Computational Models

- Deterministic Time Classes
- NonDeterministic Time Classes
- Relationship between Deterministic and Nondeterministic Time
- The classes $P$ and $NP$
- Examples of Problems in $P$ and in $NP$
- Verifiability
- The class $coNP$

Sipser, Chapter 7
Definition

Let $M$ be a deterministic TM that halts on all inputs. The running time of $M$ is a function

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

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 $f(n)$-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: In $6n^3 + 2n^2 + 20n + 45$, the term $n^3$ is the most important one.
Asymptotic Notation

Consider functions

\[ f, g : \mathbb{N} \to \mathbb{R}^+ \]

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 all \( n \geq n_0 \).
Examples

- $5n^3 + 2n = O(n^3)$
- $617n^4 + 234n^3 + 20n^2 + 5n = O(n^4)$
- $\sin(x) = O(1)$
- $2 = O(1)$. 
Consider

\[ A = \{0^m1^m \mid m \geq 0\} \]

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.
We consider the three stages separately. Let $n$ denote the input length.

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.
- Re-positioning 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 of 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 $(n/2) \cdot O(n) = O(n^2)$. 
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 (Reminder)

Let $M$ be a deterministic TM, and let

$$t : \mathbb{N} \rightarrow \mathbb{N}$$

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

\[ t : \mathbb{N} \rightarrow \mathbb{N} \]

be a function.

**Definition:**

\[ \text{DTIME}(t(n)) = \{ L | L \text{ is a language, decided by an } O(t(n))-\text{time DTM} \} \]

Note that \( t(n) \) run time is also required for strings that are not in \( L \).
Can we do it faster?

Using a naive algorithm, we have seen that
\[ A = \{0^m1^m \mid m \geq 0\} \in \text{DTIME}(n^2). \]

Can we do it faster?

Using a more sophisticated, “binary representation algorithm”, we can show: \( A \in \text{DTIME}(n \log n) \).

In fact this is the best one can do on a single tape TM.
\( M_2 \) on input string \( w \):

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

2. Repeat the following while both 0 and 1 appear on tape:
   (a) scan across tape, checking whether total number of 0s plus 1s is even or add. If odd, reject.
   (b) 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 2a:
  - 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 0 and one 1 are left
  - then no 0s and 1s are left.
Further Improvements?

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

**Answer:** Not on a single tape TM.

**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 \( \{0^n1^n\} \):

- 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?
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 there is an $O(t^2(n))$-time single tape TM that decides $L$. 

![Diagram](image_url)
On input $w = w_1 \cdots w_n$, single tape $S$:

- puts on its tape $\# w_1 w_2 \cdots w_n \# \# \# \# \# \cdots \#$
- 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 $t(n) > n$. 

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
Let $N$ be a non-deterministic TM, and let

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

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

Notice that non-accepting branches must reject within $f(n)$ many steps.
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 $O\left(b^{t(n)}\right)$,
- 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?)
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.
Complexity differences: Polynomial is small; Exponential is large

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

- **Integer arithmetic:** Addition, subtraction, multiplication, division with remainder.
- **Modular arithmetic:** Exponentiation (RSA), inverse.
- **Integer Algorithms:** Greatest common divisor (gcd).
- **Operations research:** Maximum network flow, linear programming,
- **Algebra:** Matrix multiplication, computing determinants, matrix inversion, solving systems of linear equations, factoring polynomials.
- **Graph algorithms:** DFS and DFS 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 is good
- unary is 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 \( t \)?

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

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

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

**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

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 \( y = 0 \)
   - \( x \leftarrow x \mod y \)
   - 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, $\text{RELPRIME} \in P$. 

♣
Decidability of Context-Free Languages

- **Theorem:** Every context-free language is a member of \( P \).

- We already know that there is an algorithm for each CFL that decides it. Is there one that runs in polynomial time?

- **First attempt:** Let \( L \) be a CFL generated by a CFG in Chomsky Normal Form. Then any \( w \in L \) of length has a derivation of \( 2n - 1 \) steps. Try all possible derivations of this length, and if there is one of \( w \) - accept. Else - reject.

- But the number of derivations of \( k \) steps may be exponential in \( k \).
Dynamic Programming Approach

- **Technique of dynamic programming**: accumulation of information about smaller subproblems to solve larger problems.

- Record the solution to any subproblem so that we don’t need to solve it later in a table.

- The subproblem: determine whether each variable of $G$ generates each substring of $w$.

- The $(i, j)$-th entry of the table contains all variables that generate the substring $w_i \ldots w_j$.

- For $j < i$, the entries are not used.
The algorithm fills the entries of the table for each substring of $w$: first of length 1, then of length 2,...

Suppose that the algorithm has determined which variables generate of substrings up to length $k$. To determine whether variable $A$ generates a substring of length $k + 1$, we split it to two non-empty parts in $k$ possible ways.

For each split, we examine each rule $A \rightarrow BC$ to see whether $B$ generates the first part, and $C$ generates the second part.
The Algorithm

Let $G$ be a CFG in Chomsky Normal Form. On input $w = w_1 \ldots w_n$:

1. If $w = \epsilon$ and $S \rightarrow \epsilon$, accept.

2. For $i = 1$ to $n$ and each variable $A$, test whether $A \rightarrow b$ is a rule, where $b = w_i$. If so, place $A$ in $\text{table}(i,i)$.

3. For $l = 2$ to $n$:

4. For $i = 1$ to $n - l + 1$, let $j = i + l - 1$ ($j$ is the end position of the substring).

5. For $k = i$ to $j - 1$:

6. for each rule $A \rightarrow BC$: if $\text{table}(i,k)$ contains $B$ and $\text{table}(k+1,j)$ contains $C$, put $A$ in $\text{table}(i,j)$.

7. If $S$ is in $\text{table}(1,n)$, accept. Else reject.
Analysis

- Stage 2 runs at most $Vn$ times, where $V$ is the number of variables of $G$ and is a constant (independent of $|w| = n \Rightarrow O(n)$).

- The inner loop runs in $O(n^3)$ (the number of variables is a constant independent of $|w|$).

- Overall: $O(n^3)$
Let $f : \mathbb{N} \rightarrow \mathbb{N}$ 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.
- Rougly corresponds to problems whose **positive solutions** cannot be efficiently generated ($\Rightarrow$ intractable), but can be efficiently verified.
A Hamiltonian path in a directed graph visits each node exactly once.
Hamiltonian Path

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

**Question:** How hard is it to decide this language?
Hamiltonian Path

\[ \text{HAMPATH} = \{ \langle G, s, t \rangle | G \text{ has Hamiltonian path from } s \text{ to } t \} \]

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 $p_1, \ldots, p_m$, 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.
On input \( \langle G, s, t \rangle \),

1. Non-deterministically guess and write down a list of numbers \( p_1, \ldots, p_m \).
2. Check that there are no repetitions.
3. Check whether \( p_1 = s \) and \( p_m = t \).
4. Check whether \((p_i, p_{i+1})\) is an edge in \( G \).
5. If 1 through 4 satisfied, accept, else reject.

- 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*
- But we can easily verify that a number is composite.
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.
A verifier for a language $\mathcal{A}$ is an algorithm $V$ where

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

- 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 $V$ 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 $x$. 
Theorem: A language is in NP if and only if it has a polynomial time verifier.

Proof – Intuition:

- NTM simulates verifier by guessing the certificate.
- Verifier simulates NTM by using accepting branch as certificate.