Time Complexity

Pierre-Alain Fouque

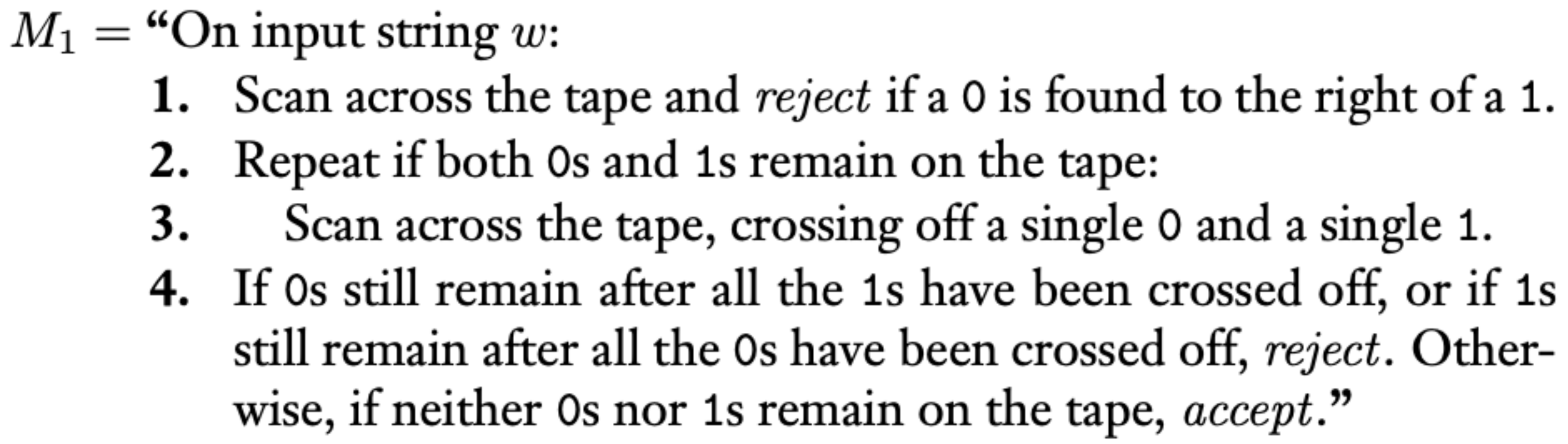
Introduction

- Even when a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or memory
- Time and Space complexity theory studies the time and memory or other resources required for solving computational problems

1. Measuring the time used to solve a problem
2. Show how to classify problems according to the amount of time required
3. Certain decidable problems require enormous amount of time
4. How to determine when you are faced with such a problem ?

Measuring Complexity

- A={$0^k 1^k$ | k≥0} is a decidable language.
- How much time does a single-tape Turing Machine need to decide A ?

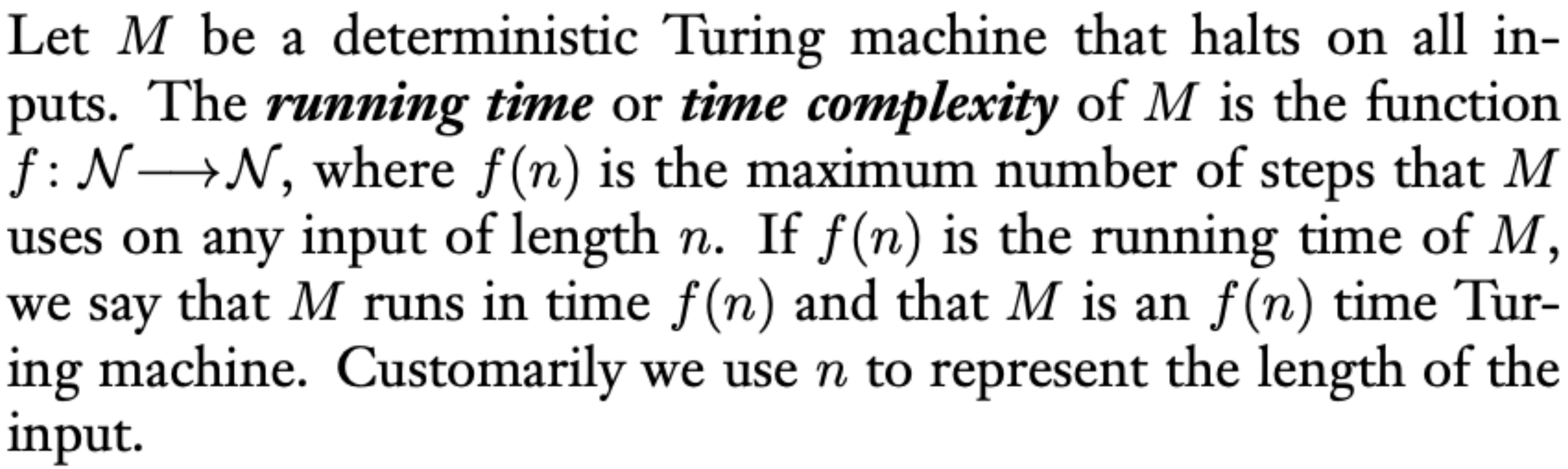


```
$M_1$ = "On input string $w$:
    1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
    2. Repeat if both 0s and 1s remain on the tape:
    3.     Scan across the tape, crossing off a single 0 and a single 1.
    4. If 0s still remain after all the 1s have been crossed off, or if 1s
       still remain after all the 0s have been crossed off, *reject*. Other-
       wise, if neither 0s nor 1s remain on the tape, *accept*."
```

Analysis the algorithm for TM M_1 deciding A to determine how much time it uses

- Number of steps that an algorithm uses on a particular input may depend on several parameters
- E.g.: if the input is a graph, the number of steps may depend on the number of nodes, the number of edges, and the maximum degree of the graph, or a combination of these
- For simplicity, we compute the running time of an algorithm purely as a function of the length of the string representing the input and don't consider any other parameters
- In worst-case analysis, we consider the longest running time of all inputs of a particular length
- In average-case analysis, the average of all the running times of inputs of a particular length

Definition



Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f\colon \mathcal{N}\longrightarrow\mathcal{N}$, 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 $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Big-O and small-O notation

- The exact running time is often a complex expression, we estimate it
- One convenient form of estimation: asymptotic analysis, we seek to understand the running time of the algorithm when it is run on large inputs
- We consider only the highest order term of the expression for the running time of the algorithm, disregarding both coefficient of that term and any lower terms, because the highest order term dominates the other terms on large inputs
- E.g.: $f(n)=6n^3+2n^2+20n+45$ has 4 terms and the highest is $6n^3$. The asymptotic notation or big-O notation is $f(n)=O(n^3)$

Asymptotic upper bound

Let f and g be functions $f, g\colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,

$$f(n) \leq c\, g(n).$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.

Intuition

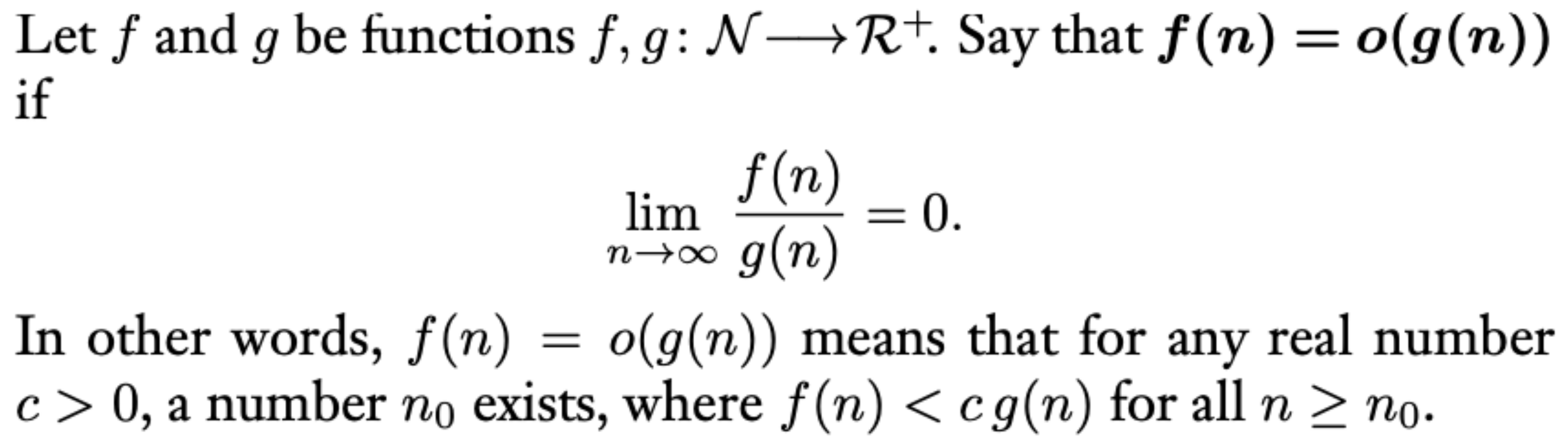
- $f(n)=O(g(n))$ means that f is less than or equal to g if we disregard differences up to a constant factor. Think of O as representing a suppressed constant
- Most functions f have an obvious highest order term h, write $f(n)=O(g(n))$, where g is h without its coefficient
- E.g.: $f_1(n)=5n^3+2n^2+22n+6$. Select the highest term $5n^3$, remove coefficient 5 gives $f_1(n)=O(n^3)$. If $c=6$ and $n_0=10$, $5n^3+2n^2+22n+6 \le 6n^3$ for all $n \ge 10$. $f_1(n)=O(n^4)$ since n^4 is larger than n^3 and is an asymptotic upper bound on f_1. However, $f_1(n)$ is not $O(n^2)$.

Examples

- The big-O interacts with logarithms. When we use log, we must specify the base, $x=\log_2 n$, meaning $2^x=n$. Changing the base b, changes $\log_b n$: $\log_b n=\log_2 n/\log_2 b$. When we write $f(n)=O(\log n)$ there is no necessary to specify the base because we suppress the constant
- E.g. $f_2(n)=2n\log_2 n+ 5n\log_2\log_2 n+2$: $f_2(n)=O(n\log n)$ as log n dominates log log n.
- Arithmetic expressions: $f(n)=O(n^2)+O(n)=O(n^2)$
- When $f(n)=2^{O(n)}$ represents an upper bound of 2^{cn} for some constant c
- $f(n)=2^{O(\log n)}$: since $n=2^{\log_2 n}$, $n^c=2^{c\log_2 n}$, $2^{O(\log n)}$ is an upper bound n^c for some c. $n^{O(1)}$ represents the same bound.
- n^c polynomial bounds, $2^{(n^d)}$ exponentials

Small-o notation

- Big-O says that one function is asymptotically no more than another
- Small-o says that one function is asymptotically less than another



Let f and g be functions $f, g\colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if

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

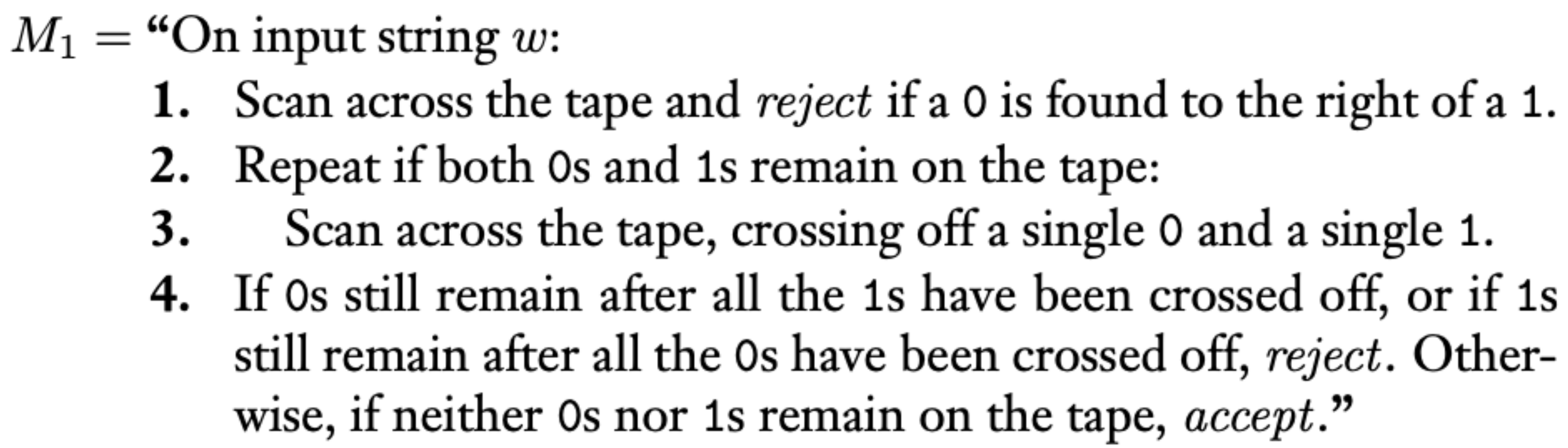
In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < c\,g(n)$ for all $n \geq n_0$.

Examples

1. $\sqrt{n}=o(n)$
2. $n=o(n\log\log n)$
3. $n\log\log n = o(n\log n)$
4. $n\log n=o(n^2)$
5. $n^2=o(n^3)$

However, $f(n)$ is never $o(f(n))$

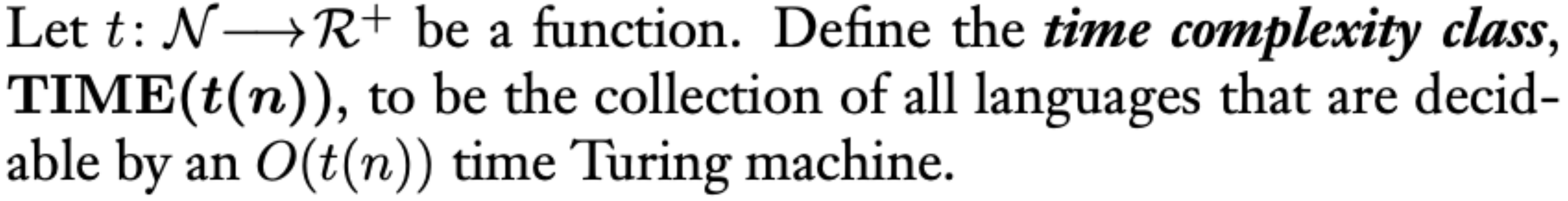
Analyzing algorithms



```
M_1 = "On input string w:
  1. Scan across the tape and reject if a 0 is found to the right of a 1.
  2. Repeat if both 0s and 1s remain on the tape:
  3.     Scan across the tape, crossing off a single 0 and a single 1.
  4. If 0s still remain after all the 1s have been crossed off, or if 1s
     still remain after all the 0s have been crossed off, reject. Other-
     wise, if neither 0s nor 1s remain on the tape, accept."
```

- Stages 1 and 4: O(n)
- Stages 2 and 3, O(n) steps and each scan crosses off 2 symbols, at most n/2 steps apart: $(n/2)O(n)=O(n^2)$ steps
- Total time of M_1: $O(n)+O(n^2)+O(n)=O(n^2)$

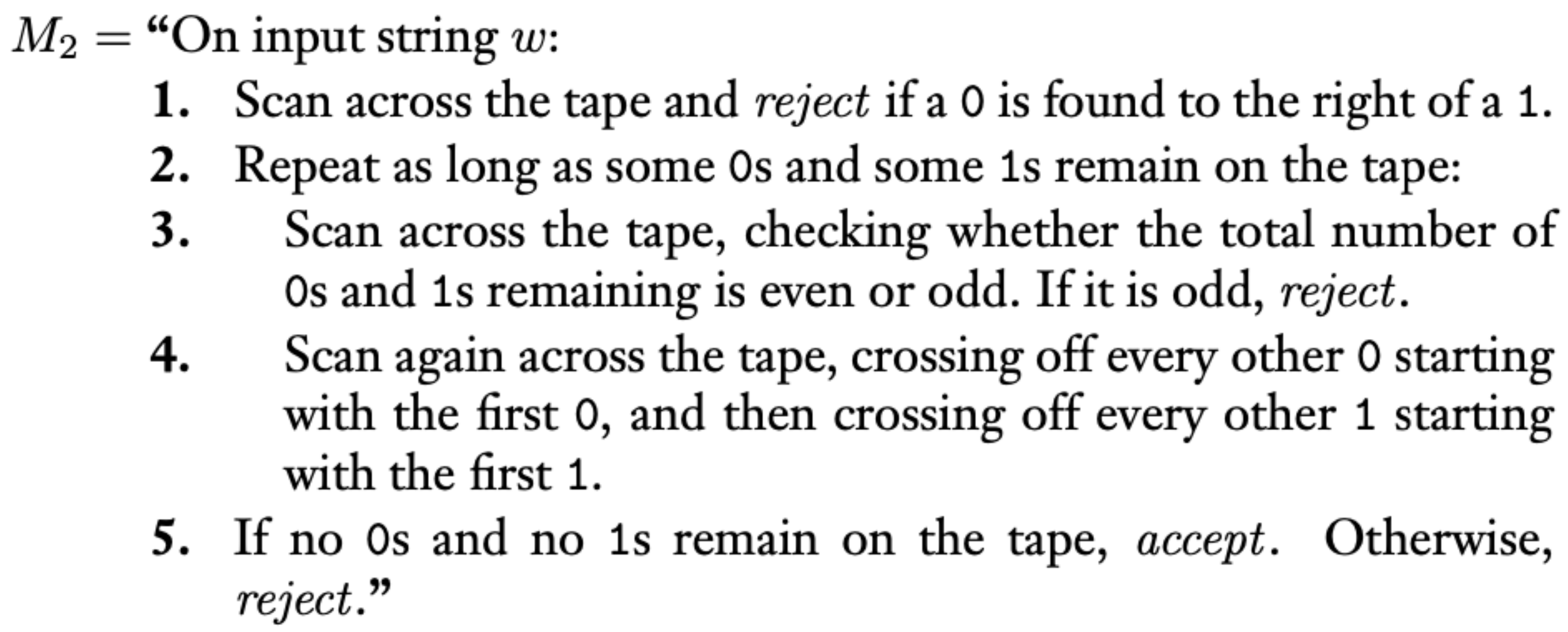
Time complexity class



Let $t\colon \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, $\textbf{TIME}(\boldsymbol{t(n)})$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

- A∈TIME(n^2) because M_1 decides A in time O(n^2) and TIME(n^2) contains all languages than can be decided in O(n^2) times.
- Is there a machine that decides A asymptotically more quickly ?
- Is A in TIME(t(n)) for t(n)=o(n^2) ? Crossing 2s and 1s changes the running time by a factor 2, but not the asymptotic running time

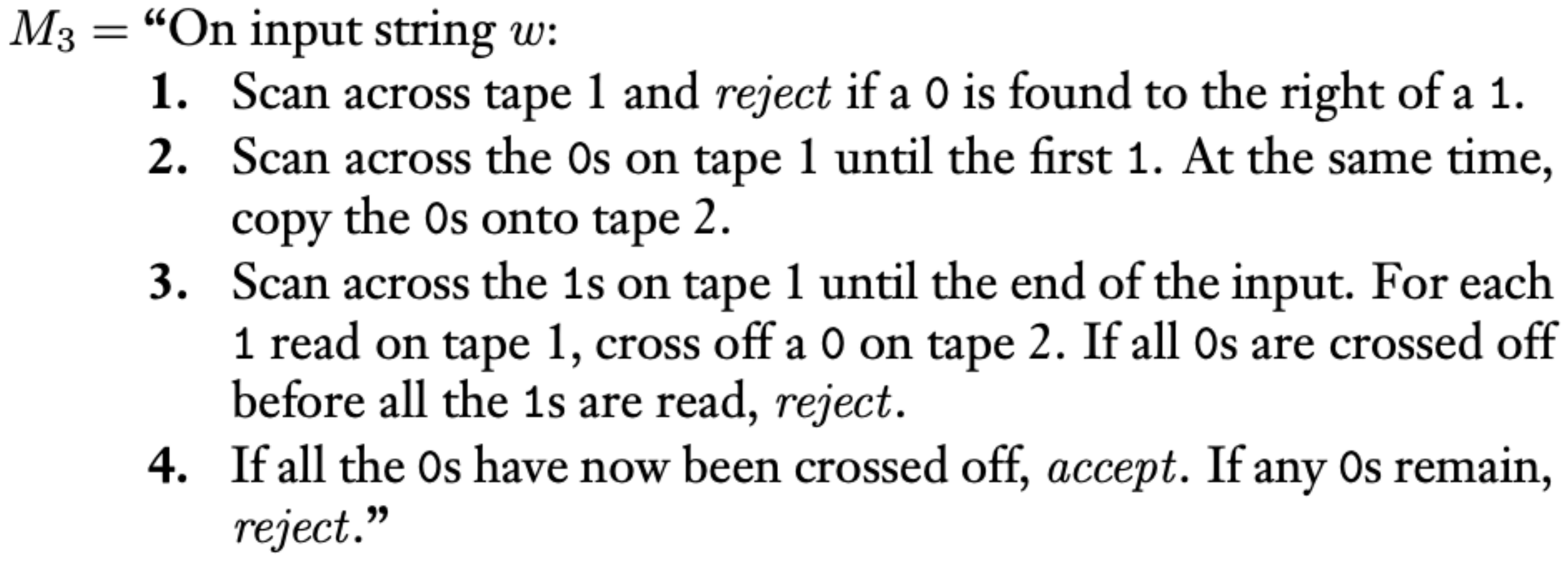
More efficient machine



```
$M_2$ = "On input string $w$:
    1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
    2. Repeat as long as some 0s and some 1s remain on the tape:
    3.     Scan across the tape, checking whether the total number of
           0s and 1s remaining is even or odd. If it is odd, *reject*.
    4.     Scan again across the tape, crossing off every other 0 starting
           with the first 0, and then crossing off every other 1 starting
           with the first 1.
    5. If no 0s and no 1s remain on the tape, *accept*. Otherwise,
       *reject*."
```

- M_2 is a more efficient machine and shows that $A \in TIME(n \log n)$
- This result cannot be further improved on single-tape machine. On single-tape machine, any language that can be decided in $o(n \log n)$ is regular.

A is in O(n): Linear Time complexity

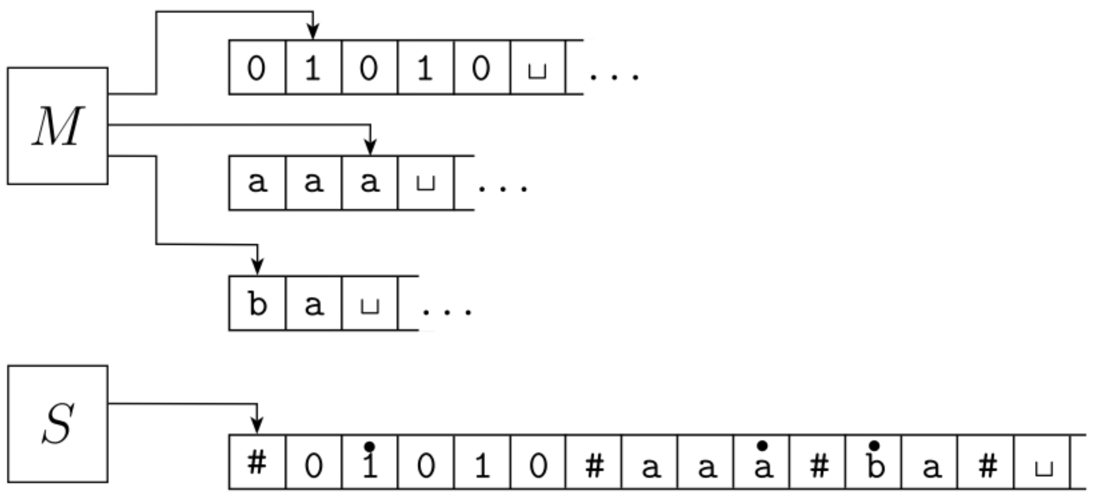


```
$M_3$ = "On input string $w$:
    1. Scan across tape 1 and *reject* if a 0 is found to the right of a 1.
    2. Scan across the 0s on tape 1 until the first 1. At the same time,
       copy the 0s onto tape 2.
    3. Scan across the 1s on tape 1 until the end of the input. For each
       1 read 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*."
```

- Note that the complexity of A depends on the model of computation selected !
- Important difference between complexity theory and computability theory. In computability, all reasonable models are equivalent (they decide the same language)
- Language that are decided in linear time on one model aren't necessarily decided in linear time on another. Fortunaltely, time requirements don't differ greatly for typical deterministic models.

Complexity relationships among models

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.



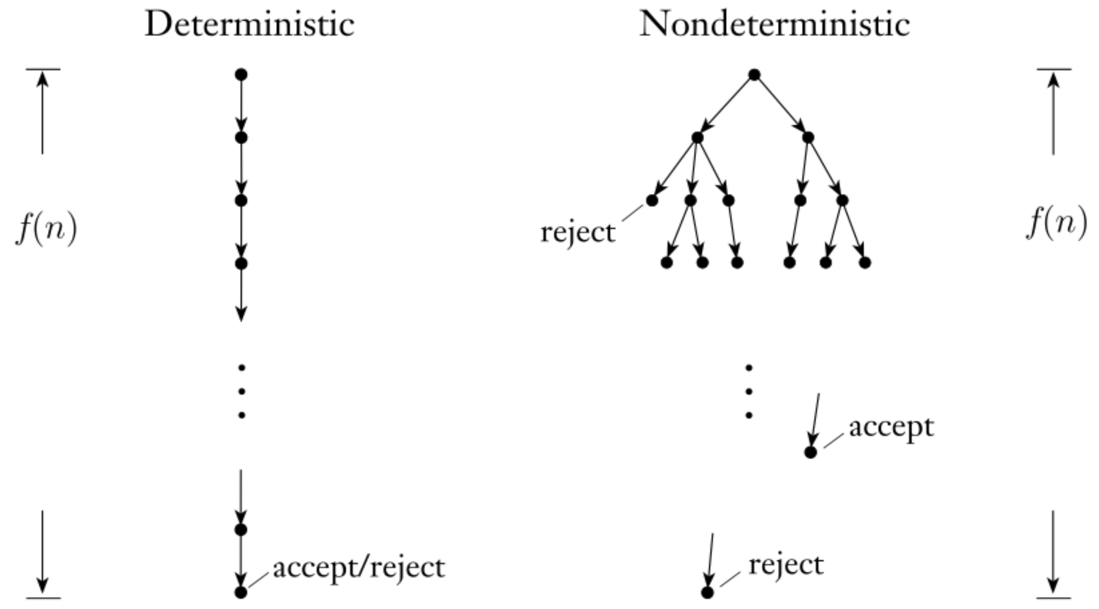
$S =$ "On input $w = w_1 \cdots w_n$:

1. First S puts its tape into the format that represents all k tapes of M. The formatted tape contains
$$\#\overset{\bullet}{w_1} w_2 \cdots w_n \#\overset{\bullet}{\sqcup}\#\overset{\bullet}{\sqcup}\# \cdots \#.$$
2. To simulate a single move, S scans its tape from the first #, which marks the left-hand end, to the $(k+1)$st #, which marks the right-hand end, in order to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to the way that M's transition function dictates.
3. If at any point S moves one of the virtual heads to the right onto a #, this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape. So S writes a blank symbol on this tape cell and shifts the tape contents, from this cell until the rightmost #, one unit to the right. Then it continues the simulation as before."

Initial stage: O(n), each active portion takes time at most t(n), S simulates at most t(n) steps requiring each O(t(n)) and so t(n)xO(t(n))=O(t^2(n)) steps.

Running time of non-deterministic Turing machines

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f\colon \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.



The definition of the running time of a non-deterministic TM is not intended to correspond to any real-world computing device

It is a useful mathematical definition that asserts in characterizing the complexity of an important class of computational problems

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

The class P

- The two previous theorems illustrate an important distinction

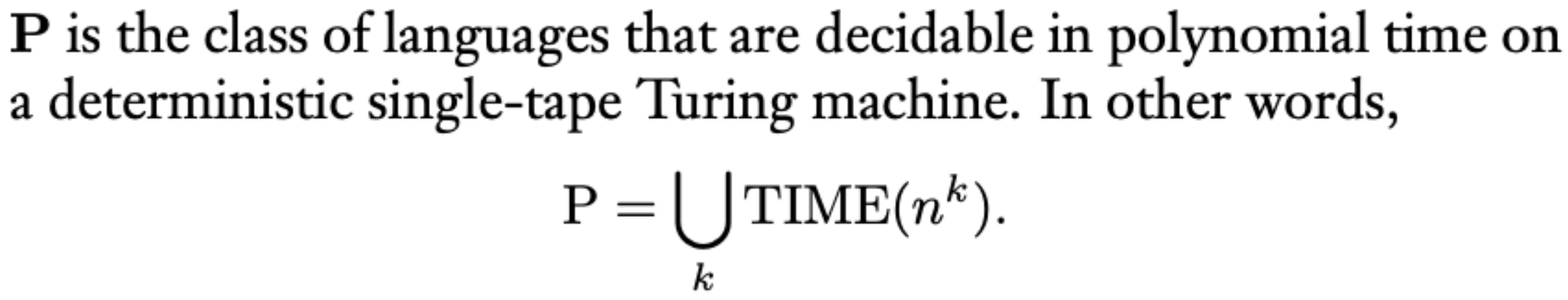
- We showed that at most a square or polynomial difference between the time complexity of problems measured on deterministic single-tape and multi-tape TM

- We showed that at most an exponential difference between the time complexity of problems on deterministic and non-deterministic TM

Polynomial Time

- Polynomial differences in the running time are considered to be small, whereas exponential differences are considered to be large
- Dramatic difference between polynomials as n^3 and exponential 2^n: for example if n=1000, 1 billion is large but manageable number whereas 2^n is much larger than the number of atoms in the universe
- Polynomial time algorithms are fast enough for any purposes, bnut exponential time algorithms rarely are useful
- Exponential time algorithms arise when we solve problems by exhaustively searching through a space of solutions: brute-force search
- All reasonsable deterministic computational models are polynomially equivalent
- Disregarding polynomial differences in running time can be seen as absurd since programmers care about such difference and even constant factor…
- But, disregarding polynomial differnces doesn’t imply that we consider them as unimportant. Questions such as the polynomiality or non-polynomiality of the factoring problem do not depend on the polynomial differences

Definition P



P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

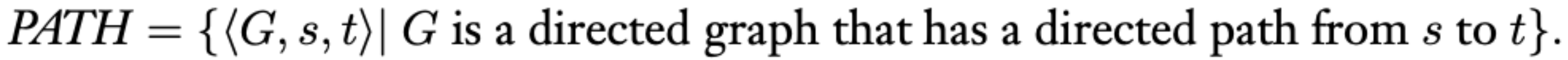
$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

- The class P plays a central role in our theory and is important because

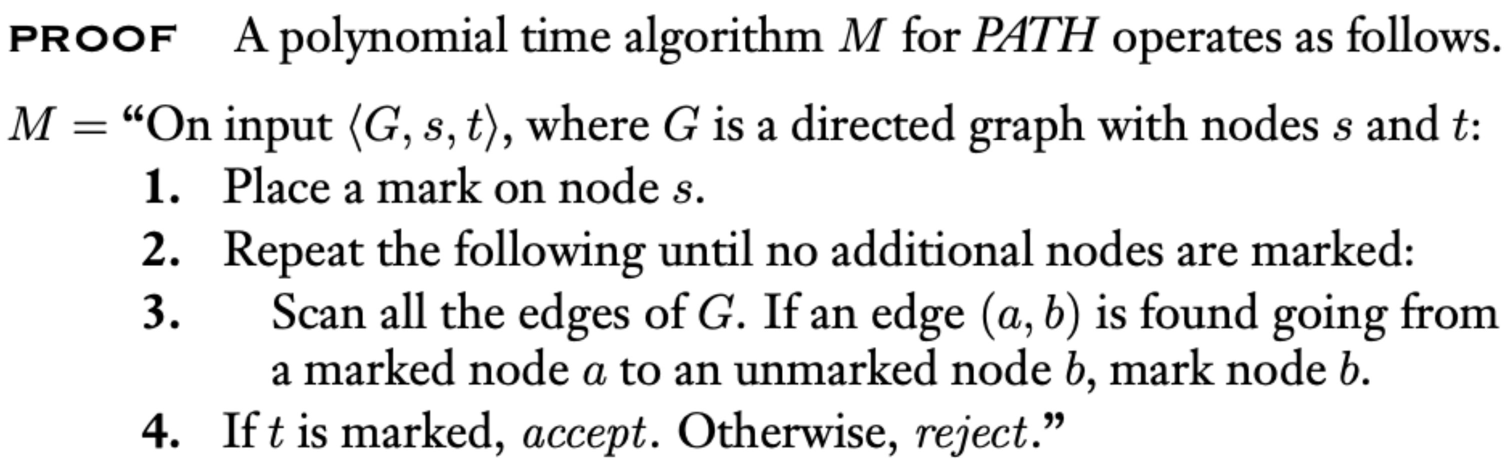
1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer. (Of course, a running time of n^{100} is unlikely to be practical, but in general we can decrease the degree.)

Examples of problems in P

- One important point is the encoding method: any reasonable method except the unary notation, in any other base



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



PROOF A polynomial time algorithm M for *PATH* operates as follows.

```
M = "On input ⟨G, s, t⟩, where G is a directed graph with nodes s and t:
    1. Place a mark on node s.
    2. Repeat the following until no additional nodes are marked:
    3.     Scan all the edges of G. If an edge (a, b) is found going from
           a marked node a to an unmarked node b, mark node b.
    4. If t is marked, accept. Otherwise, reject."
```

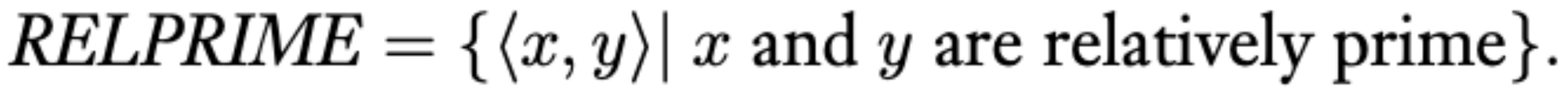
Thm: PATH ∈ P

Brute-force algorithm:

With m nodes, number of paths of length at most m is m^m

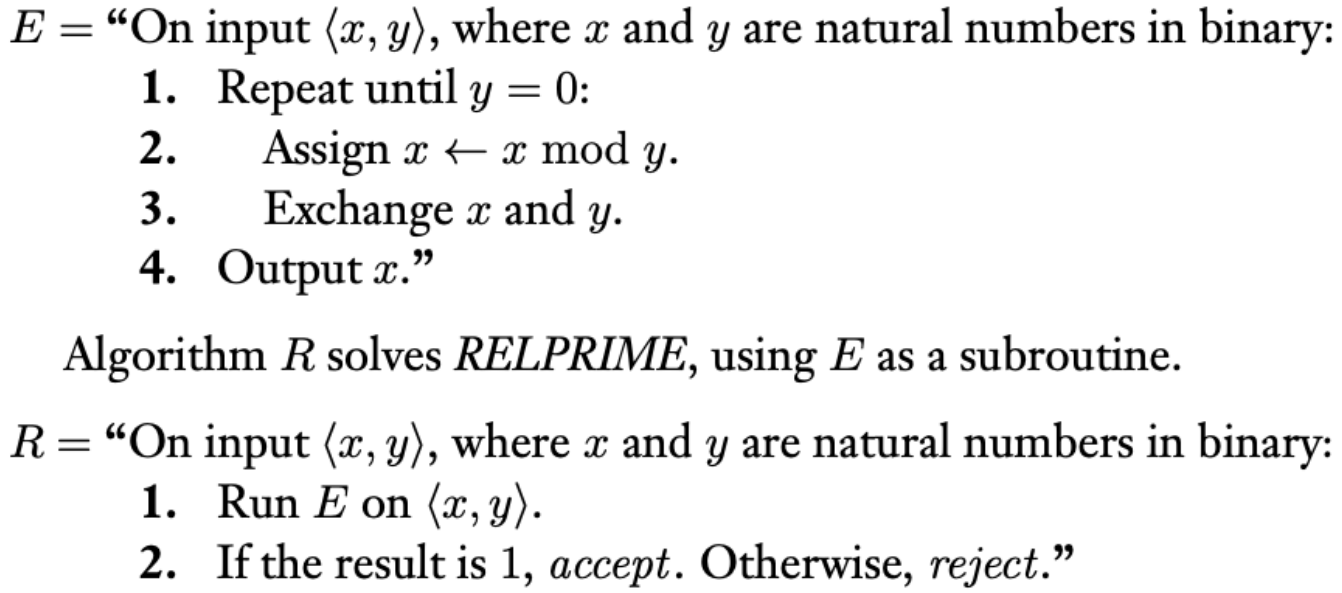
Exponential in m

RELPRIME ∈P



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

- Eucliean algorithm: at every execution of stage 2 in E (except maybe the first), the value x is cut by at least 2:
  - After Stage 2, x<y (because mod)
  - After Stage 3, x>y, Stage 2 is run: if x/2≥y, x mod y<y≤x/2 and x drops by at least half. If x/2<y, x mod y = x-y<x/2 and x drops by at least half.



```
E = "On input ⟨x, y⟩, where x and y are natural numbers in binary:
  1. Repeat until $y = 0$:
  2.     Assign $x \leftarrow x \bmod y$.
  3.     Exchange $x$ and $y$.
  4. Output $x$."
```

Algorithm R solves *RELPRIME*, using E as a subroutine.

```
R = "On input ⟨x, y⟩, where x and y are natural numbers in binary:
  1. Run $E$ on $\langle x, y\rangle$.
  2. If the result is 1, accept. Otherwise, reject."
```

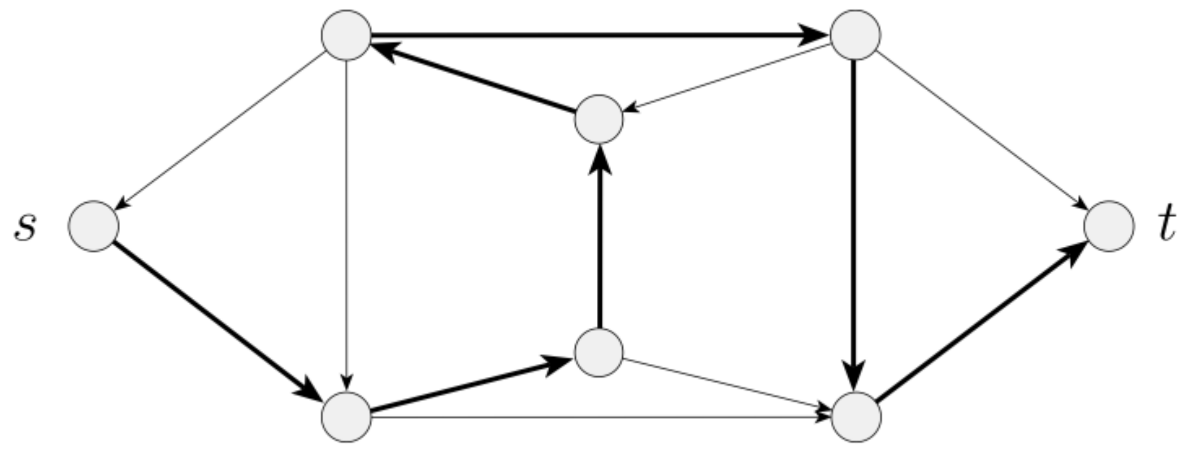
The class NP

- Sometimes, attempts to avoid brute-force in certain problems, including many interesting and useful ones, haven’t been successful, and polynomial time algorithms that solve them aren’t known to exist
- Why have we been unsuccessful in finding polynomial time algorithms for these problems ? We don’t know the answer to this important question. Perhaps these problems have as yet undiscovered polynomial time algoreithms that rest on unknown principles. Or possibly some of these problems simply cannot be solved in polynomial time. They are intrinsically difficult.
- Remarkable discovery concerning this question shows that the complexities of many problems are linked: a polynomial time algorithm for one such problem can be used to solve an entire class of problems

Hamiltonian path

- A Hamiltonian path in a directed graph G is a directed path that goes through each node exactly once. We consider the problem of testing whether a directed graph contains a Hamiltonian path connecting two specified nodes

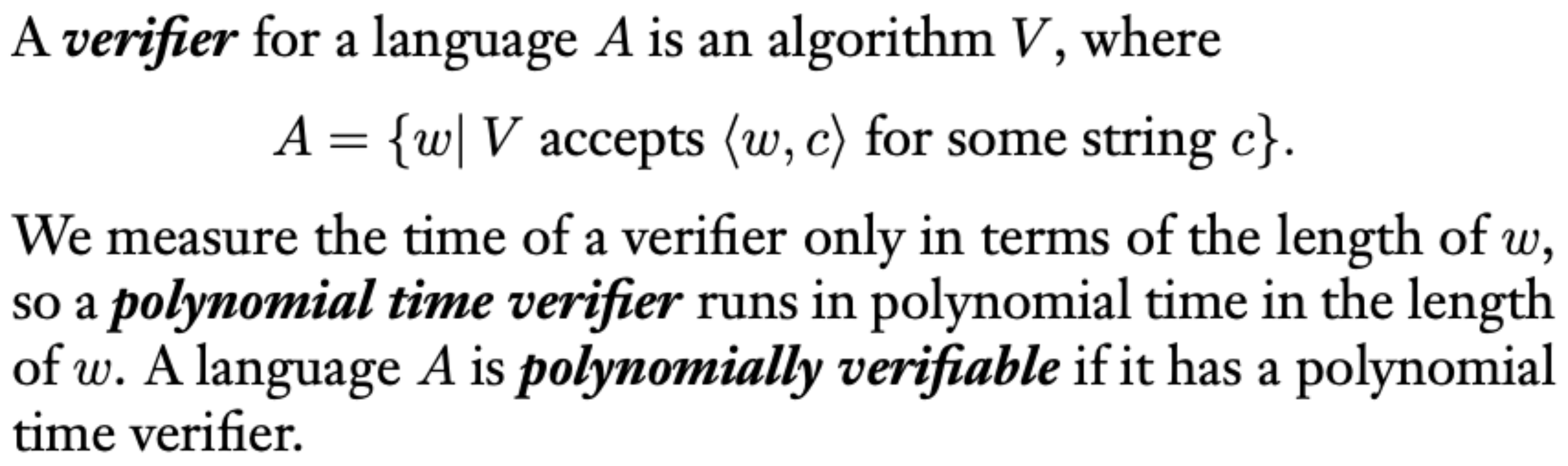
$$\begin{aligned} \mathit{HAMPATH} = \{\langle G, s, t\rangle \mid\ & G \text{ is a directed graph} \\ & \text{with a Hamiltonian path from } s \text{ to } t\}. \end{aligned}$$



HAMPATH Problem

- We can obtain an exponential time algorithm for the HAMPATH problem by modifying the brute-force algorithm for PATH. We only need to add a check to verify that the potential path is Hamiltonian
- No one knows whether HAMPATH is solvable in polynomial time
- The HAMPATH problem has a feature called polynomial verifiability that is important for understanding its complexity: we don’t know of a fast way to determine whether a graph contains a Hamiltonian path, but if such a path is discovered, we can easily convince someone by presenting it
- Another polynomially verifiable problem is compositeness: find two factors and multiply them
- Some problems may not be polynomially verifiable: The complement of HAMPATH. How can we convince someone that there is no Hamiltonian path ?

Polynomial Verifiable



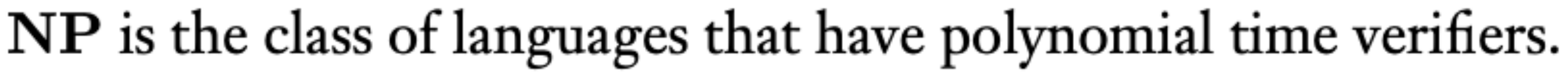
A ***verifier*** for a language A is an algorithm V, where

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

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

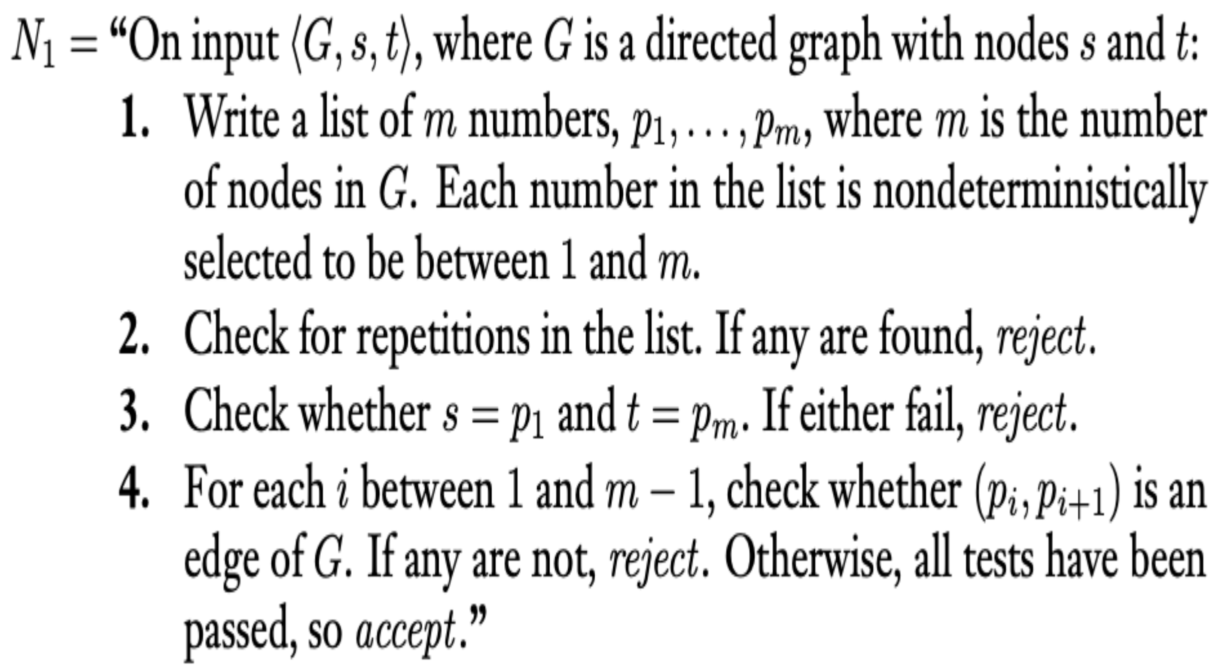
- A verifier uses additional information represented by the symbol c (called a certificate or a proof) to verify that a string w is a member of A
- For polynomial verifiers, the certificate has polynomial length (in the lenght of w) because that is all the verifier can access in its time bound
- E.f.: HAMPATH and COMPOSITES problems have polynomial time verifiers

NP definition



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

- The class NP is important because it contains many problems of practical interest as HAMPATH and COMPOSITES. COMPOSITES is a member of P but proving this stronger result is difficult
- NP term comes from nondeterministic polynomial time (alternative characterization): Problems in NP are called NP-problems



```
$N_1$ = "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, $p_1, \ldots, p_m$, 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."
```

NP definition

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

(=>) Let A∈NP and show that A is decided by a polynomial time NTM N. Let V be the polynomial time verifier for A that exists by the definition of NP. Assume that V is a TM that runs in time n^k and construct N as follows:

```
$N =$ "On input $w$ of length $n$:
    1. Nondeterministically select string $c$ of length at most $n^k$.
    2. Run $V$ on input $\langle w, c \rangle$.
    3. If $V$ accepts, accept; otherwise, reject."
```

(<=) Assume that A is decided by a polynomial time NTM N and construct a polynomial verifier V as follows:

```
$V =$ "On input $\langle w, c \rangle$, where $w$ and $c$ are strings:
    1. Simulate $N$ on input $w$, treating each symbol of $c$ as a description
       of the nondeterministic choice to make at each step (as in
       the proof of Theorem 3.16).
    2. If this branch of $N$'s computation accepts, accept; otherwise,
       reject."
```

Complexity class NTIME

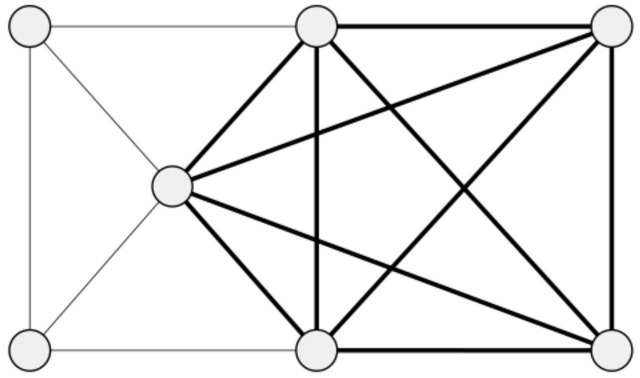
$$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k).$$

- The class NP is insensitive to the choice of reasonable non-deterministic computational model because all such models are polynomially equivalent

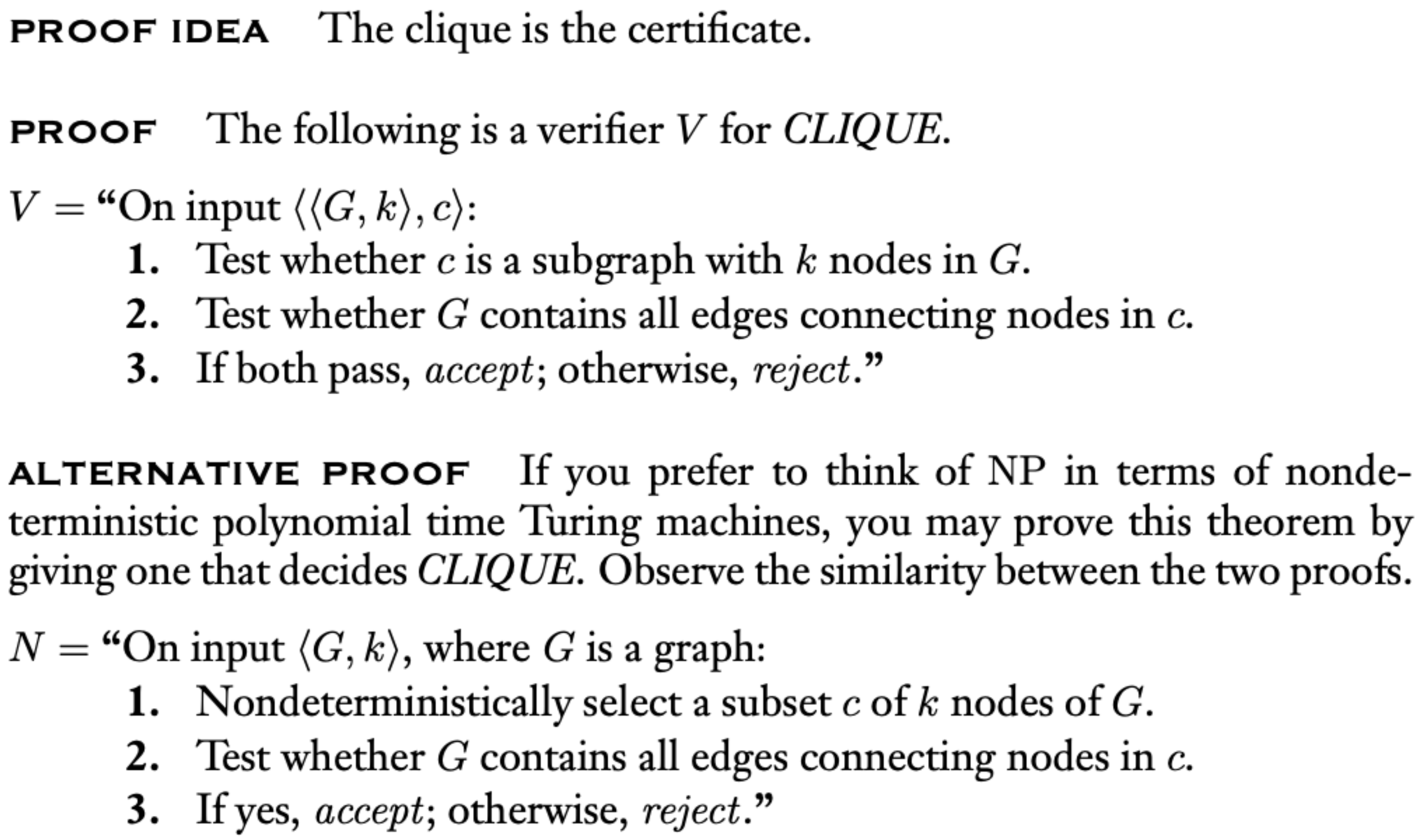
Examples of problems in NP

- A clique in a undirected graph is a subgraph, wherein every two nodes are connected by an edge. A k-clique is a clique that contains k nodes. E.g. A graph with a 5-clique



- The clique problem is to determine whether a graph contains a clique of a specified size: CLIQUE = {<G,k>| G is an undirected graph with a k-clique}

CLIQUE is in NP



PROOF IDEA The clique is the certificate.

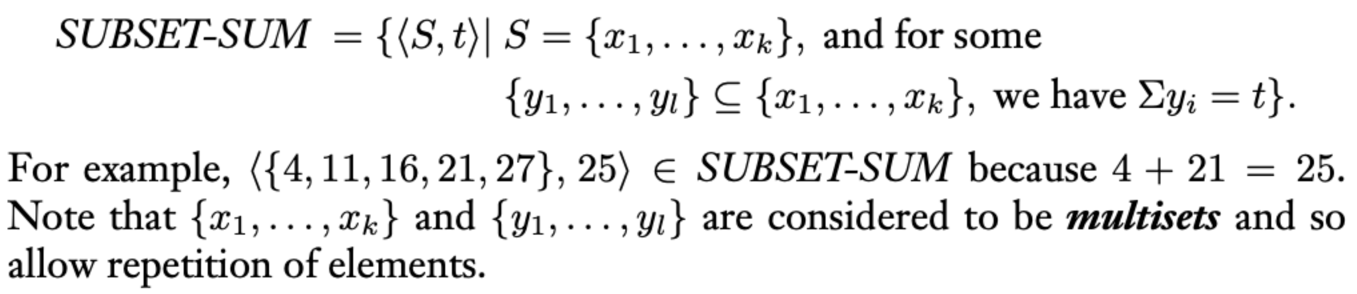
PROOF The following is a verifier V for *CLIQUE*.

```
V = "On input ⟨⟨G, k⟩, c⟩:
    1. Test whether c is a subgraph with k nodes in G.
    2. Test whether G contains all edges connecting nodes in c.
    3. If both pass, accept; otherwise, reject."
```

ALTERNATIVE PROOF If you prefer to think of NP in terms of nondeterministic polynomial time Turing machines, you may prove this theorem by giving one that decides *CLIQUE*. Observe the similarity between the two proofs.

```
N = "On input ⟨G, k⟩, where G is a graph:
    1. Nondeterministically select a subset c of k nodes of G.
    2. Test whether G contains all edges connecting nodes in c.
    3. If yes, accept; otherwise, reject."
```

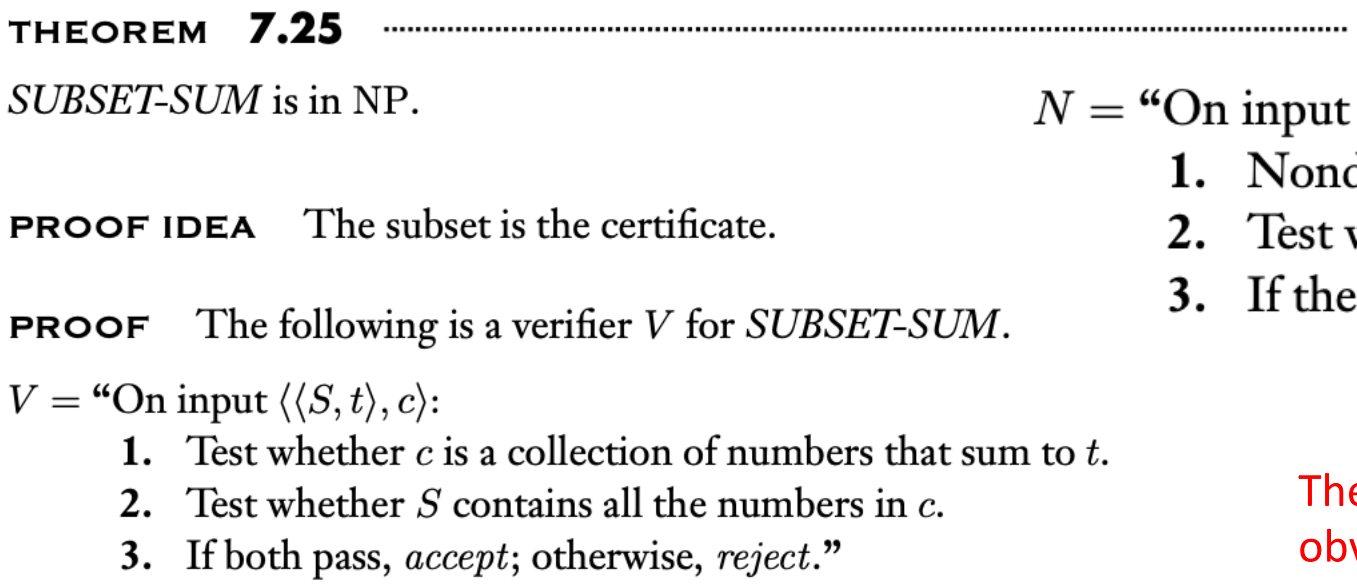
SUBSET-SUM Problem



$$SUBSET\text{-}SUM = \{\langle S,t\rangle \mid S=\{x_1,\ldots,x_k\}, \text{ and for some } \{y_1,\ldots,y_l\}\subseteq\{x_1,\ldots,x_k\}, \text{ we have } \Sigma y_i = t\}.$$

For example, $\langle\{4,11,16,21,27\},25\rangle \in$ *SUBSET-SUM* because $4+21=25$. Note that $\{x_1,\ldots,x_k\}$ and $\{y_1,\ldots,y_l\}$ are considered to be ***multisets*** and so allow repetition of elements.

- Given a set of number x_1, …., x_k and a target number t, determine whether the collection contains a subset that adds up to t



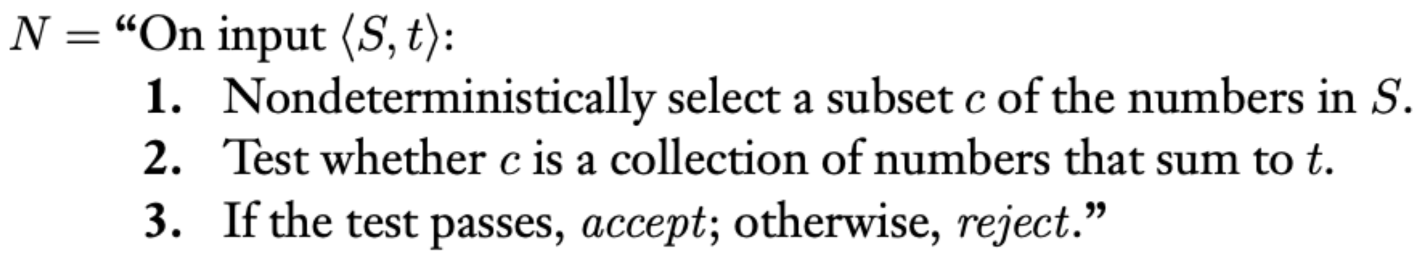
THEOREM 7.25

SUBSET-SUM is in NP.

PROOF IDEA The subset is the certificate.

PROOF The following is a verifier V for *SUBSET-SUM*.

```
V = "On input ⟨⟨S, t⟩, c⟩:
    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."
```

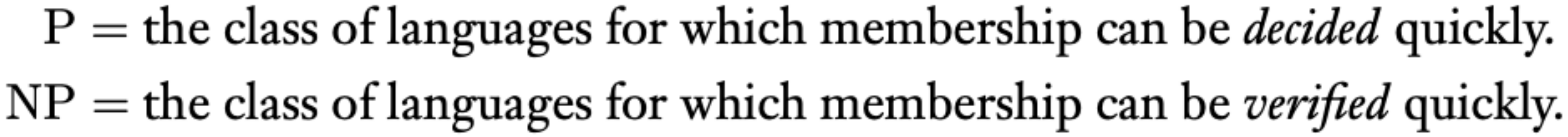


```
N = "On input ⟨S, t⟩:
    1. Nondeterministically select a subset $c$ of the numbers in $S$.
    2. Test whether $c$ is a collection of numbers that sum to $t$.
    3. If the test passes, accept; otherwise, reject."
```

The complement of CLIQUE and SUBSET-SUM are not obvious members of NP. Verifying that something is not present seems more difficult than verifying it is present

The P versus NP question

- NP is the class of languages that are solvable in polynomial time on a Non-deterministic TM or whereby membership in the language can be checked in polynomial time
- P is the class of languages where membership can be tested in polynomial time.


P = the class of languages for which membership can be *decided* quickly.
NP = the class of languages for which membership can be *verified* quickly.

P vs. NP ?

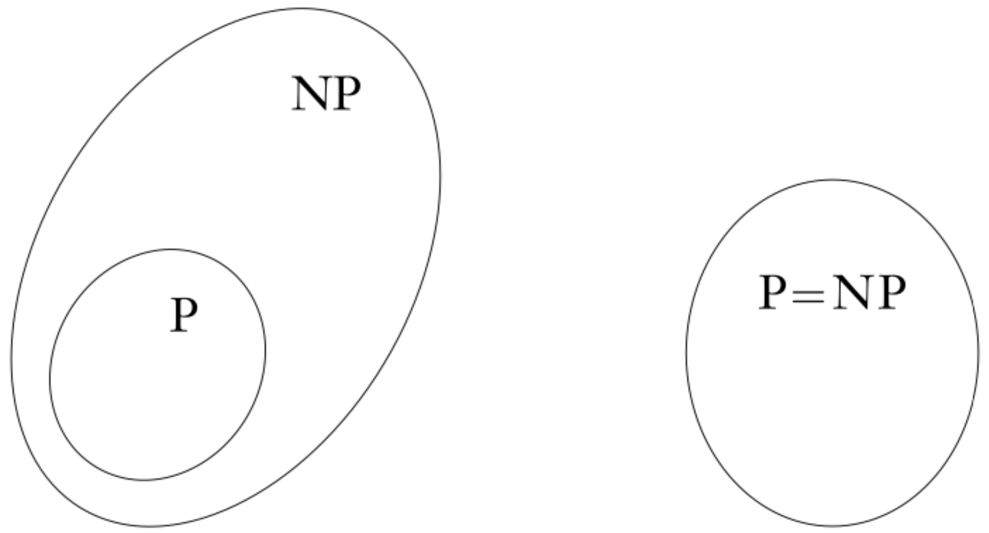


FIGURE 7.26
One of these two possibilities is correct

The best deterministic method currently known for deciding languages in NP uses exponential time. In other words, we can prove that

$$\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k}),$$

but we don't know whether NP is contained in a smaller deterministic time complexity class.