Time Complexity (1)

CSCI 2670

Original Slides were written by Dr. Frederick W Maier

Spring 2014
So far we've dealt with determining whether or not a problem is decidable.

But even if it is, it might be “too difficult” in practice to decide.

For a given string $w$ and language $L$, it might require too much time or too much memory to determine whether or not $w \in L$.

The time required to solve a problem is called its **time complexity**.

The memory required to solve a problem is called its **space complexity**.

**Computational complexity theory** is the study of the time and space complexity of problems.

Chapter 7 deals with time complexity.
Suppose we have a TM to decide $A = \{0^k1^k | k \geq 0\}$.

This language is context free and so decidable.

Informally, the time complexity is the number of steps required by the TM, as a function of the input size.

We want to know the number of steps needed to determine whether $w \in A$.

We usually express time complexity as a function of the length of the input string $w$.

Note that if different input strings $u$ and $v$ both have length $n$, it might take more time to process $u$ than $v$.

In **worst case analysis**, we are interested in the maximum number of steps required for an input string of length $n$.

In **average case analysis**, we are interested in the average number of steps required for an input string of length $n$. 


If $M$ is a deterministic TM that halts on all inputs, then the **time complexity (running time)** of $M$ is the function $f : \mathbb{N} \rightarrow \mathbb{N}$, where $f(n)$ is the maximum number of steps $M$ uses on an input of length $n$.

We say that $M$ runs in time $f(n)$ and $M$ is an $f(n)$ Turing machine.

It is often more convenient to use estimates of $f(n)$ rather than $f(n)$ itself to describe the running time of a TM.

In **asymptotic analysis**, we estimate the running time of the algorithm when it is run on large inputs.

Not all terms of the function contribute very much to the running time and so can be ignored.

The most common estimates are **big-O** and **small-O (little-O)** estimates.
**Definition**

If $f$ and $g$ are functions such that $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$, then $f(n)$ is $O(g(n))$ iff there exist positive integers $c$ and $n_0$ such that

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

$g(n)$ is an **asymptotic upper bound** ($cg(n)$ is an upper bound of $f(n)$).

“$f(n)$ is $O(g(n))$” means that if we ignore constant factors, $f(n) \leq g(n)$.

**Example**

- $5n^3 + 2n^2 + 22n + 6$ is $O(n^3)$: Let $c = 6$ and $n_0 = 10$.
- $n$ is $O(n^2)$. Let $c = 1$ and $n_0 = 1$.
- $n^2$ is not $O(n)$: Not matter what $c$ and $n_0$ are chosen, $n^2 \not\leq cn$ for some $n \geq n_0$. 
The base of a logarithm doesn’t matter when using Big-O notation.

Note that for any bases $a$ and $b$, $\log_b(n) = \frac{\log_a(n)}{\log_a(b)}$.

So, if $f(n) \leq c\log_b(n)$, then $f(n) \leq c\frac{\log_a(n)}{\log_a(b)}$.

Letting $c_1 = \lceil \frac{c}{\log_a(b)} \rceil$, it follows that $f(n) \leq c_1 \log_a(n)$.

So, if $f(n)$ is $O(\log_b(n))$, $f(n)$ is $O(\log_a(n))$.

We don’t even bother with the base: $f(n)$ is $O(\log(n))$.

Example

If $f(n) = 3n\log_2(n) + 5n\log_2(\log_2(n)) + 2$, then $f(n)$ is $O(n\log(n))$. 
If \( f_1(n) \) is \( O(g_1(n)) \) and \( f_2(n) \) is \( O(g_2(n)) \), then
- \( f_1(n) + f_2(n) \) is \( O(g_1(n)) + O(g_2(n)) \).
- \( f_1(n) + f_2(n) \) is \( \max(O(g_1(n)), O(g_2(n))) \).

If \( f(n) \) appears in an exponent, we can use the Big-O estimate there:
- \( 2^{3n^3 + 2n^2 + n + 6} \) is \( 2^{O(n^3)} \).

Frequently we derive bounds of the form \( n^c \) for \( c > 0 \). Such bounds are called \textit{polynomial bounds}.

Bounds of the form \( 2^{(n^\delta)} \) are called \textit{exponential bounds} when \( \delta \) is a real number greater than 0.
In a way, Big-O notation says that $f(n)$ is less than or equal to $g(n)$. Small-O notation says that $f(n)$ is less than $g(n)$. 

**Definition**

If $f$ and $g$ are functions such that $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$, then $f(n)$ is $o(g(n))$ iff

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

Alternatively, $f(n)$ is $o(g(n))$ iff for all real constants $c > 0$, there is an $n_0$ such that $f(n) < cg(n)$ for all $n \geq n_0$.

**Example**

- $\sqrt{n}$ is $o(n)$.
- $n$ is $o(n \log(\log(n)))$.
- $n \log(n)$ is $o(n^2)$.
- $n^2$ is $o(n^3)$. 
Consider TM $M_1$ which decides $A = \{0^k1^k | k \geq 0\}$. It works in 4 phases.

On input $w$:
1. Scan the tape, rejecting if a 0 is found to the right of a 1.
2. While both 0s and 1s are still on the tape:
3. Scan the tape, marking off a single 0 and 1.
4. Reject if a 0 remains but all 1s are marked, or vice versa. If not, accept.

What is the running time of $M_1$ as a function of $n$?
Consider TM $M_1$ which decides $A = \{0^k1^k | k \geq 0\}$. It works in 4 phases.

On input $w$:
1. Scan the tape, rejecting if a 0 is found to the right of a 1.
2. While both 0s and 1s are still on the tape:
3. Scan the tape, marking off a single 0 and 1.
4. Reject if a 0 remains but all 1s are marked, or vice versa. If not, accept.

Phase 1 scans once through the tape, taking $O(n)$ steps, where $|w| = n$. The tapehead returns left—another $n$ steps. Phase 1 takes $O(n)$ steps.

In Phase 2-3, the tape is scanned to check that both 1s and 0s appear; another scan marks off a single 0 and 1.

In each cycle, 2 symbols are marked, and so the total number of cycles is $O(n/2)$. Phases 2 and 3 together take $O(n^2)$ steps.

In Phase 4, we check to see that all 0s and 1s are marked off. This takes only a single scan of the tape: $O(n)$.

And so the running time of $M_1$ is $O(n) + O(n^2) + O(n) = O(n^2)$. 
Complexity Classes: \( \text{TIME}(t(n)) \)

- Observe that \( M_1 \) ran in time \( O(n^2) \), and it decides \( A \).
- We can classify languages by the algorithms that decide them.

**Definition**

Let \( t : \mathbb{N} \rightarrow \mathbb{R}^+ \) be a function. The **time complexity class** \( \text{TIME}(t(n)) \) is the set of all languages that can be decided in time \( O(t(n)) \).

- Observe, e.g., if \( L \in \text{TIME}(n^2) \), then \( L \in \text{TIME}(n^3) \).
- If a decider \( M \) for \( L \) runs in time \( O(t(n)) \), then \( L \in \text{TIME}(t(n)) \).
  - So, \( A = \{0^k1^k | k \geq 0\} \) is in time \( \text{TIME}(n^2) \).
- Failing to find a \( O(t(n)) \)-time decider **doesn’t imply** \( L \notin \text{TIME}(t(n)) \).
Consider TM $M_2$ which decides $A = \{0^k1^k \mid k \geq 0\}$. It works in 5 phases.

On input $w$:

1. Scan the tape, rejecting if a 0 is found to the right of a 1.
2. While some 0s and some 1s are on the tape:
   3. Scan the tape. Reject if the number of unmarked symbols is odd.
   4. Scan the tape, crossing off every other 0 and every other 1.
3. Scan the tape. If all symbols are marked, accept. Otherwise reject.

What’s the running time of $M_2$?
Consider TM $M_2$ which decides $A = \{0^k1^k | k \geq 0\}$. It works in 5 phases.

**On input $w$:**

1. Scan the tape, rejecting if a 0 is found to the right of a 1.
2. While some 0s and some 1s are on the tape:
   3. Scan the tape. Reject if the number of unmarked symbols is odd.
   4. Scan the tape, crossing off every other 0 and every other 1.
5. Scan the tape. If all symbols are marked, accept. Otherwise reject.

Phase 1 again takes $O(n)$ steps, as does phase 5.
To check that some 0s and 1s appear (phase 2) takes $O(n)$ steps.
Each execution of phase 3 and 4 takes $O(n)$ steps.
Each execution of phase 4 cuts the number of 0s and 1s by half.
Phase 3, 4 run at most $1 + \log(n)$ times; phases 2-4 take time $O(n\log(n))$.
Total running time of $M_2$: $O(n) + O(n\log(n)) = O(n\log(n))$.
As such, $A \in \text{TIME}(n\log(n))$. 
Analyzing Algorithms

- TMs $M_1$ and $M_2$ were single tape deterministic machines.
- $M_3$ is a 2-tape machine that decides $A$.

On input $w$:
1. Scan tape 1, rejecting if a 0 is found to the right of a 1.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, reject.
4. If all the 0s have now been crossed off, accept. If any 0s remain, reject.”

- What’s the running time of $M_3$?
Analyzing Algorithms

- TMs $M_1$ and $M_2$ were single tape deterministic machines.
- $M_3$ is a 2-tape machine that decides $A$.

On input $w$:
1. Scan tape 1, rejecting if a 0 is found to the right of a 1.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, reject.
4. If all the 0s have now been crossed off, accept. If any 0s remain, reject.

- $M_3$ runs in time $O(n)$.
- Note that this running time is the best possible because $n$ steps are necessary just to read the input.
Analyzing Algorithms

- $M_2$ decides $A$ in time $O(n \log(n))$.
- It turns out that no single tape TMs can decide $A$ in $o(n \log(n))$.
- In fact, a language decidable by a 1-tape TM in $o(n \log(n))$ is regular.
- Yet a multitape TM can decide $A$ (a nonregular language) in $O(n)$.
- It is important to realize the following:

  The complexity class of a language depends on the computational model.

- Contrast this with the Church-Turing Thesis, which asserts that the computational model doesn’t affect decidability.
- One question to ask is: How do the complexity classes of one computational model relate to those of another model?
Complexity Relationships Among Models: k-tape vs. 1-tape

**Theorem**

*Every $t(n)$ time multitape TM $M$, where $t(n) \geq n$, has an equivalent $O(t^2(n))$ single tape TM $S$.***

We already have a way to convert a k-tape TM $M$ into a single tape machine $S$. We simply need to analyze the behavior of $S$. 

![Diagram of TM conversion](image)
To simulate the behavior of the multi-tape machine, the single tape machine makes multiple passes across its tape, updating the virtual tapes appropriately.

If a virtual tape head moves onto an unread blank, the contents of the single tape must be shifted appropriately to make room.
Complexity Relationships Among Models: k-tape vs. 1-tape

**Theorem**

*Every* $t(n)$ *time multitape TM* $M$, where $t(n) \geq n$, *has an equivalent* $O(t^2(n))$ *single tape TM* $S$.

$M$: a k-tape TM; $S$: a single tape TM.

- For each step of $M$, $S$ makes two passes over its tape: one to collect information about the current configuration of $M$, and one pass to update $M$’s configuration.
- Note that each of $M$’s tapes can use **at most** $t(n)$ tape cells.
- $S$ must simulate each of the $t(n)$ steps of $M$.
- To update a given portion of $S$ tape (representing one of the $k$ tapes), a shift of its contents might be needed. This takes no more than $t(n)$ steps. So to update all $k$ tape portions takes $O(kt(n))$ or $O(t(n))$ steps.
- As such, the total running time of $S$ is $t(n)O(t(n))$ or $O(t^2(n))$. 
In a nondeterministic TM $N$, the computations form a tree.

- Each node represents a configuration.
- Each edge represents a possible transition from one configuration to another.
- The TM accepts $w$ if any branch ends in an accepting configuration.
- It rejects if all branches end in a rejecting configuration.
- In a nondeterministic decider, all computation branches halt.
Complexity Relationships Among Models: NTMs vs DTMs

**Definition**

Let \( N \) be a nondeterministic TM decider. The running time \( f(n) \) of \( N \) is a function \( f : \mathbb{N} \rightarrow \mathbb{N} \) where \( f(n) \) is the maximum number of steps taken by any of \( N \)'s computation branches on any input of length \( n \).

**Theorem**

Every \( t(n) \) time nondeterministic single-tape TM, where \( t(n) \geq n \), has an equivalent \( 2^{O(t(n))} \)-time deterministic single-tape TM.

- Observe that we have a way to simulate a nondeterministic TM (NTM) \( N \) into a deterministic 3-tape machine \( D \).
An NTM $N$ can be simulated with deterministic TM (DTM) $D$.

- $D$ uses 3 tapes:
  - **Tape 1**—Records the input string. It’s reused many times.
  - **Tape 2**—used as $N$’s tape.
  - **Tape 3**: holds a string $d_1d_2d_3\ldots d_n$, where each $d_i$ indicates a choice to make at step $i$. Each $d_i$ is taken from $1,2,\ldots,b$, where $b$ is the maximum number of children of any node. The string $d_1d_2d_3\ldots d_n$ indicates a computation branch of $N$. 

1. The DTM $D$ begins with tape 2,3 empty; tape 1 holds the input.
2. Wipe tape 2, and copy the input string from tape 1 to tape 2.
3. Simulate the NTM $N$ on tape 2.
   3.1 At each step $i$, determine the value $v$ of cell $d_i$ on tape 3.
   3.2 If $v$ is a valid transition choice for $N$, then update tape 2.
   3.3 If not, abort the branch: GOTO step 4.
   3.4 Also abort if the transition represents reject.
4. Update the value on tape 3 by choosing the lexicographically next string. GOTO step 2.
When simulating the NTM $N$, $D$ uses a **breadth first search** (it visits all nodes of depth $d$ before visiting nodes of depth $d+1$). This is due to how strings for tape 3 are chosen (lexicographically).

- Let $b$ be the maximum number of children of any node.
- Every branch of $N$’s nondeterministic computation tree has a length of at most $t(n)$.
- The time for starting from the root and traveling down to a node is $O(t(n))$. 
The number of leaves in the tree is at most: $b^{t(n)}$.

The total number of nodes in the tree is less than twice the maximum number of leaves.

So the number of nodes in the tree is bounded by the number of leaves: $O(b^{t(n)})$.

As such, the total running time of $D$ is $O(t(n)b^{t(n)})$. 
Definition

Let $N$ be a nondeterministic TM decider. The running time $f(n)$ of $N$ is a function $f : \mathbb{N} \rightarrow \mathbb{N}$ where $f(n)$ is the maximum number of steps taken by any of $N$’s computation branches on any input of length $n$.

Theorem

Every $t(n)$ time nondeterministic single-tape TM, where $t(n) \geq n$, has an equivalent $2^{O(t(n))}$-time deterministic single-tape TM.

- The total running time of $D$ is $O(t(n)b^{t(n)})$.
- $t(n)b^{t(n)} = 2^{\log_2(t(n))b^{t(n)}} = 2^{\log_2(t(n)) + \log_2(b^{t(n)})} = 2^{\log_2(t(n)) + t(n) \log_2(b)}$
- Running time of $D$ is $O(2^{\log_2(t(n)) + t(n) \log_2(b)}) = 2^{O(t(n))}$.
- From $D$ we can construct a single tape DTM $M$ running in squared time.
- Running time of $M$ is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$. 
The Class P

- There is a polynomial difference in the running time of a deterministic k-tape TM and a single tape TM.
- There is an exponential difference in the running time of a nondeterministic TM and a deterministic TM.
- The difference between algorithms that run in polynomial running times and those with exponential running times is considered great.
  - Polynomial time algorithms are considered tractable.
  - Exponential time algorithms are rarely usable for large input sizes.
- All reasonable deterministic computational models are polynomially equivalent. That is, any one of them can simulate another with only a polynomial increase in running time.
- For the moment, we will focus on the class of problems that are decidable in polynomial time.
- This is the class $\mathbf{P}$. 

The Class P

Definition

$\mathbf{P}$ is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine.

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

- The various “reasonable” models of computation that have been developed are all \textit{polynomially equivalent}—each can be simulated by another with only a polynomial increase in running time.
- As such, the emphasis on deterministic single-tape Turing machines is not important.
- We could replace deterministic single-tape Turing machines with some other reasonable model of computation, and $\mathbf{P}$ would be left unaffected.
- That is, $\mathbf{P}$ is \underline{invariant} for all computation models that are polynomial equivalent to deterministic single tape TMs.
- Roughly, $\mathbf{P}$ corresponds to problems that are realistically solvable by computers.
A note about encoding schemes

- How a problem is encoded has an affect on the running time of an algorithm.

- For instance, we might choose to encode a graph of \( n \) nodes as a list of numbers (nodes) and number pairs (edges). We might choose to encode a TM \( M \) as a list of its 7-tuples. They are reasonable.

- But note that unary notation for encoding numbers (as in the number 17 encoded by the unary string \( 11111111111111111 \)) is not reasonable because it is exponentially larger than truly reasonable encodings, such as base \( k \) notation for any \( k \geq 2 \).

- We only consider “reasonable” encoding schemes, those that can be converted to another reasonable encoding scheme in polynomial time.
The language $PATH$ is in $P$:

$PATH = \{ \langle G, s, t \rangle | G \text{ is a directed graph with directed path from } s \text{ to } t \}$.

- A path can be represented as a list of nodes.
- Suppose $G$ has $m$ nodes. Then a path in $G$ has at most length $m$.
- A brute force approach of checking all potential paths in $G$ would take exponential time.
- The below algorithm $M$, however, will take polynomial time.

On input $\langle G, s, t \rangle$ (where $G$ has $m$ nodes):

1. Mark $s$.
2. Repeat Step 3 until no additional nodes are marked:
3. Scan the nodes and edges of $G$. If $a$ is a marked node and $(a, b)$ is an edge and $b$ is unmarked, then mark $b$.
4. If $t$ is marked, accept. If not reject.
On input $\langle G, s, t \rangle$ (where $G$ has $m$ nodes):

1. Mark $s$.
2. Repeat Step 3 until no additional nodes are marked:
   3. Scan the nodes and edges of $G$. If $a$ is a marked node and $(a, b)$ is an edge and $b$ is unmarked, then mark $b$.
   4. If $t$ is marked, accept. If not reject.

- 1 and 4 are executed once. They require (at most) scanning $G$ once.
- 3 is executed at most $m$ times, since a node is marked with each iteration.
- Executing 3 can clearly be done in polynomial time (relative to $m$).
- So the running time of the algorithm is polynomial.
Integers \( x \) and \( y \) are relatively prime if their greatest common divisor is 1.

That is, \( x \) and \( y \) are relatively prime if 1 is the only integer to evenly divide them both.

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

Observe that a brute force approach of checking every integer \( n \) between 2 and, e.g., \( \min(x, y) \) won’t work, because the value of \( n \) will be exponentially larger than any reasonable encoding of \( n \).

Instead, the **Euclidean algorithm** can be used.
Problems in P: \textit{RELPRIME}

- The \textbf{Euclidean algorithm} is encoded in $E$.
- $R$ uses $E$ to decide \textit{RELPRIME}.
- If $E$ runs in polynomial time, then $R$ does.

\begin{itemize}
\item[$E =$] On input $\langle x, y \rangle$ ($x, y \in \mathbb{N}$, represented in binary.):
  \begin{enumerate}
  \item Repeat until $y = 0$:
  \item assign $x \mod y$ to $x$.
  \item swap $x$ and $y$.
  \item return $x$.
  \end{enumerate}
\item[$R =$] On input $\langle x, y \rangle$ ($x, y \in \mathbb{N}$, represented in binary.):
  \begin{enumerate}
  \item Run $E$ on $\langle x, y \rangle$.
  \item If $E$ returns 1, then accept. Otherwise reject.
  \end{enumerate}
\end{itemize}
E = On input \langle x, y \rangle (x, y \in \mathbb{N}, \text{represented in binary.}): 
1. Repeat until \( y = 0 \):
2. assign \( x \mod y \) to \( x \).
3. swap \( x \) and \( y \).
4. return \( x \).

▶ In iteration 1, if \( x < y \), then \( x \) and \( y \) are swapped. So assume \( x > y \).
▶ After stage 2, \( x < y \) (because \( x \mod y < y \)). After stage 3, \( x > y \).
▶ Either \( x/2 \geq y \) or else \( x/2 < y \).
▶ If \( x/2 \geq y \), then since \( x \mod y < y \), it follows that \( x \mod y < y \leq x/2 \).
▶ If \( x/2 < y \), then \( x < 2y \), and so \( x \mod y = x - y \). Note that \( x - y < x/2 \) (since \( x - x/2 < y \) implies \( x - y < x/2 \)). \( \therefore x \mod y < x/2 \).
▶ So, either way, \( x \) drops by half. Since \( x \) and \( y \) are swapped each iteration, the values of each are cut in half every other iteration.
▶ So 2-3 are run \( \min(\log_2(x), \log_2(y)) \).
▶ This is proportional to the encoding size of \( x \) and \( y \), and so the algorithm runs in time \( O(n) \).
Which of the following pairs of numbers are relatively prime? Show the calculations that led to your conclusions.

1. 81 and 625
2. 375 and 147
Every context free language is in $\mathbf{P}$

- This can be shown using the CYK algorithm, which determines whether string $w$ is derivable by CNF grammar $G$.
- The algorithm divides $w$, $|w| = n$, into substrings of length $i = 1, 2, \ldots, n$, and then determines the variables deriving each substring of length $i$.
- The algorithm begins with substrings of length $i = 1$.
- For subsequent lengths $i$, results for previous lengths are combined to provide answers for the current value of $i$.
- The CYK algorithm is an example of dynamic programming: a problem is subdivided into smaller ones, and solutions to the smaller problems are used to solve the larger one.
- Importantly, solutions to each smaller problem are recorded, so that work is not needlessly repeated.
1: procedure CYK(w_1 w_2 \ldots w_n) // \( G = (V, \Sigma, R, S) \).
2:     if \( w = \varepsilon \) and \( S \rightarrow \varepsilon \in R \) then // \( w = \varepsilon \) is a special case.
3:         return accept
4:     for \( i = 1 \) to \( i = n \) do
5:         for each \( A \in V \) do // find all variables yielding each \( w_i \)
6:             if \( A \rightarrow w_i \in R \) then
7:                 add \( A \) to \( \text{table}(i, i) \)
8:     for \( \text{len} = 2 \) to \( \text{len} = n \) do // compute variables for substring length \( \text{len} \).
9:         for \( i = 1 \) to \( i = n - \text{len} + 1 \) do
10:             \( j := i + \text{len} - 1 \) // \( j \) is the end of the substring.
11:         for \( \text{split} = i \) to \( \text{split} = j - 1 \) do
12:             for each \( A \rightarrow BC \in R \) do
13:                 if \( B \in \text{table}[i][\text{split}] \) and \( C \in \text{table}[\text{split} + 1][j] \) then
14:                     add \( A \) to \( \text{table}(i, j) \)
15:     if \( S \in \text{table}[1][n] \) then
16:         return accept
17:     else
18:         return reject
19: end procedure
We can assume that $G$, and hence it’s rules and variables, etc. are fixed. That is, they are constant.

Thus, we are only interested in the algorithm as the size $n$ of the input string changes.

Steps 2 and 3 can be done in constant time.

Since $V$ and $R$ are constant, Steps 4-7 take time $O(n)$.

The loops at 8, 9, and 11 run at most $n$ times.

The loop at 12 is bounded by a constant (the number of rules).

The remaining if statements, etc., can be done in constant time.

As such, the total running time of the algorithm is $O(n^3)$. 
Use the CYK algorithm to determine whether the following grammar generates string "babab".

\[
\begin{align*}
S & \rightarrow AB \mid AC \\
A & \rightarrow BC \mid AA \\
B & \rightarrow b \\
C & \rightarrow a
\end{align*}
\]
Difficult problems

Problems exist for which no polynomial time algorithms are known.

- A Hamiltonian path through a directed graph (digraph) $G$ is a path that enters every node exactly once.
- No polynomial time algorithms are known for $HAMPATH$:

$$HAMPATH = \{ \langle G, s, t \rangle | G \text{ is a digraph with a Hamiltonian path from } s \text{ to } t \}$$

- A brute force enumeration of all possible paths would decide the language, but it runs in exponential time.
- We would need only check that a found path is Hamiltonian.
Observe that it is easy to check that a given sequence of nodes is a path and that it is Hamiltonian.

*HAMPATH* is **polynomially verifiable**. If given a Hamiltonian path $p$, we can verify that $p$ is a Hamiltonian path in polynomial time. This is much easier than discovering a solution.

*COMPOSITES* = $\{x | x = pq \text{ for integers } p, q > 1\}$ is also polynomially verifiable. Given $p$ and $q$, we can easily check that $pq = x$. 

\[ s \xrightarrow{a} p \xrightarrow{b} q \xrightarrow{c} s \]
Polynomial Verifiability

- *HAMPATH* might not be polynomially verifiable. (It’s difficult to test for the non-existence of something—in this case a Hamiltonian path).

**Definition**

- A **verifier** for a language $A$ is an algorithm $V$, where
  \[ A = \{ w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c \} . \]
- The verifier halts on every input.
- A **polynomial verifier** runs in polynomial time in the length of $w$.
- String $c$ is called a **certificate** (**proof**) of $w$’s membership in $A$.
- For a polynomial verifier, $c$’s length is a polynomial of $w$’s length.
- Language $A$ is **polynomially verifiable** if it has a polynomial verifier.

- For *HAMPATH*, the certificate would be a Hamiltonian path from $s$ to $t$.
- For *COMPOSITES*, the certificate would be one of $x$’s divisors.
Polynomial NTM

The following is a nondeterministic Turing machine (NTM) $N_1$ that decides the $HAMPATH$ problem in nondeterministic polynomial time.

$N_1 =$ “On input $\langle G, s, t \rangle$, where $G$ is a directed graph with nodes $s$ and $t$:

1. Write a list of $m$ numbers, $p_1, p_2, \cdots, p_m$, where $m$ is the number of nodes in $G$. Each number in the list is nondeterministically selected to be between 1 and $m$.
2. Check for repetitions in the list. If any are found, reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
4. For each $i$ between 1 and $m - 1$, check whether $(p_i, p_{i+1})$ is an edge of $G$. If any are not, reject. Otherwise, all tests have been passed, so accept.”
The Class NP

**Definition**

NP is the class of languages that have polynomial time verifiers.

- P is clearly a subclass of NP. If a problem can be decided in polynomial time, then a polynomial verifier can be made (ignore the certificate).
- The term NP comes from “nondeterministic polynomial time”.
- NP is the class of languages that can be decided in polynomial time by nondeterministic single-tape Turing machines.

**Theorem**

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

- The idea is to convert a polynomial time verifier into a nondeterministic polynomial time decider, and vice versa.
The Class NP

Theorem

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

Proof.

For the forward direction of this theorem (LR), let \( A \in NP \) and show that \( A \) is decided by a polynomial time NTM \( N \). Let \( V \) be the polynomial time verifier of \( A \) that runs in time \( n^k \), where \( k \) is an integer constant. And construct \( N \) as follows.

\[ N = \text{“On input } w \text{ of length } n:\]

1. Nondeterministically generate a certificate \( c \) of length at most \( n^k \).
2. Run \( V \) on input \( \langle w, c \rangle \).
3. If \( V \) accepts, accept; if \( V \) rejects, reject.
The Class NP

Theorem

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

Proof.

To prove the other direction (LR), assume that $A$ is decided by a polynomial time NTM $N$ and construct a polynomial time verifier $V$ as follows.

$V = \text{"On input } \langle w, c \rangle, \text{ where } w \text{ and } c \text{ are strings:
1. Simulate } N \text{ on input } w, \text{ treating each symbol of } c \text{ as a description of the nondeterministic choice to make at each step (} c \text{ encodes a computation branch of } N).$

2. If this branch of $N$'s computation accepts, accept; otherwise, reject."
The Class NP

The nondeterministic time complexity class \( NTIME(t(n)) \).

**Definition**

\[ NTIME(t(n)) = \{ L | L \text{ is a language decided by a } O(t(n)) \text{ time NTM} \} \]

**Corollary**

\[ \mathbf{NP} = \bigcup_{k} NTIME(n^k) \]

- Like \( \mathbf{P} \), \( \mathbf{NP} \) is invariant under the choice of model.
- The problems \( \text{CLIQUE} \) and \( \text{SUBSET-SUM} \) are in \( \mathbf{NP} \).
Problems in NP: CLIQUE

- In an undirected graph, a **clique** is a set of nodes such that each pair in the clique is connected by an edge.
- A **k-clique** is a clique of $k$ nodes.

**Theorem**

$\text{CLIQUE} = \{\langle G, k \rangle | G \text{ is an undirected graph with a } k\text{-clique} \}$ is in $\text{NP}$. 

$\text{CSCI 2670 | Time Complexity (1)}$
Problems in NP: CLIQUE

Theorem

\[ \text{CLIQUE} = \{ \langle G, k \rangle | G \text{ is an undirected graph with a } k\text{-clique} \} \text{ is in } \mathbf{NP}. \]

Proof Idea: the clique is the certificate.

Proof.

The following is a verifier \( V \) for CLIQUE.

\( V = \) “On input \( \langle \langle G, k \rangle, c \rangle \):

1. Test whether \( c \) is a set of \( k \) nodes in \( G \).
2. Test whether \( G \) contains all edges connecting nodes in \( c \).
3. If both pass, accept; otherwise, reject.”
Theorem

\[ \text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \} \text{ is in } \text{NP}. \]

Alternative Proof: think of NP in terms of nondeterministic polynomial time Turing machines.

Proof.

\( N = \) “On input \( \langle G, k \rangle \), where \( G \) is a graph:

1. Nondeterministically select a subset \( S \) of \( k \) nodes of \( G \).
2. Test whether \( G \) contains all edges connecting nodes in \( S \): for every \( a, b \in S \) (with \( a, b \) distinct), check that an edge for \( a, b \) exists in \( G \).
3. If yes, accept; otherwise, reject.”
Given $S = \{x_1, \ldots, x_n\}$, and a number $t$, we want a subset of $S$ that sums to $t$.

**Theorem**

$\text{SUBSET-SUM}$ is in $\text{NP}$, where $\text{SUBSET-SUM}$ is

$$\{\langle S, t \rangle | S = \{x_1, \ldots, x_n\} \text{ and there is a } S' \subseteq S \text{ such that } \sum_{y \in S'} y = t\}.$$

For example, $\langle \{4, 11, 16, 21, 27\}, 25 \rangle \in \text{SUBSET-SUM}$ because $4 + 21 = 25$.

Note that $S$ and $S'$ are considered to be multisets and so allow repetition of elements.
Theorem

SUBSET-SUM is in NP, where SUBSET-SUM is

\[ \{(S, t) | S = \{x_1, \ldots, x_n\} \text{ and there is a } S' \subseteq S \text{ such that } \sum_{y \in S'} y = t\} \]

Proof Idea: the subset is the certificate.

Proof.

The following is a verifier \( V \) for SUBSET-SUM.

\( V = \text{"On input } (S, t, c)\text{:} \)

1. Test whether \( c \) is a collection of numbers that sum to \( t \).
2. Test whether \( S \) contains all the numbers in \( c \).
3. If both pass, accept; otherwise, reject."
Theorem

SUBSET-SUM is in **NP**, where SUBSET-SUM is

\[ \left\{ \langle S, t \rangle \mid S = \{x_1, \ldots, x_n\} \text{ and there is a } S' \subseteq S \text{ such that } \sum_{y \in S'} y = t \right\} \]

Alternative Proof: think of NP in terms of nondeterministic polynomial time Turing machines.

Proof.

\[ N = \text{“On input } \langle S, t \rangle:\]

1. Nondeterministically select a subset \( c \) of numbers in \( S \).
2. Test whether \( c \) is a collection of numbers that sum to \( t \).
3. If the test passes, accept; otherwise, reject."
P versus NP

- P is the class of languages decidable in polynomial time by deterministic TMs.
- NP is the class of languages decidable in polynomial time by nondeterministic TMs.

Or, more colloquially...

- P is the class of languages for which membership can be decided quickly.
- NP is the class of languages for which membership can be verified quickly.

- It should be clear that P is a subset of NP.
- What we don’t know is whether P is a proper subset of NP.
- That is, We don’t know whether P = NP or P ≠ NP.
P versus NP

▶ We don’t know whether $P = \text{NP}$ or $P \neq \text{NP}$.

▶ This is one of the greatest unsolved problems in theoretical computer science.

▶ If $P = \text{NP}$, then all problems in $\text{NP}$ would have deterministic polynomial time algorithms. That is, they would all be quickly solvable.

▶ Very few people think that $P$ is $\text{NP}$, but no one has been able to prove it.
P versus NP

- The best method known for solving languages in NP deterministically uses exponential time.

- And so, what we do know is that $\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k})$.

- $\text{EXPTIME}$ is the set of languages decidable by deterministic TMs with running times of $O(2^{p(n)})$, where $p(n)$ is a polynomial function of $n$.

- We don’t know whether $\text{NP}$ is in a smaller deterministic complexity class.
P is closed under union, concatenation, and complement.

NP is closed under union and concatenation.

Prove that

P is closed under complement.

NP is closed under concatenation.
\textbf{P} is closed under complement.

\textbf{Proof.}

For any language $L \in P$, let $M$ be the TM that decides it in polynomial time. We construct a TM $M'$ that decides the complement of $L$ in polynomial time:

$M' = \text{"On input } \langle w \rangle:\$

1. Run $M$ on $w$.
2. If $M$ accepts, reject. If it rejects, accept.
NP is closed under concatenation.

Proof.

For any two languages $L_1$ and $L_2$, $L_1, L_2 \in \text{NP}$, let $M_1$ and $M_2$ be the NTMs that decide them in polynomial time. We construct a NTM $M'$ that decides $L_1 \circ L_2$ in polynomial time: 

$$M' = \text{“On input } \langle w \rangle:$$

1. For each way to cut $w$ into two substrings $w = w_1w_2$:
2. Run $M_1$ on $w_1$.
3. Run $M_2$ on $w_2$. If both accept, accept; otherwise continue with the next choice of $w_1$ and $w_2$.
4. If $w$ is not accepted after trying all the possible cuts, reject.”
Let \( CONNECTED = \{ \langle G \rangle | G \text{ is a connected graph} \} \), show that \( CONNECTED \) is in \( P \).

Recall that a graph is \textbf{connected} if every node can be reached from every other node by traveling along the edges of the graph.
More Examples in P

Let $CONNECTED = \{ \langle G \rangle | G \text{ is a connected graph} \}$, show that $CONNECTED$ is in P.

Proof.

We construct a TM $M$ that decides $CONNECTED$ in polynomial time. $M =$ “On input $\langle G \rangle$ where $G$ is a graph:

1. Select the first node of $G$ and mark it.
2. Repeat the following state until no new nodes are marked:
3. For each node in $G$, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of $G$ to determine whether they all are marked. If they are, accept; otherwise, reject.”
Proof.

We construct a TM $M$ that decides $CONNECTED$ in polynomial time. $M = \text{"On input} \langle G \rangle \text{ where } G \text{ is a graph:}\$

1. Select the first node of $G$ and mark it.
2. Repeat the following state until no new nodes are marked:
   3. For each node in $G$, mark it if it is attached by an edge to a node that is already marked.
   4. Scan all the nodes of $G$ to determine whether they all are marked. If they are, accept; otherwise, reject.”

- Stage 1: $O(1)$
- Stage 2: at most $(n + 1)$ iteration for loops
- Stage 3: $O(n^2)$
- Stage 4: $O(n)$
- Therefore, the algorithm runs in $O(n^3)$ time and $CONNECTED$ is in P.
Show that $\text{ALL}_{DFA} = \{ \langle A \rangle | A \text{ is a DFA and } L(A) = \Sigma^* \}$ is in P.
Show that $ALL_{DFA} = \{ \langle A \rangle | A \text{ is a DFA and } L(A) = \Sigma^* \}$ is in P.

Proof.

We construct a TM $M$ that decides $ALL_{DFA}$ in polynomial time. $M =$ “On input $\langle A \rangle$ where $A$ is a DFA:

1. Construct DFA $B$ that recognizes $\overline{L(A)}$ by swapping the accept and non accept states in DFA $A$.
2. Run the decider $S$ of $E_{DFA}$ on input $\langle B \rangle$.
3. If $S$ accepts, accept. If $S$ rejects, reject.”

Can we determine if this decider $M$ runs in polynomial time?
Show that $ALL_{DFA} = \{\langle A \rangle | A \text{ is a DFA and } L(A) = \Sigma^* \}$ is in P.

**Proof.**

We construct a TM $M$ that decides $ALL_{DFA}$ in polynomial time. $M =$ “On input $\langle A \rangle$ where $A$ is a DFA:

1. Construct DFA $B$ that recognizes $\overline{L(A)}$ by swapping the accept and non accept states in DFA $A$.
2. Test whether a path exists from the start state to each accept state in $B$.
3. If no such path exists, accept; otherwise, reject.”
Show the following languages are in P.

1. Let $\text{MODEXP} = \{\langle a, b, c, p \rangle | a, b, c, \text{ and } p \text{ are binary integers such that } a^b \equiv c \pmod{p} \}$.

2. A permutation on the set $\{1, 2, \ldots, k\}$ is a one-to-one, onto function on this set. When $p$ is a permutation, $P^t$ means the composition of $p$ with itself $t$ times. Let $\text{PERM-POWER} = \{\langle p, q, t \rangle | p = q^t \text{ where } p \text{ and } q \text{ are permutations on } \{1, 2, \ldots, k\} \text{ and } t \text{ is a binary integer}\}$. 

Hint: note that the most obvious algorithm does not run in polynomial time. Try it first where $b$ (for question 1) and $t$ (for question 2), each is a power of 2.