

Computational Models - Lecture 11

Fall 04/05

- A Short Review of Last Class
- Deterministic Time Classes
- NonDeterministic Time Classes
- Relationship between Deterministic/Nondeterministic Time
- The classes **P** and **NP**
- Examples of Problems in **P** and in **NP**
- Verifiability

- Sipser, Chapter 7

Short Review of Last Class

Three major topics:

- Undecidability of **Tiling** (Domino) Problems
- The **Recursion** Theorems.
- Unrestricted Grammars and TMs

Undecidability of Tiling Problems

Our domain is tilings of regions in the two dimensional plane by colored tiles. A tiling of any region in the plane is called **legal** if every edge shared by two tiles is colored by the **same** color.

An **instance** of the tiling problem consists of pairs: (set of tiles, initial tiling).

- A finite set $\{S_1, \dots, S_k\}$ of tiles, where each tile has rectangular shape, and each of its four edge is colored (colors on different edges need not be distinct).
- An **initial tiling**: A finite row of tiles $\{R_1, \dots, R_k\}$, such that for each j , $R_j \in \{S_1, \dots, S_k\}$, and the consecutive row of tiles $\{R_1, \dots, R_k\}$ is legal.

Undecidability of Tiling Problems

$L_\varepsilon \triangleq \{ \langle M \rangle \mid M \text{ halts on the empty input string} \}$.

Theorem: $\overline{L_\varepsilon} \leq_m L_{\text{tile}}$.

Since $L_\varepsilon \notin \mathcal{R}$ (why?), the reduction implies that $L_{\text{tile}} \notin \mathcal{R}$ as well.

The Recursion Theorem

Theorem: Let $\varphi : \mathcal{N} \longrightarrow \mathcal{N}$ be a total recursive function, thought of as a function from TM indices to TM indices. There exist an index x_0 such that for all y ,

$$f_{x_0}(y) = f_{\varphi(x_0)}(y) .$$

Such x_0 is called a **fixed point** of φ . The function φ can be viewed as modifying TMs descriptions.

The Recursion Theorem: Proof

Theorem: Let $\varphi : \mathcal{N} \longrightarrow \mathcal{N}$ be a total recursive function, thought of as a function from TM indices to TM indices. There is an index x_0 such that for all y , $f_{x_0}(y) = f_{\varphi(x_0)}(y)$.

Proof (first half):

- For each integer i construct a TM, M , that on input y computes $f_i(i)$.
- Then M runs the $f_i(i)$ -th TM on y .
- Let $g(i)$ be the index of M 's encoding.
- For all i and y , $f_{g(i)}(y) = f_{f_i(i)}(y)$.
- Notice that $g : \mathcal{N} \longrightarrow \mathcal{N}$ is a **total** computable function, even if $f_i(i)$ is **not defined** (in such case $f_i(i)$ on y is **not defined**).

The Recursion Theorem: Proof ?

Part of proof:

- For each integer i construct a TM, M , that on input y computes $f_i(i)$.
- Then M runs the $f_i(i)$ -th TM on y .
- Let $g(i)$ be the index of M 's encoding.
- For all i and y , $f_{g(i)}(y) = f_{f_i(i)}(y)$.

Complaint: But this is cheating. If $f_i(i)$ is undefined, neither is $g(i)$.

The Recursion Theorem: Proof ?!!!

Complaint: But this is cheating. If $f_i(i)$ is undefined, neither is $g(i)$.

Reply: Is that so? What about the following TM, M_w :

"on input x :

Run $\langle M \rangle$ on w

if halts, output $x^2 \bmod 13$."

Notice that $\langle M_w \rangle$ is well defined even if $\langle M \rangle$ does not halt on w .

Exactly the same is true for $g(i)$ being well defined even if $f_i(i)$ is undefined!

The Recursion Theorem

Theorem: Let $\varphi : \mathcal{N} \longrightarrow \mathcal{N}$ be a total recursive function, thought of as a function from TM indices to TM indices. There exist an index x_0 such that for all y ,

$$f_{x_0}(y) = f_{\varphi(x_0)}(y) .$$

- The recursion theorem can be used to prove that A_{TM} is **undecidable**.
- It can be used to prove **Rice theorem**.
- It can be used to prove existence of a **self printing program**, one that **generates a copy of its own source text**.

Unrestricted Grammars

- Set of rules $R \subset (V^*(V - \Sigma)V^* \times V^*)$
- In an Unrestricted Grammar, the left-hand side of a rule contains a string of terminals and non-terminals (at least one of which must be a non-terminal)
- Rules are applied just like CFGs:
 - Find a substring that matches the LHS of some rule
 - Replace with the RHS of the rule

Unrestricted Grammars

- To **non-deterministically** generate a string according to a given **unrestricted grammar**:
 - Start with the initial symbol
 - While the string contains at least one non-terminal:
 - Find a substring that matches the LHS of some rule
 - Replace that substring with the RHS of the rule

Unrestricted Grammars

- Let UG be the set of languages that can be described by an Unrestricted Grammar:
- $UG = \{L : \exists \text{ Unrestricted Grammar } G \text{ such that } L[G] = L\}$
- Claim: $UG = RE$
- To Prove:
 - Show $UG \subseteq RE$
 - Show $RE \subseteq UG$

$UG \subseteq RE$

- Given any Unrestricted Grammar G , we create a Turing Machine M that accepts $L[G]$.
- M will be non-deterministic, simulating derivations of G .

$\mathcal{RE} \subseteq \mathcal{UG}$

- Given any language $L \in \mathcal{RE}$, let M be a **deterministic** Turing Machine that accepts it. We can create an Unrestricted Grammar G such that $L[G] = L$
 - Grammar: Generates a string
 - Turing Machine: Works from string to accept state
- Two formalisms work in different directions
- Simulating Turing Machine with a Grammar can be difficult.

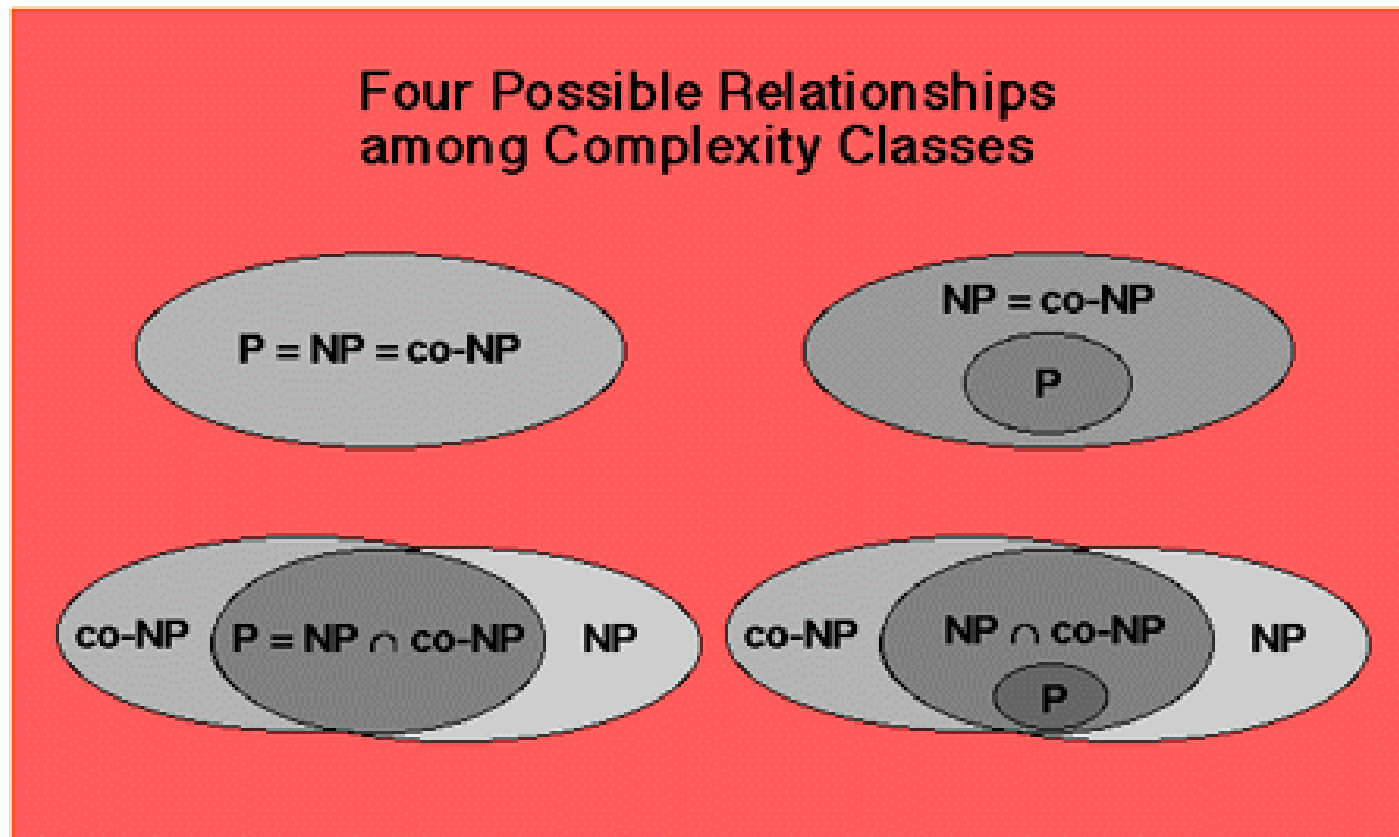
$\mathcal{RE} \subseteq \mathcal{UG}$

- Simulating Turing Machine with a Grammar can be difficult.
- Requires **working backwards**.
- Derivations works from short, accepting configuration, to initial configuration of M , and finally to the **bare string**, $w \in \Sigma^*$.

Ladies and Gentlemen, Boys and Girls

We are about to begin the third part of the course:

Introduction to Computational Complexity



Time Complexity

Consider

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

Clearly this language is decidable.

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

Time Complexity

M_1 : On input string w ,

1. Scan across tape and **reject** if 0 is found to the right of a 1.
2. While both 0s and 1s appear on tape, repeat the following
 - scan across tape, crossing of a single 0 and a single 1 in each pass.
3. If no 0s and 1s remain, **accept**, otherwise **reject**.

Analysis (1)

We consider the three stages separately.

1. Scan across tape and **reject** if **0** is found to the right of a **1**. If not, return to starting point.

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

Analysis (2)

2. While both 0s and 1s appear on tape, repeat the following

- scan across tape, crossing off a single 0 and a single 1 in each pass.
- Each scan requires $O(n)$ steps.
- Since each scan crosses off two symbols, the number of scans is at most $n/2$.
- Total number of steps is $(n/2) \cdot O(n) = O(n^2)$.

Analysis (3)

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.

Final Analysis

Total cost for stages

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

which is $O(n^2)$

Deterministic Time

Let M be a deterministic TM, and let

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

We say that M runs in time $t(n)$ if

- For **every** input x of length n ,
- the number of steps that M uses,
- is **at most** $t(n)$.

Time Classes Definition

Let

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

be a function.

Definition:

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

Do It Faster, Please

We have seen that

- $A = \{0^n 1^n \mid n \geq 0\}$,
- $A \in \text{DTIME}(n^2)$.

Can we do better, *i.e.* faster?

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) in each pass.
3. If no 0s or 1s remain, **accept**, otherwise **reject**.

Analysis

First, we verify that M_2 indeed halts.

- on each scan in step 2.2,
 - The total number of 0s is cut in half,
 - and if there was a remainder, it is discarded.
 - Same for 1s.
- Example: start with 13 0s and 13 1s,
 - first pass: 6 0s and 6 1s are left
 - second pass: 3 0s and 3 1s are left
 - third pass: one 0s and one 1s are left
 - then no 0s and 1s are left.

Analysis

We now verify 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)
- The result, written right to left, is 1101, which is the binary representation of 13.
- Each pass checks equality of one more bit.
- Inequality in any specific bit will be detected (total number of 0s and 1s will be odd).

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**.

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**.

- One pass in each stage (1,2.1,2.2,3) takes $O(n)$ time.
- stage 1 and 3: each executed once
- 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: $O(n) + O(n \log n) = O(n \log n)$.

Further Improvements, Anybody?

Question: Can the running time be made $o(n \log n)$?

Answer: Not on a **single tape** TM (proof omitted).

Question: But why do we have to stick with
single tape TMs?

Answer: We don't!

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*.

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 $M_3: O(n)$.

Important difference between complexity and computability:

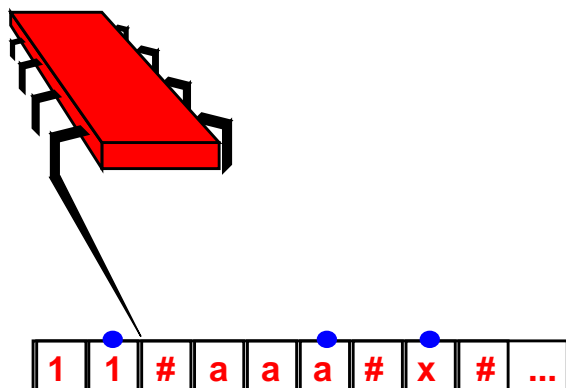
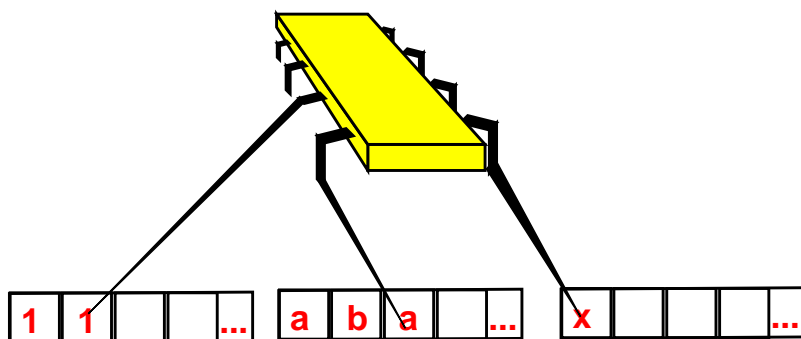
- Computability: all reasonable models **equivalent** (Church-Turing)
- Complexity: choice of model **does affect run-time**.

Q: By **how much** does model affect complexity?

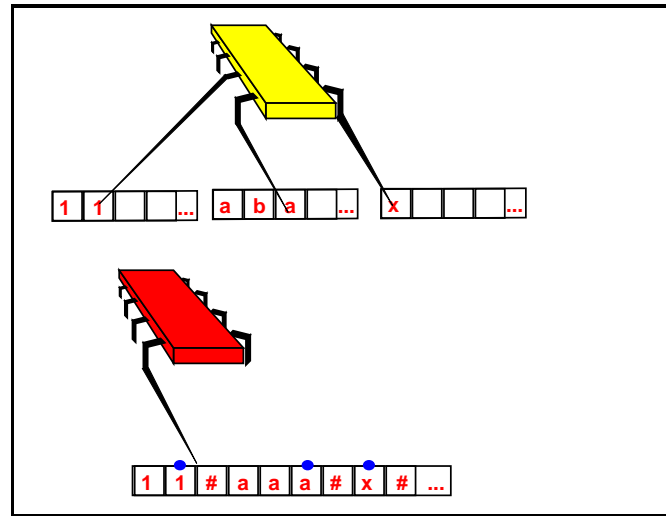
Models and Complexity

Let $t(n)$ be a function where $t(n) \geq n$, and let $L \subseteq \Sigma^*$ be a language.

Claim: If a $t(n)$ -time multitape TM decides L , then \exists an $O(t^2(n))$ -time single tape TM that decided L .



Reminder: Simulating MultiTape TMs



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

- puts on its tape $\# \overset{\bullet}{w_1} w_2 \cdots w_n \# \sqcup \# \sqcup \# \cdots \#$
- 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 of Simulation

For each step of M , S performs

- two scans
- up to k rightward shifts

On input of length n , M makes $O(t(n))$ many steps, so active portion of each tape is $O(t(n))$ long.

Total number of steps S makes:

- $O(t(n))$ steps to simulate **one step** of M .
- Total simulation $O(t(n)) \times O(t(n)) = O(t^2(n))$.
- Initial tape arrangement $O(n)$.
- Grand total: $O(n) + O(t^2(n)) = O(t^2(n))$ steps,
- under the reasonable assumption (**why?**) that $t(n) > n$.

Time Classes Definition, Again

Let

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

be a function.

Definition:

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

Relations among Time Classes

Let $t_1, t_2 : \mathcal{N} \longrightarrow \mathcal{N}$ be two functions.

- Claim: If $t_1(n) = O(t_2(n))$ then

$$\text{DTIME}(t_1(n)) \subseteq \text{DTIME}(t_2(n)) .$$

- Stated informally, more time does **not hurt**.
- But does it actually **help**?
- Claim: If $t_1(n) = O(t_2(n)/\log(n))$ then

$$\text{DTIME}(t_1(n)) \subsetneq \text{DTIME}(t_2(n)) .$$

- Informally, **sufficiently more** time **does help**.
- Proofs – sophisticated diagonalizations (omitted).

Non-Deterministic Time

Let N be a non-deterministic TM, and let

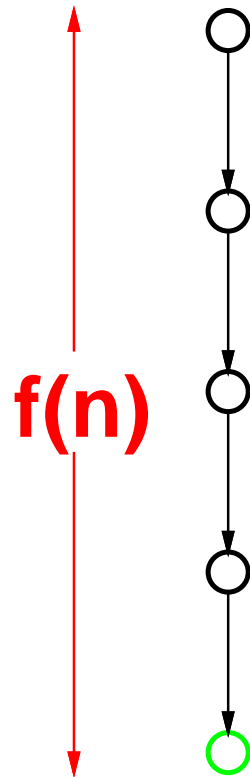
$$f : \mathcal{N} \longrightarrow \mathcal{N}$$

We say that N runs in time $f(n)$ if

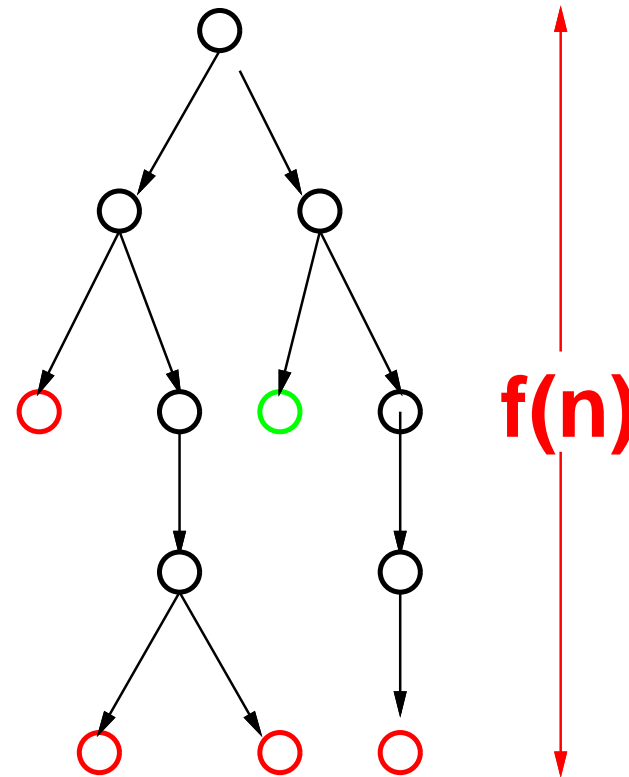
- For **every** input x of length n ,
- the **maximum** number of steps that N uses,
- on **any branch** of its computation tree on x ,
- is **at most** $f(n)$.

Deterministic vs. Non-Deterministic

deterministic



nondeterministic



Notice that **non-accepting** branches must **reject** within $f(n)$ many steps.

Models and Complexity

Claim: Suppose N is a **nondeterministic** TM that runs in time $t(n)$ and decides the language L .

Then there is an $2^{O(t(n))}$ -time **deterministic** TM, D , that decided L .

Note contrast with multi-tape result.

Simulation

Let N be a non-deterministic TM running in $t(n)$ time. Want to describe the deterministic TM, D , simulating N .

Basic idea of simulation:

- D tries all possible branches.
- If D finds any accepting state, it accepts.
- If all branches reject, D rejects.
- Notice N has **no looping branches**, so exactly one of two possibilities must occur.

Simulation Details

N 's computation is a tree:

- root is starting configuration,
- each node has bounded fanout $\leq b$ (why?),
- each branch has length $\leq t(n)$,
- total number of leaves at most $b^{t(n)}$,
- total number of nodes in tree $O(b^{t(n)})$,
- time to arrive from root to any node is $O(t(n))$.
- \implies Time to visit all nodes is

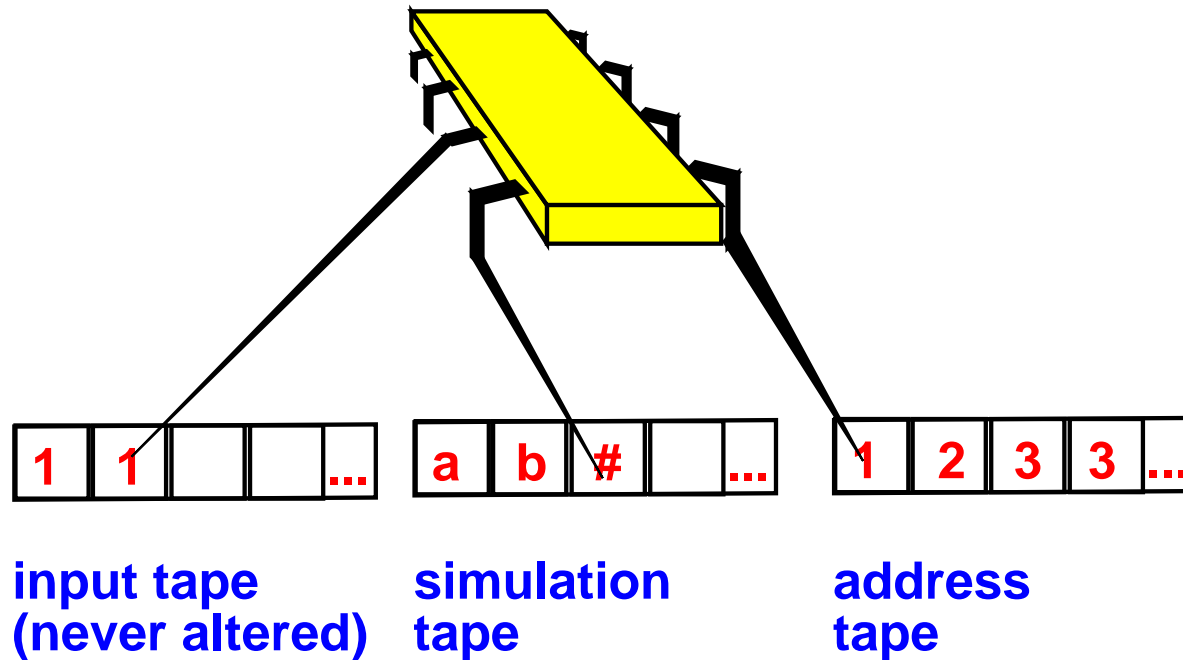
$$O\left(t(n) \times b^{t(n)}\right) = O\left(2^{O(t(n))}\right).$$

Remark

Breadth-first search used in simulation

- Inefficiently traverses from root to visit each node.
- Can be improved upon by using depth-first search (why is it OK now?) or other tree search strategies.
- Still, doing this may save constants, but nothing substantial (why?)

Remark



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

Important Distinction

- At most **polynomial** gap in time to perform tasks between different deterministic models (single- vs. multi-tape TMs, TM vs. **RAM**, etc.)
- compared to
- **Apparently exponential** gap in time to perform tasks between **deterministic** and **non-deterministic** models.

The Good, the Bad, and the Ugly

Complexity differences:
Polynomial is small; Exponential is 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 is Good, Exponential is Bad

Claim: All “reasonable” models of computation are polynomially equivalent.

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

Question: Is a given problem solvable in

- Linear time? **model-specific.**
- Polynomial time? **model-independent.**
- We are interested in computation, not in models *per se!*

The Class P

Definition: P is the set of languages decidable in polynomial time on deterministic TMs.

$$P = \bigcup_{c \geq 0} \text{DTIME}(n^c)$$

- The class P is important because:
- Invariant for all TMs with any number of tapes.
- Invariant for all models of computation polynomially equivalent to TMs.
- Roughly corresponds to **realistically solvable** (tractable) problems.

The Class P

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

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 a more efficient one.

Examples: Problems in P

- **Arithmetic:** Addition, subtraction, multiplication, division with remainder.
- **Integer Algorithms:** Greatest common divisor (gcd).
- **Operations research:** Maximum flow, linear programming,
- **Algebra:** Matrix multiplication, computing determinants, matrix inversion, solving systems of linear equations, factoring polynomials.
- **Graph algorithms:** DFS and BFS in graphs, minimum spanning trees, finding Eulerian path.

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 not realistic (exponentially longer)

For 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 ?

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

Complexity of PATH

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?

Complexity of 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
 - Oh, oh. Does not sound like P to me ...

Complexity of PATH

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 node a to unmarked node b ,
 - then mark node b .
3. If t marked, **accept**, otherwise **reject**.

Question: What is the complexity of this algorithm?

Complexity of 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 at least one 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 and marks nodes in graph, also polynomial.

Total time complexity is polynomial.



Relative Primality

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:

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

Relative Primality

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

First Idea: Search through all possible divisors of x, y and test divisibility.

If x, y in unary:

- size of $\langle x \rangle$ is x
- testing all potential divisors of x, y is polynomial

If x, y in binary:

- size of $\langle x \rangle$ is $\log x$
- testing all potential divisors of x, y is exponential

Important Notation: Such algorithm is called pseudo-polynomial.

GCD

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

Euclid's greatest common divisor algorithm, E :

On input $\langle x, y \rangle$

1. Repeat until $y = 0$
 - $x \leftarrow x \bmod y$
 - exchange x and y
2. output x

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

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

Euclid's Algorithm

Enough to check that Euclid's Algorithm is polynomial.

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

total running time is polynomial.

Correctness holds (but should be verified).

Consequently, **RELPRIME** $\in P$.



NTime Classes Definition

Let

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

be a function.

Definition:

$$\text{NTIME}(f(n)) = \{L \mid L \text{ is a language, decided by an } O(f(n))\text{-time NTM}\}$$

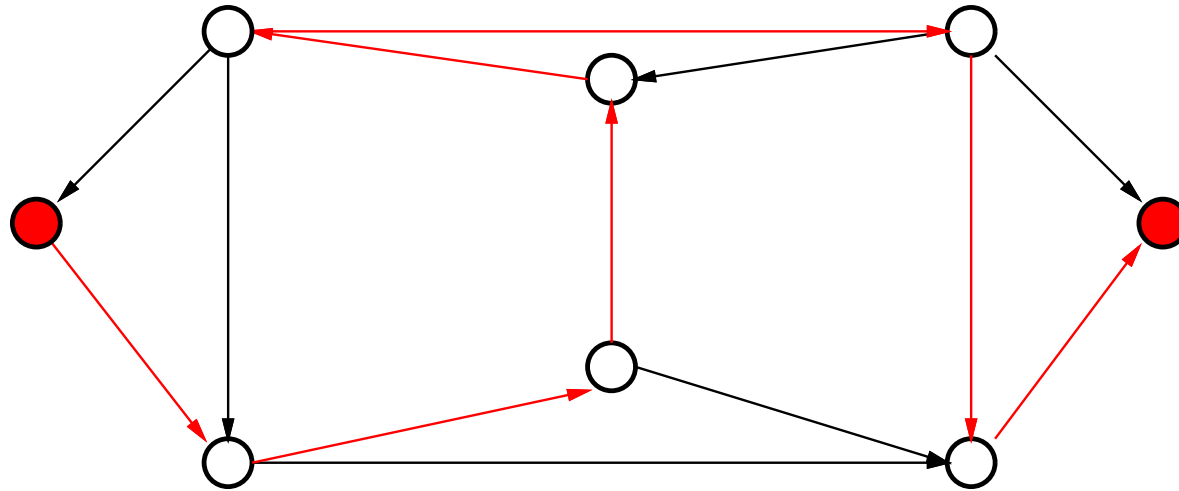
The Class NP

Definition: NP is the set of languages decidable in polynomial time on **non**-deterministic TMs.

$$NP = \bigcup_{c \geq 0} \text{NTIME}(n^c)$$

- The class NP is important because:
- Invariant for all TMs with any number of tapes.
- NP is insensitive to choice of reasonable non-deterministic computational model.
- Roughly corresponds to problems whose **positive solutions** cannot be efficiently **generated** (\Rightarrow intractable), but can be efficiently **checked**.

Hamiltonian Path



A **Hamiltonian path** in a directed \mathcal{G} visits each **node** exactly once.

Hamiltonian Path

HAMPATH = $\{\langle G, s, t \rangle \mid G \text{ has Hamiltonian path from } s \text{ to } t\}$

Question: How hard is it to decide this language?

Hamiltonian Path

HAMPATH = $\{\langle G, s, t \rangle \mid G \text{ has Hamiltonian path from } s \text{ to } t\}$

Easy to obtain **exponential time** algorithm:

- **generate** each potential path
- **check** whether it is Hamiltonian

The Class \mathcal{NP}

Here is an **NTM** that decides **HAMPATH** in poly time.

On input $\langle G, s, t \rangle$,

1. **Guess** and write down a list of numbers p_1, \dots, p_m , where m is number of nodes in G , and $1 \leq p_i \leq m$.
2. Check for repetitions in list. If any found, *reject*.
3. Check whether $p_1 = s$ and $p_m = t$. If either does not hold, *reject*.
4. For $i, 1 \leq i \leq m - 1$, check whether (p_i, p_{i+1}) is an **edge** in G . If any is not, *reject*. Otherwise *accept*.

NP

On input $\langle G, s, t \rangle$,

1. **Guess** and write down a list of numbers $p_1, \dots, p_m \dots$
2. Check for repetitions ...
3. Check whether $p_1 = s$ and $p_m = t \dots$
4. Check whether (p_i, p_{i+1}) is an **edge** in $G \dots$

- Stage 1 polynomial time
- Stages 2 and 3 simple checks.
- Stage 4 simple poly-time too.

Hamiltonian Path

This problem has one very interesting feature:
polynomial verifiability.

- we don't know a fast way to **find** a Hamiltonian path
- but we can **check** whether a **given path** is Hamiltonian in polynomial time.

In other words,

- **verifying** correctness of a path is much **easier**
- than **determining** whether one exists

Composite Numbers

A natural number is **composite** if it is the product of two integers greater than one.

$$\text{COMPOSITES} = \{x \mid x = pq \text{ for integers } p, q > 1\}$$

- we don't know a polynomial-time algorithm for **deciding** this problem^{* a}
- But we can easily **verify** that a number is composite (**how?**)

^{a*} Actually, in summer 2002, two Indian undergrads and their advisor found how to do this. However, let us pretend we're still in 1/1/2002...

Verifiability

Not all problems are polynomially verifiable.

There is no known way to verify HAMPATH in polynomial time.

In fact, we will see many examples where L is polynomially verifiable, but its complement, \overline{L} , is not known to be polynomially verifiable.

Verifiability

A **verifier** for a language \mathcal{A} is an algorithm \mathcal{V} where

$$\mathcal{A} = \{w \mid \mathcal{V} \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$$

- The verifier uses the additional information c to verify $w \in \mathcal{A}$.
- We measure verifier run time by length of w .
- The string c is called a **certificate** (or **proof**) for w if \mathcal{V} accepts $\langle w, c \rangle$.
- A **polynomial verifier** runs in polynomial time in $|w|$ (so $|c| \leq |w|^{O(1)}$).
- A language \mathcal{A} is **polynomially verifiable** if it has a polynomial verifier.

Examples

For **HAMPATH**, a certificate for

$$\langle G, s, t \rangle \in \text{HAMPATH}$$

is simply the Hamiltonian path from s to t .

Can **verify** in time polynomial in $|\langle G \rangle|$ whether given path is Hamiltonian.

Examples

For **COMPOSITES**, a certificate for

$$x \in \text{COMPOSITES}$$

is simply one of its divisors.

Can **verify** in time polynomial in $|x|$ if given divisor indeed divides x .