CSC363 Tutorial 8

What does P stand for?

Paul “sushi_enjoyer” Zhang

University of Humongus Chungus Amogus Pekora 🥕

March 9, 2021

What does P stand for?

P stands for **Practical!**[1]

Not really. P stands for *polynomial-time*. It is formally defined as

$$P = \bigcup_{k=1}^{\infty} \text{TIME}(n^k)$$

where $\text{TIME}(n^k)$ is the set of languages that can be decided by a $O(n^k)$ TM.

Less formally, P is the set of languages that can be decided by a poly-time TM.

Remember: just because *some* algorithm for deciding a language A isn't poly-time does not mean *all* algorithms for deciding A aren't poly-time.

[1] not really. some problems that we think are not in P are still of practical importance! for example, the **Travelling Salesman Problem (TSP)** is believed to not be in P, yet we need it for stuff like, uh, sushi delivery? 🍣 🍣

What does P stand for?

P stands for **Practical!**[1]

Not really. P stands for *polynomial-time*. It is formally defined as

$$P = \bigcup_{k=1}^{\infty} \mathrm{TIME}(n^k)$$

where $\mathrm{TIME}(n^k)$ is the set of languages that can be decided by a $O(n^k)$ TM.

Less formally, P is the set of languages that can be decided by a poly-time TM.

Remember: just because *some* algorithm for deciding a language A isn't poly-time does not mean *all* algorithms for deciding A aren't poly-time.

[1] not really. some problems that we think are not in P are still of practical importance! for example, the **Travelling Salesman Problem (TSP)** is believed to not be in P, yet we need it for stuff like, uh, sushi delivery? 🍣 🍣

What does P stand for?

P stands for **Practical!**[1]

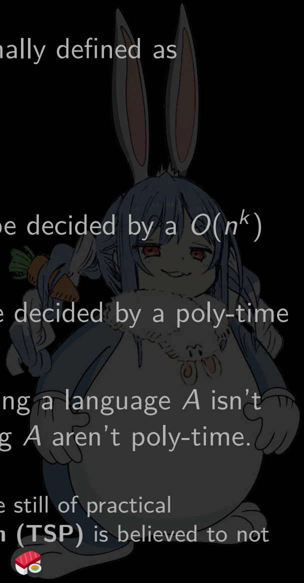
Not really. P stands for *polynomial-time*. It is formally defined as

$$\mathrm{P} = \bigcup_{k=1}^{\infty} \mathrm{TIME}(n^k)$$

where $\mathrm{TIME}(n^k)$ is the set of languages that can be decided by a $O(n^k)$ TM.

Less formally, P is the set of languages that can be decided by a poly-time TM.

Remember: just because *some* algorithm for deciding a language A isn't poly-time does not mean *all* algorithms for deciding A aren't poly-time.



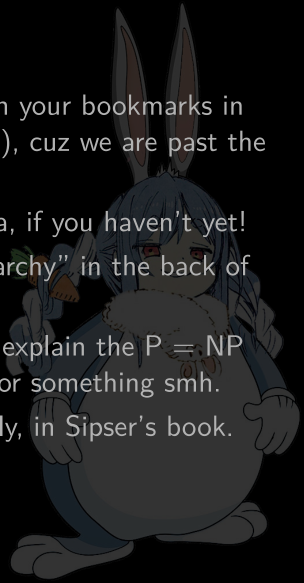
[1]not really. some problems that we think are not in P are still of practical importance! for example, the **Travelling Salesman Problem (TSP)** is believed to not be in P, yet we need it for stuff like, uh, sushi delivery? 🍣 🍣

Learning objectives this tutorial

By the end of this tutorial, you should...

- Remove `turingmachinesimulator.com` from your bookmarks in Microsoft Edge (or whichever browser you use), cuz we are past the low-level Turing machine stuff :(
- Watch the video on P vs NP pinned on Piazza, if you haven't yet!
- Have a brief image of the "computability hierarchy" in the back of your head.
- Understand what NP means. Now you get to explain the P $=$ NP problem to your friends and be a cool person or something smh.

`helo_fish.jpg` certified readings: 7.1-7.3, probably, in Sipser's book.



When Turing machine is sus 😳 😳

A non-deterministic Turing machine takes the next CSC363 quiz. It *chooses* the correct answers.

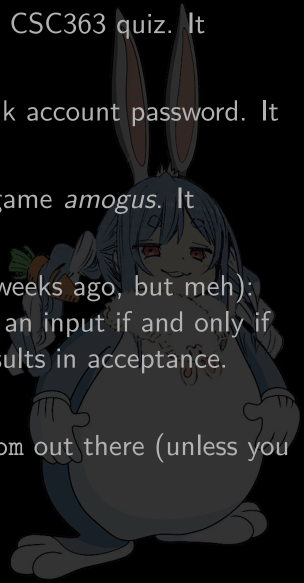
A non-deterministic Turing machine forgets its bank account password. It *chooses* the correct password.

A non-deterministic Turing machine plays the hit game *amogus*. It *chooses* the imposter to vote out.

Recall (i'm not sure if you do, from the tutorial 2 weeks ago, but meh): we say a non-deterministic turing machine accepts an input if and only if there is *some* execution path for the NTM that results in acceptance.

Unfortunately, there isn't any `nondeterministicturingmachinesimulator.com` out there (unless you want to create it!).

When Turing machine is sus 😳 😳



A non-deterministic Turing machine takes the next CSC363 quiz. It *chooses* the correct answers.

A non-deterministic Turing machine forgets its bank account password. It *chooses* the correct password.

A non-deterministic Turing machine plays the hit game *amogus*. It *chooses* the imposter to vote out.

Recall (i'm not sure if you do, from the tutorial 2 weeks ago, but meh): we say a non-deterministic turing machine accepts an input if and only if there is *some* execution path for the NTM that results in acceptance.

Unfortunately, there isn't any `nondeterministicturingmachinesimulator.com` out there (unless you want to create it!).

H-path

Task: Guess what the H stands for in HPATH. I'll actually give you 0.2 bonus marks on your next assignment if you know the answer.

Answer: H stands for *Hamilton*. Not from Ontario. Hamilton is this Irish lad:

helo

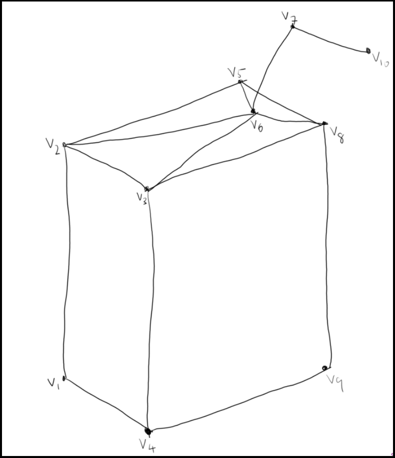
H-path

Task: Guess what the H stands for in HPATH. I'll actually give you 0.2 bonus marks on your next assignment if you know the answer.
Answer: H stands for *Hamilton*. Not from Ontario. Hamilton is this Irish lad:

helo

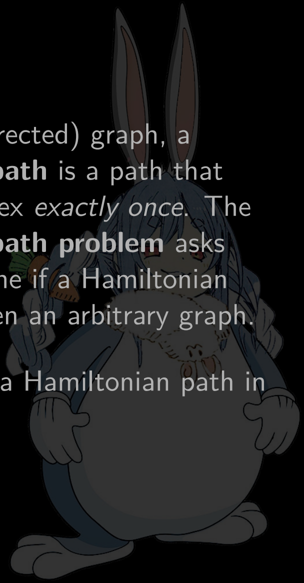
H-path

sowwy, tikz is too tedious ;-; you'll have to refer to my hand drawn graphs instead.



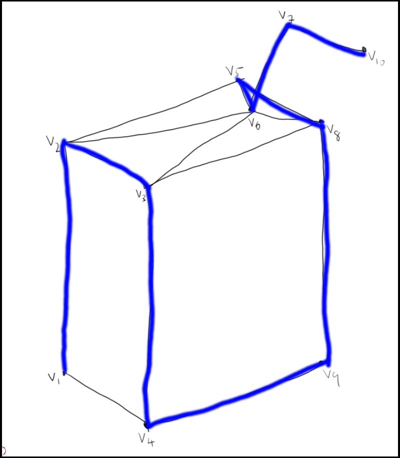
Given an (undirected) graph, a **Hamiltonian path** is a path that visits each vertex *exactly once*. The **Hamiltonian path problem** asks you to determine if a Hamiltonian path exists given an arbitrary graph.

Task: Is there a Hamiltonian path in this graph?

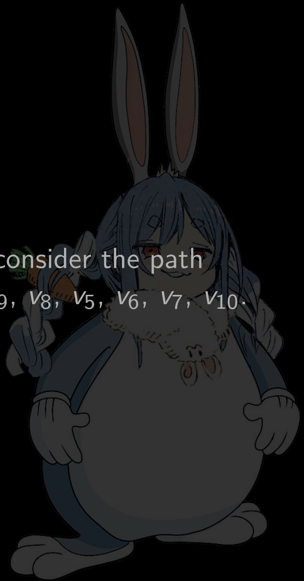


H-path

sowwy, tikz is too tedious ;-; you'll have to refer to my hand drawn graphs instead.

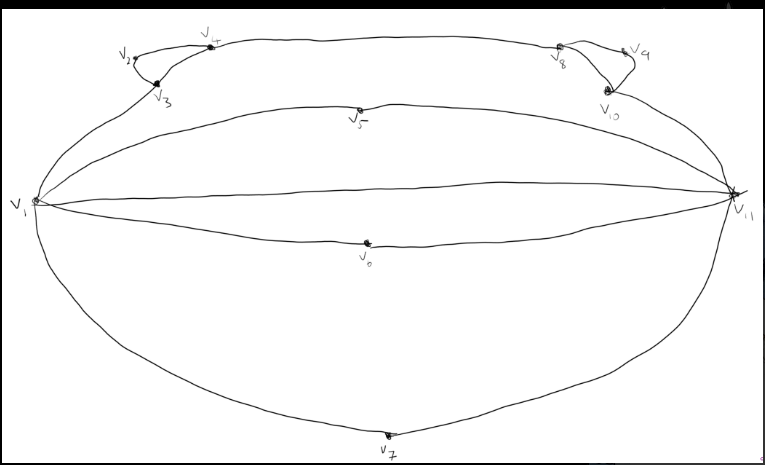


Answer: yes! consider the path $v_1, v_2, v_3\ v_4, v_9, v_8, v_5, v_6, v_7, v_{10}$.



H-path

sowwy, tikz is too tedious ;-; you'll have to refer to my hand drawn graphs instead.

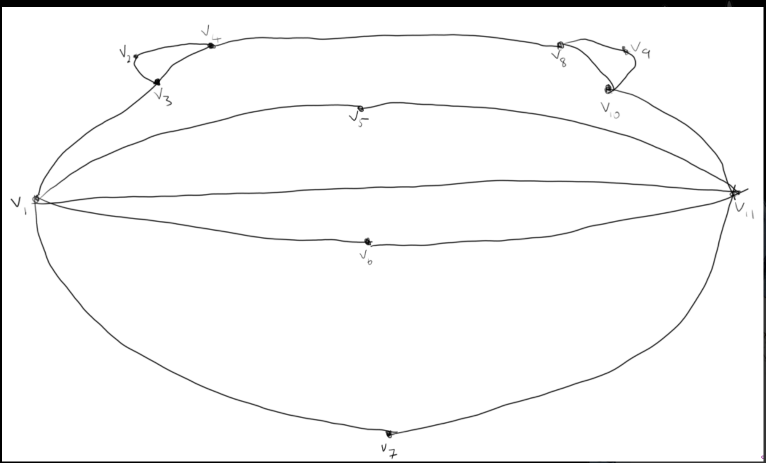


Task: Is there a Hamiltonian path in this graph?

Answer: No. (I've checked every possible Hamiltonian path).

H-path

sowwy, tikz is too tedious ;-; you'll have to refer to my hand drawn graphs instead.

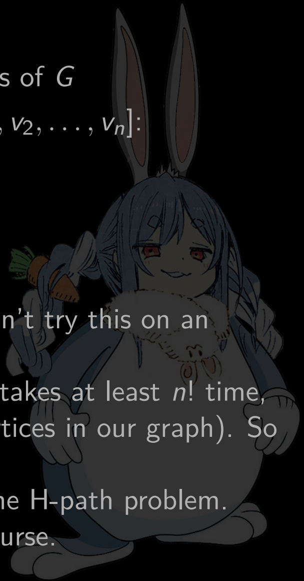


Task: Is there a Hamiltonian path in this graph?
Answer: No. (I've checked every possible Hamiltonian path).

H-path

Here's a deterministic TM M to check if there is a H-path, given a graph input G:

```
$M(G)$ :Let $v_1, v_2, \ldots, v_n$ be the vertices of $G$
        For every permutation $P$ of $[v_1, v_2, \ldots, v_n]$:
            If $P$ is a valid path in $G$:
                Accept
        Reject
```



But this is very inefficient, and you probably wouldn't try this on an interview.

Task: Convince yourself that the above algorithm takes at least $n!$ time, where n is the size of the input (the number of vertices in our graph). So the above algorithm is not polynomial time.

Task:[2] Design a poly-time algorithm that solves the H-path problem. Send it to me. I'll give you a 10% bonus in this course.

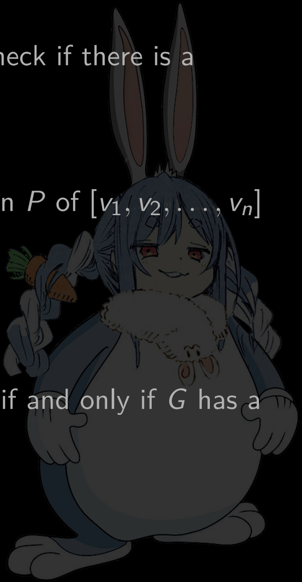
[2] Don't actually try this.

H-path

Here's a non-deterministic Turing machine M to check if there is a H-path, given a graph input G:

```
$M(G)$ :Let $v_1, v_2, \dots, v_n$ be the vertices of $G$
        Nondeterministically *choose* a permutation $P$ of $[v_1, v_2, \dots, v_n]$
        If $P$ is a valid path in $G$:
            Accept
        Reject
```

Task: Ask yourself, "Why does M accept input G if and only if G has a Hamiltonian path?"



H-path

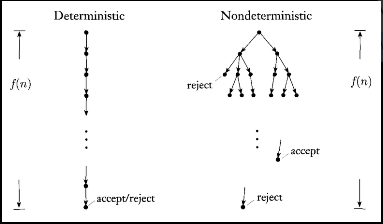
We've said that the **running time** of a deterministic TM M is the function $f(n)$ such that

$$f(n) = \max\{s : M(x) \text{ halts in exactly } s \text{ steps}, |x| = n\}$$

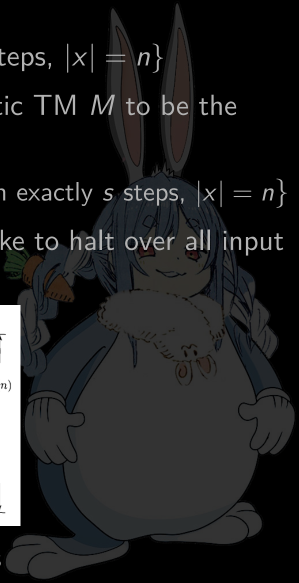
We'll define the **running time** of a non-deterministic TM M to be the function

$$f(n) = \max\{s : \text{for some execution path, } M(x) \text{ halts in exactly } s \text{ steps}, |x| = n\}$$

In other words, $f(n)$ is the maximum time it will take to halt over all input of length n and all execution paths.



pls dont sue me google images



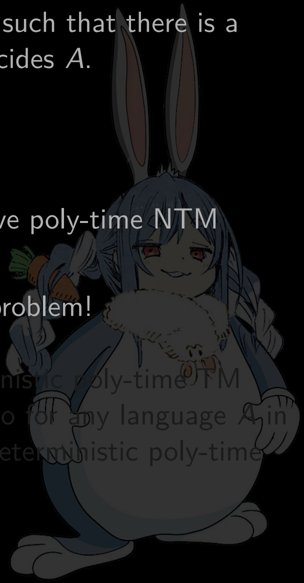
H-path

Define NTIME($f(n)$) to be the set of languages A such that there is a $O(f(n))$ non-deterministic Turing machine that decides A.
Define

$$\mathrm{NP} = \bigcup_{k=1}^{\infty} \mathrm{NTIME}(n^k).$$

In other words, NP is the set of languages that have poly-time NTM deciders.

Now you can tell your friends about the P vs NP problem!
Task: Show P $\subseteq$ NP.
Answer: Every poly-time TM is also a nondeterministic poly-time TM (except it only has one possible execution path). So for any language A in P, a poly-time decider for A would also be a non-deterministic poly-time decider for A, so $A \in$ NP.



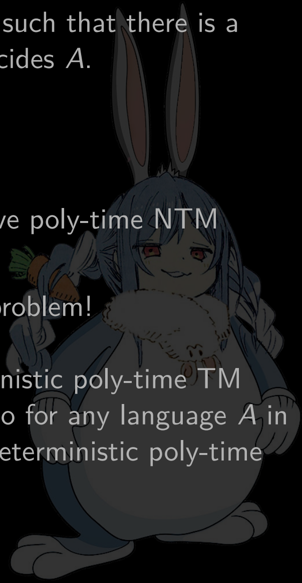
H-path

Define NTIME($f(n)$) to be the set of languages A such that there is a $O(f(n))$ non-deterministic Turing machine that decides A.
Define

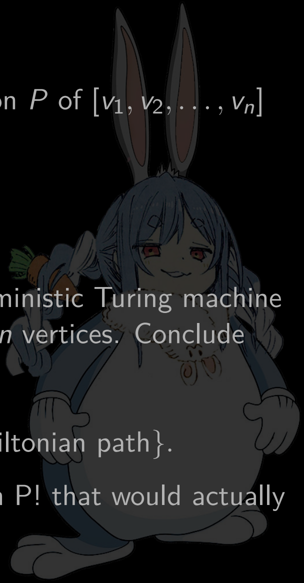
$$\mathrm{NP} = \bigcup_{k=1}^{\infty} \mathrm{NTIME}(n^k).$$

In other words, NP is the set of languages that have poly-time NTM deciders.

Now you can tell your friends about the P vs NP problem!
Task: Show $\mathrm{P} \subseteq \mathrm{NP}$.
Answer: Every poly-time TM is also a nondeterministic poly-time TM (except it only has one possible execution path). So for any language A in P, a poly-time decider for A would also be a non-deterministic poly-time decider for A, so $A \in \mathrm{NP}$.



H-path



```
$M(G)$ :Let $v_1, v_2, \ldots, v_n$ be the vertices of $G$
        Nondeterministically *choose* a permutation $P$ of $[v_1, v_2, \ldots, v_n]$
        If $P$ is a valid path in $G$:
            Accept
        Reject
```

Task: Convince yourself that the above non-deterministic Turing machine takes polynomial time to halt, given a graph G of n vertices. Conclude that the following language is in NP:

$$\text{HPATH} = \{G : G \text{ is a graph with a Hamiltonian path}\}.$$

(we actually don't know if the above language is in P! that would actually amount to proving P = NP.)

sushi juice, part 2.

Task: Guess what I've ate today.

Task: Guess what is in this "juice".

Task: Guess who was feeling sadistic while designing Assignment 4.

sushi juice, part 2.

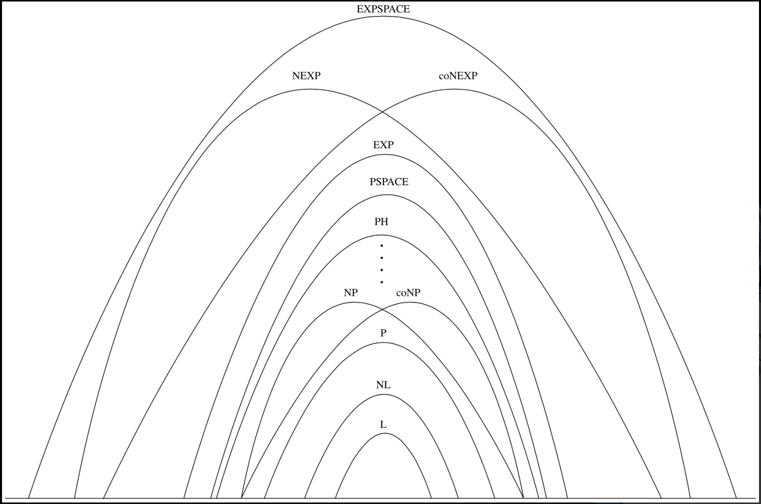
Task: Guess what I've ate today.
Task: Guess what is in this "juice".
Task: Guess who was feeling sadistic while designing Assignment 4.

sushi juice, part 2.

Task: Guess what I've ate today.
Task: Guess what is in this "juice".
Task: Guess who was feeling sadistic while designing Assignment 4.

here's a zoo.

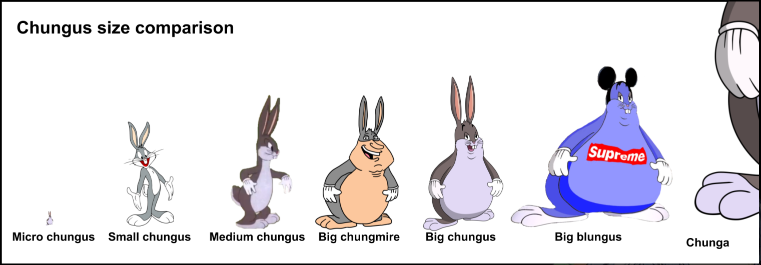
We'll call it the *computational complexity zoo*.



Don't worry, in this course you only need to know about P, NP, coNP, and possibly PSPACE.

here's another zoo.

We'll call it the *chungus zoo*.



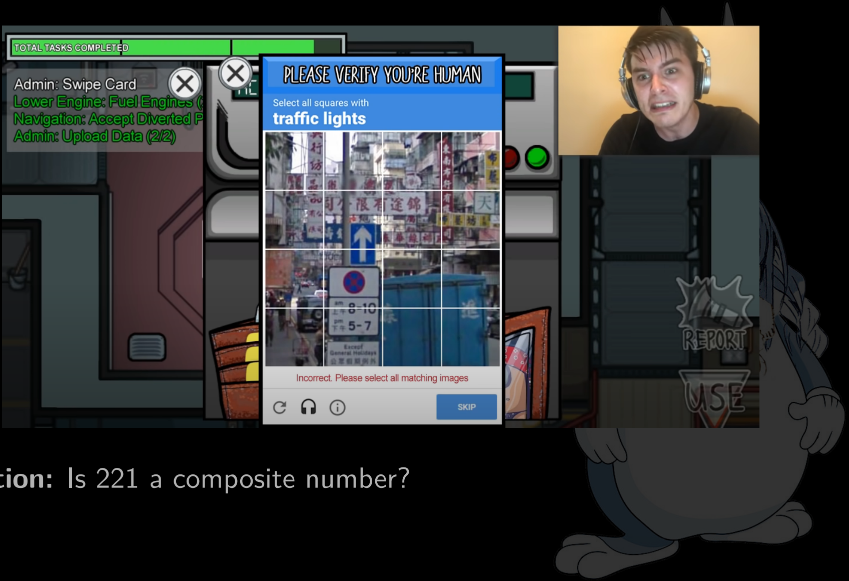
This is what your computer science education has come to.

Anyway, just like there are many different sizes of chungus, there are many different "computational complexity" classes. As you go up the "computational complexity scale", you get harder and harder problems. Here's all we know so far:

$$\text{anything practical} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{Decidable} \subseteq \text{Enumerable} \subseteq \Sigma_2 \subseteq \dots$$

(we don't know if $\text{P} \subsetneq \text{NP}$.)

verification


TOTAL TASKS COMPLETED
Admin: Swipe Card
Lower Engine: Fuel Engines (
Navigation: Accept Diverted P
Admin: Upload Data (2/2)
PLEASE VERIFY YOU'RE HUMAN
Select all squares with
traffic lights
Incorrect. Please select all matching images
SKIP
REPORT
USE

Question: Is 221 a composite number?

verification

Question: Is 221 a composite number?
Answer: Yes. $221 = 13 \cdot 17$. (calculator gang)

Question: Does 13 divide 221?
Answer: Yes.

Question: Which of the above two questions is easier to answer?
Answer: Probably the "does 13 divide 221" one.

verification

Question: Is 221 a composite number?
Answer: Yes. $221 = 13 \cdot 17$. (calculator gang)

Question: Does 13 divide 221?
Answer: Yes.

Question: Which of the above two questions is easier to answer?
Answer: Probably the "does 13 divide 221" one.

verification

Question: Is 221 a composite number?
Answer: Yes. $221 = 13 \cdot 17$. (calculator gang)

Question: Does 13 divide 221?
Answer: Yes.

Question: Which of the above two questions is easier to answer?
Answer: Probably the “does 13 divide 221” one.

verification

The point is, checking if a number is composite is hard. We can create a TM M that checks if a number is composite:

```
$M(x)$ :For all integers $1 < i < x$:
    If $i$ divides $x$:
        Accept.
    Reject.
```

We can be more efficient with a NTM M:

```
$M(x)$ :Nondeterministically choose an integer $1 < i < x$:
    If $i$ divides $x$:
        Accept.
    Reject.
```

verification

To check if a number x is composite, you're gonna have to go through all integers from 2 to $x-1$. But *verifying* that a number x is composite, given a potential factor is easier.

Recall the Hamiltonian path problem: Given a graph G, to check that it has a H-path would require going through all permutations of the vertices, while it is easier to verify that a given permutation of vertices defines a valid Hamiltonian path.

Given a language A, a **verifier** V of A is a Turing machine such that

$$A = \{w : V \text{ accepts } (w, c) \text{ for some string } c\}.$$

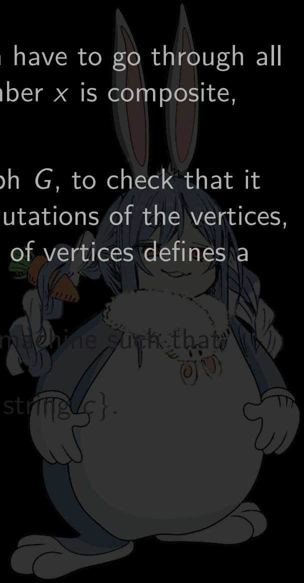
verification

To check if a number x is composite, you're gonna have to go through all integers from 2 to $x - 1$. But *verifying* that a number x is composite, given a potential factor is easier.

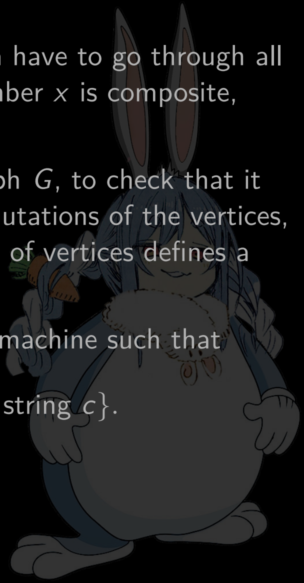
Recall the Hamiltonian path problem: Given a graph G, to check that it has a H-path would require going through all permutations of the vertices, while it is easier to verify that a given permutation of vertices defines a valid Hamiltonian path.

Given a language A, a **verifier** V of A is a Turing machine such that

$$A = \{w : V \text{ accepts } (w, c) \text{ for some string } c\}.$$



verification



To check if a number x is composite, you're gonna have to go through all integers from 2 to $x - 1$. But *verifying* that a number x is composite, given a potential factor is easier.

Recall the Hamiltonian path problem: Given a graph G, to check that it has a H-path would require going through all permutations of the vertices, while it is easier to verify that a given permutation of vertices defines a valid Hamiltonian path.

Given a language A, a **verifier** V of A is a Turing machine such that

$$A = \{w : V \text{ accepts } (w, c) \text{ for some string } c\}.$$

verification

Given a language A, a **verifier** V of A is a (deterministic) Turing machine such that

$$A = \{w : V \text{ accepts } (w, c) \text{ for some string } c\}.$$

For example, let

$$A = \{x : x \text{ is a binary string that corresponds to a composite number}\}.$$

Define V to be the following TM:

```
$V(x, y)$ :If $1 < y < x$ and $y$ divides $x$ :
    Accept
  Reject
```

Then $x \in A$ if and only if $V(x, y)$ accepts for some y. So

$$A = \{x : V \text{ accepts } (x, y) \text{ for some string } y\}$$

meaning V is a verifier for A.

Task: Make sense of this.

verification

Nice theorem! (Go read Sipser's book.)

Theorem

A language A is in NP if and only if it has a poly-time verifier.

(in fact, I think Sipser's book uses existence of poly-time verifier as the *definition* of NP, and then shows $\mathrm{NP} = \bigcup_{k=1}^{\infty} \mathrm{NTIME}(n^k)$.)

verification

fun fact! we can actually determine if a number is prime in poly-time. Search up the “AKS Primality Test”.

Not so fun fact: nobody actually uses the AKS primality test, because it’s like $O(n^{13})$ or something (which is still polynomial, but impractical!)

Fun reading: `https://en.wikipedia.org/wiki/Galactic_algorithm`