Complexity Classes

Warm-up question (4/27/22)

What are TWO things you would like me to do on Friday’s class (last class)?

Send me your response by Canvas email.

Complexity relationships among models

- We consider three models to examine how the choice of computational model can affect the time complexity of languages:
  - Single-tape Turing Machine
  - Multiple-tape Turing Machine
  - Nondeterministic Turing Machine

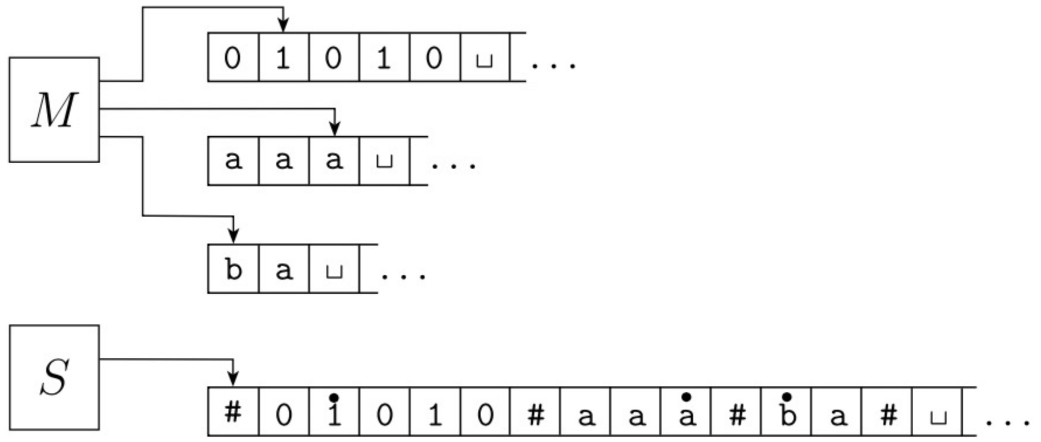
Single-tape vs multi-tape

Theorem:

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.

PROOF IDEA The idea behind the proof of this theorem is quite simple. Recall that in Theorem 3.13, we showed how to convert any multitape TM into a single-tape TM that simulates it. Now we analyze that simulation to determine how much additional time it requires. We show that simulating each step of the multitape machine uses at most $O(t(n))$ steps on the single-tape machine. Hence the total time used is $O(t^2(n))$ steps.

Simulation of multi-tape on single tape TM (old slide from lecture on 3/27)



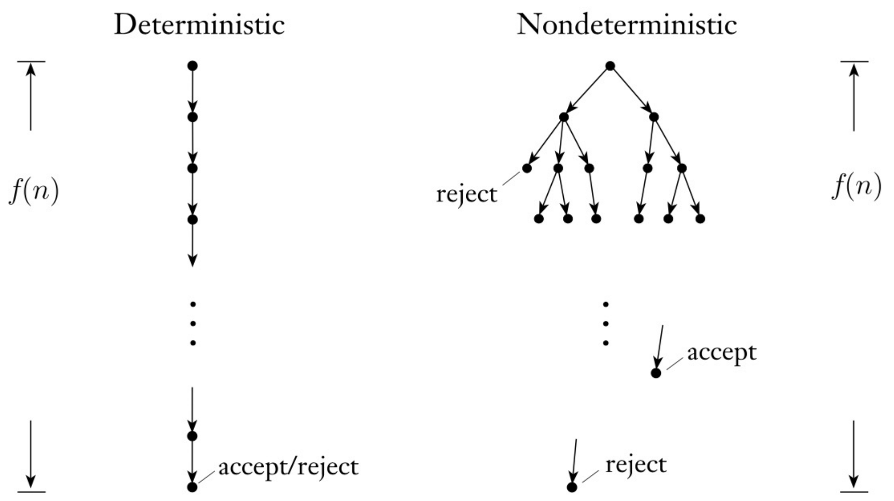
Simulation Idea:

- Say M has k tapes
- S simulates the effect of k tapes by storing their information on its single tape
- Delineate tapes: S uses the new symbol # as a delimiter to separate contents of the different tapes
- Track tape heads: S writes a dot above a tape symbol to mark the place where the head on that tape would be

Deterministic vs nondeterministic TM

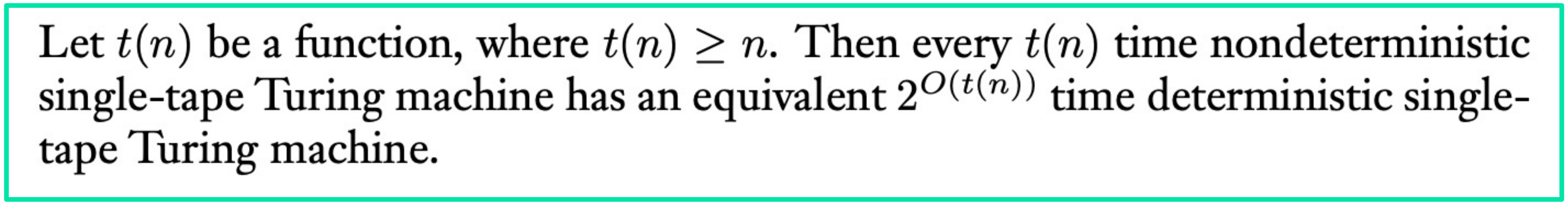
Definition:

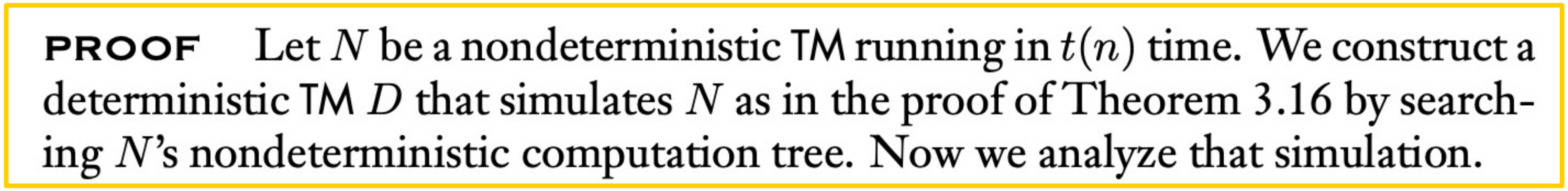
Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f\colon \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.



Deterministic vs nondeterministic TM

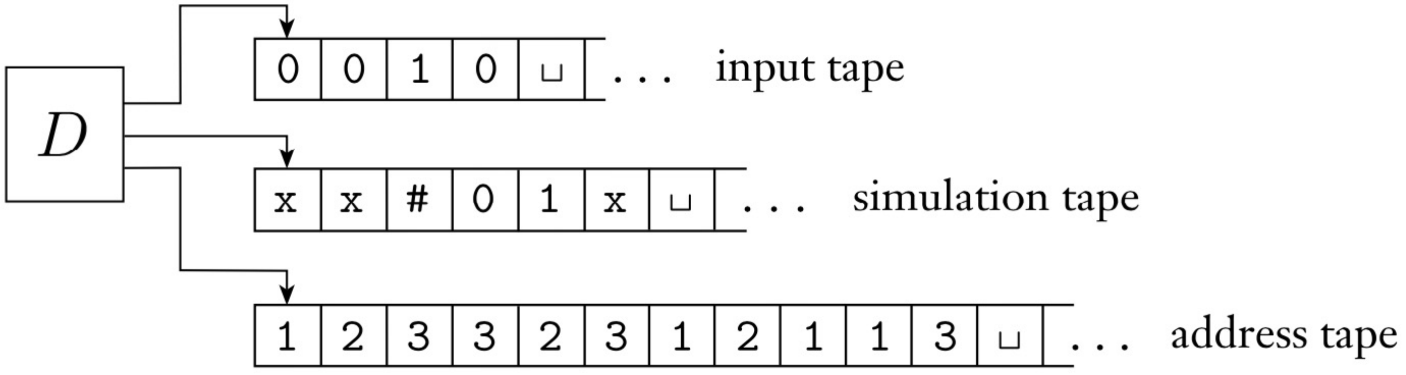
Theorem:


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.


PROOF Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N as in the proof of Theorem 3.16 by searching N's nondeterministic computation tree. Now we analyze that simulation.

Simulation of nondeterministic TM on deterministic TM (old slide from lect. in March)

- The simulating DTM D has three tapes (this is equivalent to having a single tape)
- Tape 1: contains the input string, never altered
- Tape 2: maintains a copy of N's tape on some branch of its ND computation
- Tape 3: keeps track of D's location in N's ND computation tree



Classes of problems

- The two results we saw (single- vs multi-tape TM; deterministic vs non-deterministic TM) illustrate an important distinction

- On the one hand, we saw at most a *square*, or polynomial, difference between the time complexity of problems measured on deterministic single-tape and multi-tape TMs

- On the other hand, we saw at most an exponential difference between the time complexity of problems on deterministic and nondeterministic TMs

Polynomial Time

- For our purposes, polynomial differences in running time are considered small, whereas exponential differences are considered large
- This separation between polynomials and exponentials (rather than between some other classes of functions) is a good one for several reasons:
  - There is a dramatic difference between the growth rate of typically occurring polynomials (such as n^3) and typically occurring exponentials such as 2^n
  - Exponential time algorithms typically arise when we solve problems by exhaustively searching through a space of solutions (brute-force search). Sometimes brute-force search maybe avoided through a deeper understanding of a problem, which may reveal a polynomial time algorithm
  - 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.

The Class P

Definition:

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

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

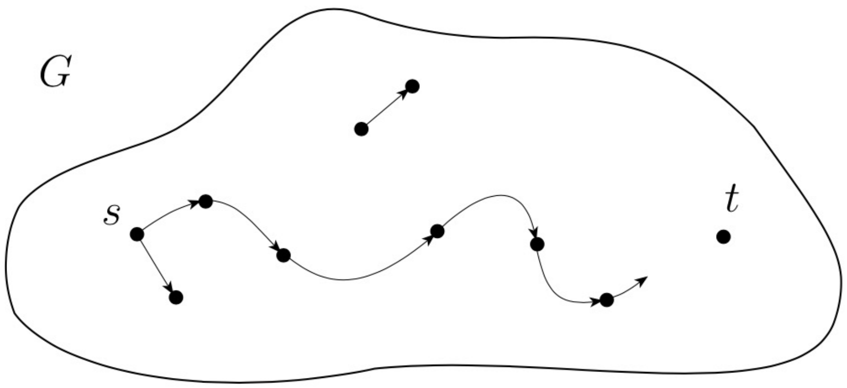
The class P plays a central role in our theory and is important because

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.

Examples of problems in P

Example 1

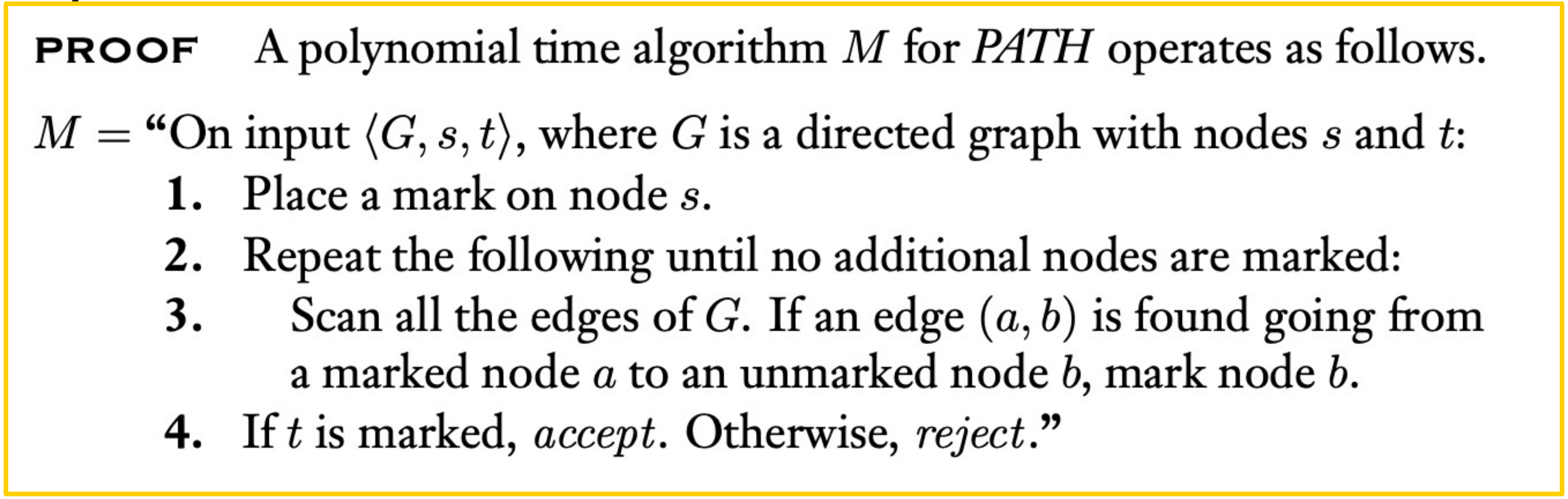
$$\boxed{PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.}$$



Theorem:

$$\boxed{PATH \in \mathrm{P}.}$$

Proof of PATH is in P



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."
```

- Stages 1 and 4 are executed only once
- Stage 3 runs at most m times, where m is the number of nodes
- Thus, total number of stages is 1+1+m, giving a polynomial
in the size of G
- Stages 1 and 4 of M are easily implemented in polynomial time
on any reasonable deterministic model. Stage can also be easily
implemented in polynomial time
- Hence M is a polynomial time algorithm

Examples of problems in P

Example 2

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

Theorem:

$$\mathit{RELPRIME} \in \mathrm{P}.$$

Proof: uses Euclidean algorithm for gcd

Examples of problems in P

Example 3

Context free languages

Theorem:

Every context-free language is a member of P

Proof: key ingredients of the proof are use of

- a CFL generated by a CFG in Chomsky normal form
- *Dynamic programming*

The Class NP

Definition:

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

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

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.

Definition:

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

The Class NP

Theorem:

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

Definition:

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

Corollary:

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

Examples of problems in NP

Example 1

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

Example 2

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

The P vs NP question

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

Whether P = NP is one of the greatest unsolved problems in theoretical computer science.

One of the two possibilities depicted in this figure is possible.