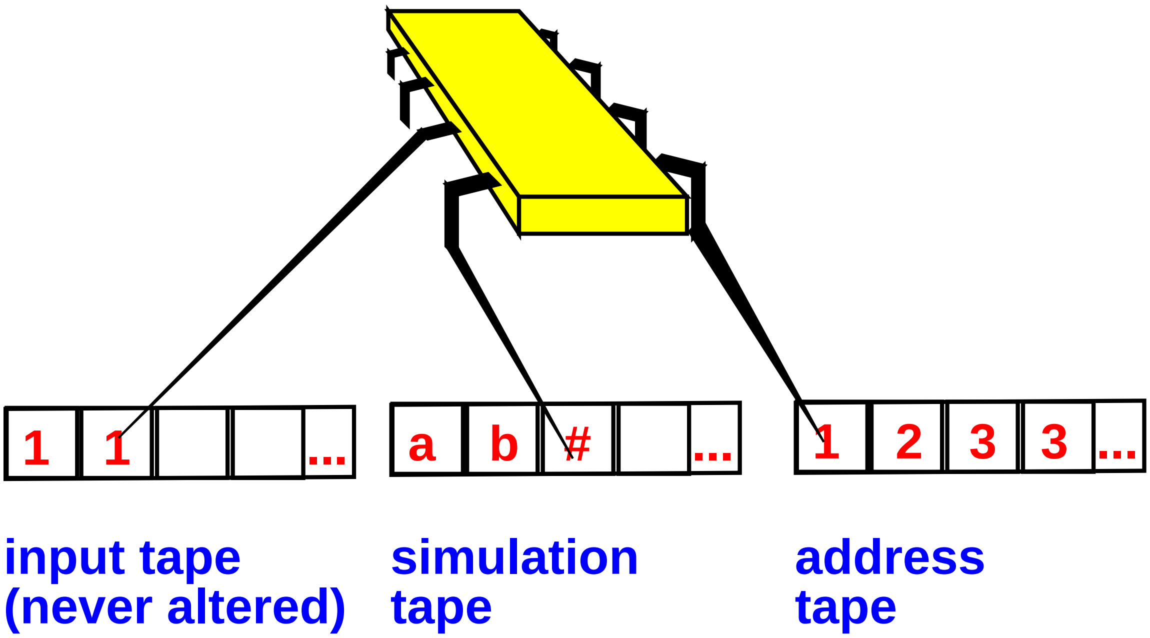
Time to visit each node: $O(t(n) \times b^{t(n)}) = O(2^{O(t(n))})$.

## Remark

Breadth-first search in simulation

- inefficiently traverses from root to visit each node
- but fixing it doesn't change anything

# Remark




- Simulation uses three-tape machine.
- Single-tape simulation:
  $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$.

## Important Distinction

- *polynomial* distinction between deterministic single- and multi-tape TMs, vs.
- *exponential* distinction between deterministic and non-deterministic TMs

## Polynomial Good, Exponential Bad

Complexity differences:

- Polynomial: small
- Exponential: large

| | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|
| $n$ | .00001 second | .00002 second | .00003 second | .00004 second | .00005 second | .00006 second |
| $n^2$ | .00001 second | .00004 second | .00009 second | .00016 second | .00025 second | .00036 second |
| $n^3$ | .00001 second | .00008 second | .027 second | .064 second | .125 second | .216 second |
| $n^5$ | .1 second | 3.2 seconds | 24.3 seconds | 1.7 minute | 5.2 minutes | 13.0 minutes |
| $2^n$ | .001 second | 1.0 second | 17.9 minutes | 12.7 days | 35.7 years | 366 centuries |
| $3^n$ | .059 second | 58 minutes | 6.5 years | 3855 centuries | $2 \cdot 10^8$ centuries | $1.3 \cdot 10^{13}$ centuries |

## Polynomial Good, Exponential Bad

**Claim:** All "reasonable" models of computation are polynomially equivalent.

Any one can simulate another with only polynomial increase in running time.

Question: is problem solvable in

- linear time? *model-specific*
- polynomial time? *model-independent*

We are interested in computation, not models *per se*!

## The Class $P$

**Definition:** $P$ is the set of languages decidable in polynomial time on deterministic single-tape TMs.

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

Important because

- Invariant for all models of computation polynomially equivalent to deterministic single-tape TM
- roughly corresponds to realistically solvable (tractable) problems.

## The Class $P$

- Invariant for all models of computation polynomially equivalent to deterministic single-tape TM
  - not affected by particulars of model . . .
  - go ahead, have another tape, they're pretty small . . .

# The Class $P$

- roughly corresponds to realistically solvable (tractable) problems.
  - actually depends on context
  - going from exponential to polynomial algorithm usually requires major insight,
  - if you find an inefficient polynomial algorithm, you can often find an efficient one.

## Examples

Typical analysis:

- break algorithm into stages
- check that each stage is polynomial
- check that number of stages is polynomial
- Ergo, the algorithm is polynomial.

# Encoding

For numbers

- binary good
- unary bad (exponentially worse)

Graphs

- list of nodes and edges (good)
- adjacency matrix (good)

# Path

Given

- directed graph $G$
- nodes $s$ and $t$
- is there a path from $s$ to $t$?

$$\text{PATH} = \{\langle G, s, t\rangle | G \text{ has directed path from } s \text{ to } t\}$$

**Theorem:**

$$\text{PATH} \in P$$

When in doubt, try brute force.

- let $m$ be the number of nodes in $G$
- any path from $s$ to $t$ need not repeat nodes
- examine each path in $G$ of length $\leq m$,
- check if it goes from $s$ to $t$.

**Question:** What is the complexity of this algorithm?

# Path

- let $m$ be the number of nodes in $G$
- any path from $s$ to $t$ need not repeat nodes
- examine each path in $G$ of length $\leq m$,
- check if it goes from $s$ to $t$.
- there are $m^m$ possible paths
- exponential in number of nodes
- exponential in input size

Uh, oh. Try again.

**Theorem:**

$$\text{PATH} \in P$$

**1.** Place mark on $s$

**2.** Repeat until no additional nodes marked:

- scan edges of $G$.
- If edge $(a, b)$ found from marked $a$ to unmarked $b$,
- then mark $b$.

**3.** If $t$ marked, *accept*, otherwise *reject*.

**Question:** What is the complexity of this algorithm?

## Path

1. Place mark on $s$
2. Repeat until no additional nodes marked:
   - scan edges of $G$.
   - If edge $(a, b)$ found from marked $a$ to unmarked $b$,
   - then mark $b$.
3. If $t$ marked, *accept*, otherwise *reject*.

How many stages?

- Stages 1 and 3 run once.
- Stage 2 runs at most $m$ times, because each time (except last) it marks new node.

Total number of stages is polynomial.

## Path

1. Place mark on $s$
2. Repeat until no additional nodes marked:
   - scan edges of $G$.
   - If edge $(a, b)$ found from marked $a$ to unmarked $b$,
   - then mark $b$.
3. If $t$ marked, *accept*, otherwise *reject*.

How much is each stage?

- Stages 1 and 3 polynomial.
- Stage 2 scans input, also polynomial.

Total time complexity is polynomial.

## Example

Two numbers are *relatively prime* if 1 is the largest integer that evenly divides them both.

- 10 and 21 are relatively prime
- 10 and 22 are not.

**Definition:**

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

## Example

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

**Idea**: Search through all possible divisors and test.

If $x$ is unary:

- size of $\langle x\rangle$ is $x$
- testing $x$ values is polynomial

If $x$ is binary:

- size of $\langle x\rangle$ is $\log x$
- testing $x$ values is exponential!

**Important:** This algorithm is *pseudo-polynomial*.

## Example

RELPRIME $= \{\langle x, y\rangle |$ $x$ and $y$ are relatively prime$\}$

Euclidian Algorithm:

On input $\langle x, y\rangle$

1. Repeat until $y = 0$
   - $x \leftarrow \quad \pmod{()} x, y)$
   - exchange $x$ and $y$
2. output $x$

$R$: on input $\langle x, y\rangle$

- Run $E$ on $\langle x, y\rangle$
- if the result is 1, *accept*, otherwise *reject*.

## Euclidian Algorithm

Enough to check that Euclidian Algorithm is polynomial.

- each execution of Stage 1 cuts $x$ by at least half
- after each loop $x < y$
- values swapped
- number of stages $\min(\log_2 x, \log_2 y)$

total running time is polynomial.

## Context-Free Languages

**Theorem:** Every CFL is in $P$.

We know it's decidable: let's review how.

## Chomsky Normal Form

Every rule has the form

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \\ S &\rightarrow \varepsilon \end{aligned}$$

where $S$ is the start symbol, $A$, $B$ and $C$ are any variable, except $B$ and $C$ not the start symbol.

## Theorem (continued)

**Lemma:** If $G$ is in Chomsky normal form and $|w| = n$, then $w$ has a derivation of length $2n - 1$ or less.

**Idea:**

- convert $G$ to Chomsky normal form
- need only consider finite number of derivations

# Algorithm

**Theorem:** $A_{\mathsf{CFG}}$ is a decidable language.

On input $\langle G, w \rangle$, where $G$ is a grammar and $w$ a string,

**1.** Convert $G$ to Chomsky normal form.

**2.** List all derivations with $2n - 1$ steps, were $n = |w|$.

**3.** If any generates $w$, *accept*, otherwise *reject*.

## Algorithm

**Question:** What is the complexity of this algorithm?

**Answer:** The number of derivations of length $k$ may be exponential in $k$.

Need to try something else.

## Dynamic Programming

Powerful technique.

- basic idea: remember and reuse subproblem solutions
- store them in a table as they are generated
- look them up when needed

## Dynamic Programming

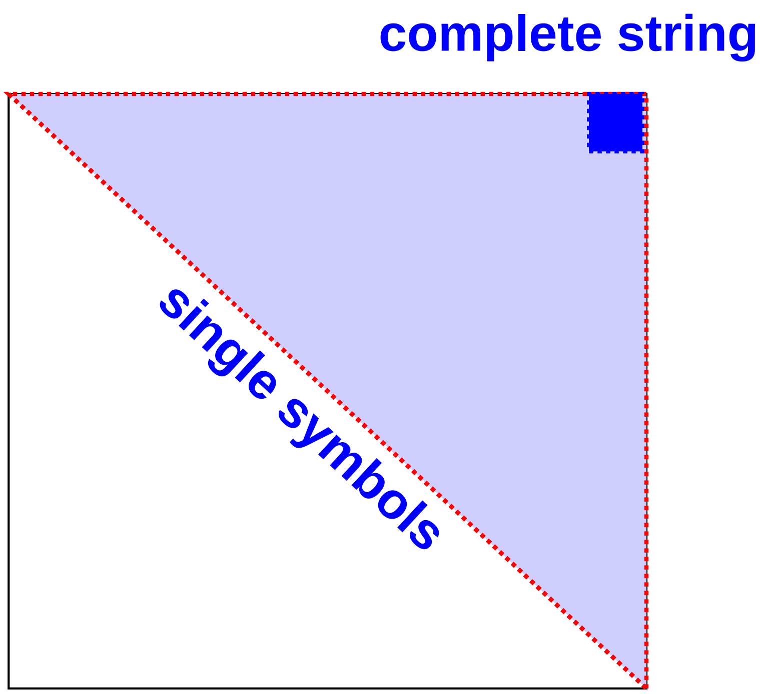
Input is $\langle G, w \rangle$.

- $G$ is CFG
- $w$ is a string

Create a table

- whether each variable of $G$ generates each substring of $w$
- $n \times n$ table
- for $i \leq j$, entry $(i, j)$ contains variables known to generate substring $w_i w_{i+1} \cdots w_j$.
- for $i > j$, entry unused.

# Dynamic Programming



## Dynamic Programming

Input is $\langle G, w \rangle$.

- $G$ is CFG
- $w$ is a string

Algorithm

- fills in table entries for each substring
- first fills in entries for length 1
- then length 2, . . .
- uses shorter lengths to help construct longer lengths

## Dynamic Programming

For example, suppose algorithm has figured out which variables generate each substring of length $k$.

To determine whether $A$ generates substring of length $k + 1$

- split substring into 2 non-empty strings
- in all $k$ ways.
- For each split, algorithm examines rules $A \to BC$
- to see if $B$ generates first piece
- and $C$ generates second,
- by consulting table.
- If so, adds it to the table.

## The Algorithm

On input $w = w_1 \cdots w_n$

1. If $w = \varepsilon$ and $S \to \varepsilon$ is a rule in $G$, then *accept*.
2. For $i = 1$ to $n$ (length 1 substrings)
3. For each variable $A$
4. Test whether $A \to b$ is a rule, where $b = w_i$
5. If so, place $A$ in *table*$(i, i)$
6. For $\ell = 2$ to $n$ ($\ell$ is substring length)
7. For $i = 1$ to $n - \ell + 1$ (start position of substring)
8. Let $j = i + \ell - 1$ (end position of substring)
9. For $k = i$ to $j - 1$ (split position)
10. For each rule $A \to BC$
11. If *table*$(i, k)$ contains $B$ and *table*$(k + 1, j)$ contains $C$, put $A$ in *table*$(i, j)$
12. If $S$ is in *table*$(1, n)$ *accept*, else *reject*.

## Complexity

Each stage is polynomial

**4.** Test whether $A \to b$ is a rule, where $b = w_i$
**5.** If so, place $A$ in *table*$(i, i)$

4 and 5 run at most $nv$ times

- where $v$ is number of variables in $G$
- $v$ is independent of $n$
- thus 4 and 5 run $O(n)$ times

# Complexity

6. For $\ell = 2$ to $n$ ($\ell$ is substring length)
7. For $i = 1$ to $n - \ell + 1$ (start position of substring)
8. Let $j = i + \ell - 1$ (end position of substring)
9. For $k = i$ to $j - 1$ (split position)
10. For each rule $A \rightarrow BC$
11. If *table*$(i, k)$ contains $B$ and *table*$(k + 1, j)$ contains $C$, put $A$ in *table*$(i, j)$

Each time

- 6 runs, 7 runs at most $n$ times
- 7 runs, 8 and 9 run at most $n$ times
- 9 runs, 10 runs $r$ times ($r$=# rules)
- 11 runs $O(n^3)$ times

Grand total: $O(n^3)$.