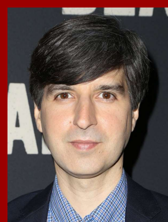
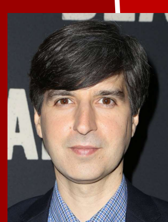
15-251: Great Theoretical Ideas in Computer Science
Fall 2018, Lecture 8

Time Complexity

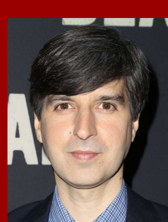


Dammit I'm mad!
– is a palindrome



In 1993, comedian Demetri Martin took
a math course at Yale called *Fractal Geometry.*

His final project: a 225-word palindromic poem.



In 1993, noted comedian Demetri Martin took a math course at Yale called *Fractal Geometry.*

His final project: a 225-word palindromic poem.

What does that have to do with fractals?

I don't know, it's a liberal arts school.

Dammit I'm mad, by Demetri Martin

Dammit I'm mad
Evil is a deed as I live.
God, am I reviled?
I rise, my bed on a sun, I melt.
To be not one man emanating is sad. I piss.
Alas it is so late. Who stops to help? Man, it is hot.

I'm in it.
I tell.
I am not a devil.
I level "Mad Dog".

Ah, say burning is as a deified gulp
in my halo of a mired rum tin.
I erase many men. Oh, to be man, a sin.
Is evil in a clam? In a trap?
No. It is open.
On it I was stuck.

Rats peed on hope.
Elsewhere dips a web.
Be still if I fill its ebb.
Ew, a spider ... eh?
We sleep.

Oh no!
Deep, stark cuts saw it in one position.
Part animal, can I live? Sin is a name.
Both, one ... my names are in it. Murder?
I'm a fool. A hymn I plug,
Deified as a sign in ruby ash - a goddam level I lived at.

On mail let it in. I'm it.
Oh, sit in ample hot spots.
Oh, wet!
A loss it is alas (sip). I'd assign it a name.
Name not one bottle minus an ode by me:
"Sir, I deliver. I'm a dog."
Evil is a deed as I live.
Dammit I'm mad.

That's nothing.

In 1986, one Lawrence Levine wrote an entire palindromic **novel**.

It had ~100,000 letters.

Dr. Awkward & Olson in Oslo
by Lawrence Levine

"Tacit, I hate gas (aroma of evil), masonry, tramps, a wasp martyr. Remote liberal ceding is idle – if... heh-heh," Sam X. Xmas murmured in an undertone to tow-trucker Edwards. "Alas. Simple – hot." To didos, no tracks, Ed decided. "Or – eh – trucks abob."

(…160 pages and 100,000 characters later…)

"Bob, ask Curt. He rode diced desk carton. So did Otto help Miss Alas draw Derek-cur. Two tote? Not red Nun. A nide. Rum. Rum Sam X. Xmas. Heh, heh. Field, I sign. I declare bile to merry tramps. A wasp martyr? No, Sam – live foam or a sage Tahiti Cat."

Suppose you are the proofreader.
You have to check if there's a mistake…

"Tacit, I hate gas (aroma of evil), masonry, tramps, a wasp martyr. Remote liberal ceding is idle – if... heh-heh," Sam X. Xmas murmured in an undertone to tow-trucker Edwards. "Alas. Simple – hot." To didos, no tracks, Ed decided. "Or – eh – trucks abob."

(…160 pages and 100,000 characters later…)

"Bob, ask Curt. He rode diced desk carton. So did Otto help Miss Alas draw Derek-cur. Two tote? Not red Nun. A nide. Rum. Rum Sam X. Xmas. Heh, heh. Field, I sign. I declare bile to merry tramps. A wasp martyr? No, Sam – live foam or a sage Tahiti Cat."

Want to solve the PALINDROME problem on an instance with n = 10^5 characters.

Today's lecture:
Defining, discussing, and debating
the words and ideas in the following sentence:

The intrinsic time complexity
of solving the PALINDROME problem is $\Theta(n)$.

Where we've been, where we're going

Lecture 1-2: Overview & Review

Lectures 3–5: Defining computation…

- What is a computational problem?
- What is an algorithm?
- *Computability*: Which problems *can* be solved by algorithms, and which can't.

Where we've been, where we're going

- *Computability*: Which problems *can* be solved by algorithms, and which can't.

The PALINDROME problem *cannot* be solved by a wimpy notion of algorithms (DFAs),

but *can* be solved by the full notion of algorithms (Turing Machines; equivalently, Python, C, SML…).

Where we've been, where we're going

- *Computability*: Which problems *can* be solved by algorithms, and which can't.

Once we know a problem *can* be solved, in *principle*, we usually ask about *practical* computability.

- *Complexity*: How *efficiently* various problems can be solved by algorithms.

Complexity: How *efficiently* various problems can be solved by algorithms.

Interesting Questions:

- Efficiency with respect to what?
  (**Time**, space/memory, parallelizability, …)
- What is the right model/level of abstraction?
- How to show efficient algorithms *don't* exist?
- "P vs. NP"…

Warning

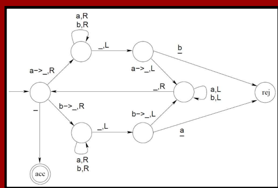
For computability, the model doesn't matter.
Computability is the same for TMs, C, Python, …

For complexity, the model **does** matter.
Not *too* much, but somewhat.

Today:

8 Great Ideas
in Theoretical Computer Science

Running time of deciding PALINDROME



*I stole this picture from the Internet. It doesn't even decide PALINDROME, it decides $\{ww^R : w \text{ in } \{a,b\}^*\}$.

How many steps does it take to decide if input x is in language PALINDROME?

Depends on the length of x!

Great Idea #1:

Measure running time as a function of the input length.

Instance/input length

Usually denoted **n**.

PALINDROME: Input is a string x.

n = # characters in x.

Instance/input length

Usually denoted **n**.

PRIMALITY: Input is a number $B \in \mathbb{N}^+$.

n depends on choice of encoding.

The default is binary (base 2).

Thus **n** = # binary digits = $\lceil \log_2(B+1) \rceil$

Sometimes we might sloppily say

"# of digits", and "log(B)".

Instance/input length

Usually denoted **n**.

PRIMALITY: Input is a number $B \in \mathbb{N}^+$.

$$n \neq B$$

This would mean encoding numbers in unary, which is a horrible idea.

Instance/input length

Usually denoted **n**.

MULTIPLICATION: Input is pair of number, (B_1, B_2).

n = $\lceil \log_2(B_1+1) \rceil + \lceil \log_2(B_2+1) \rceil$

+ 1 (for the delimeter)

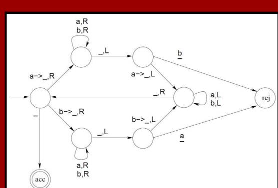
Instance/input length

Usually denoted **n**.

Warning: Sometimes you'll see it specified that **n** is something else.

E.g., for the SORTING problem, it is traditional for **n** to denote the number of items to be sorted (as opposed to total # of input bits).

Running time of deciding PALINDROME

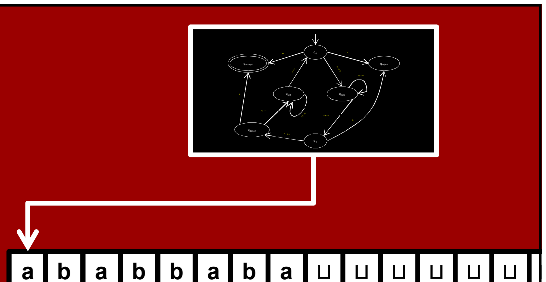


*I stole this picture from the Internet. It doesn't even decide PALINDROME, it decides $\{ww^R : w \text{ in } \{a,b\}^*\}$.

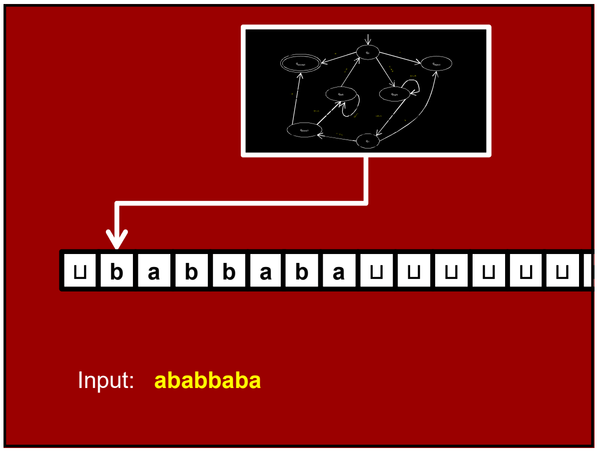
Number of steps to decide if x ∈ PALINDROME…

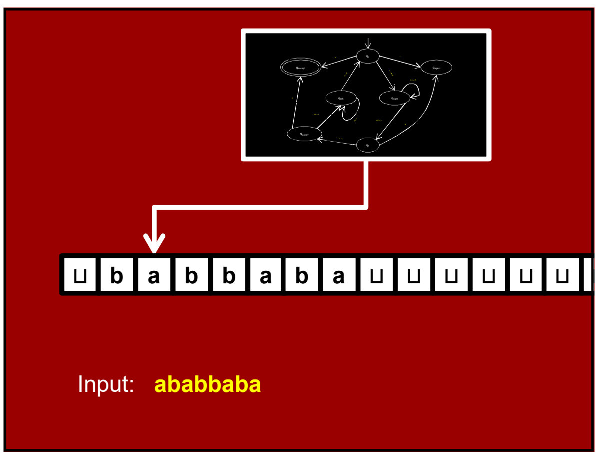
Depends on n, the length of x.

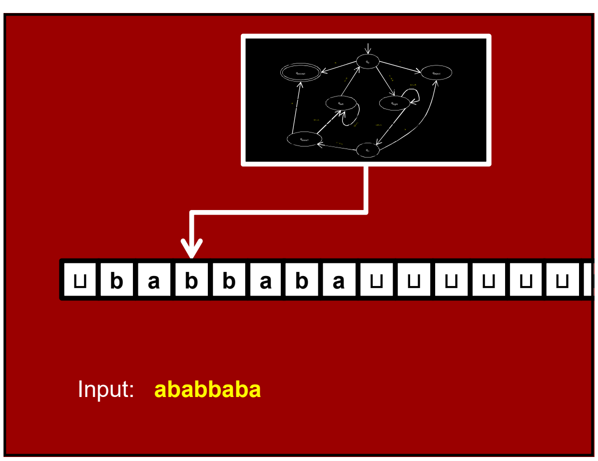
Also depends on x itself!

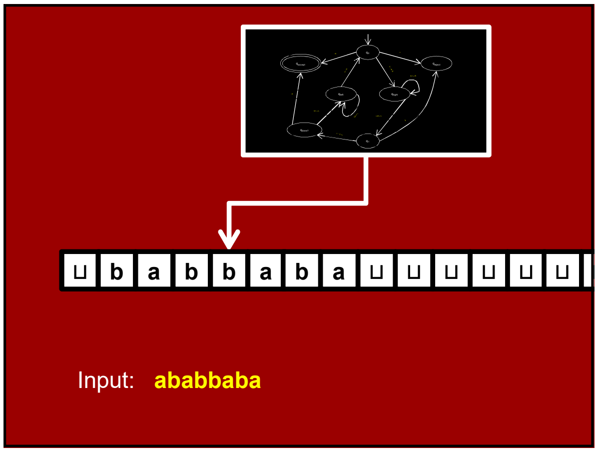


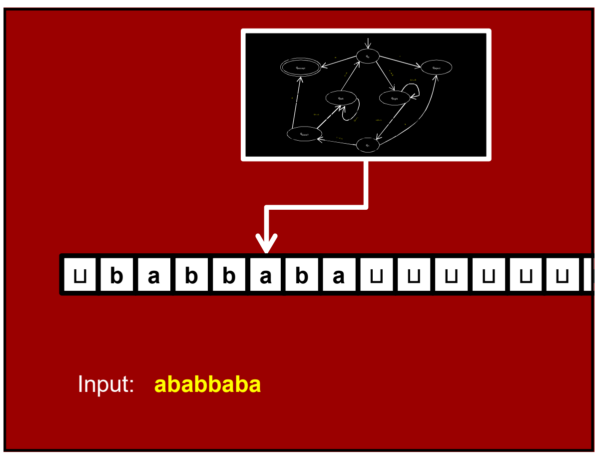
Input: **ababbaba**

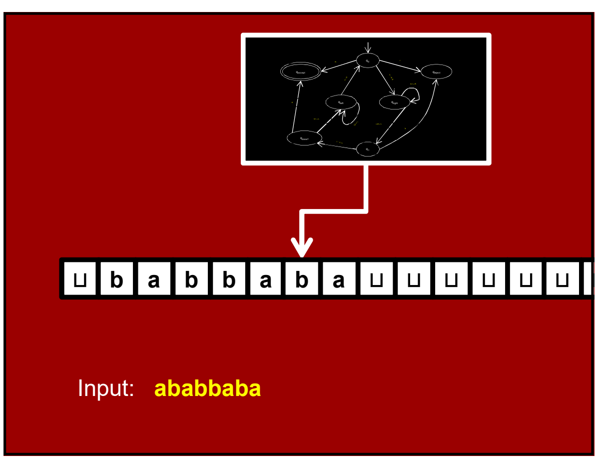

⊔ b a b b a b a ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

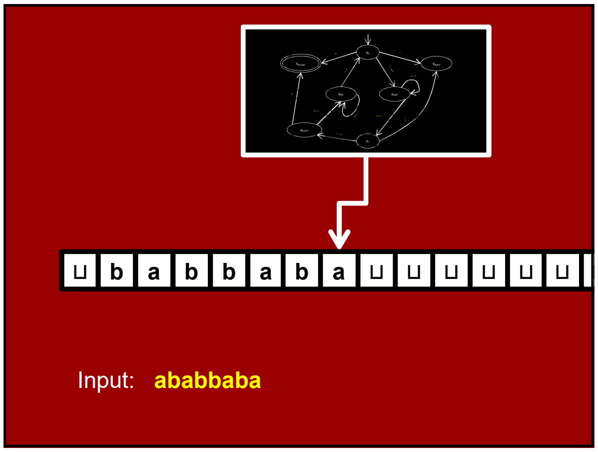

⊔ b a b b a b a ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

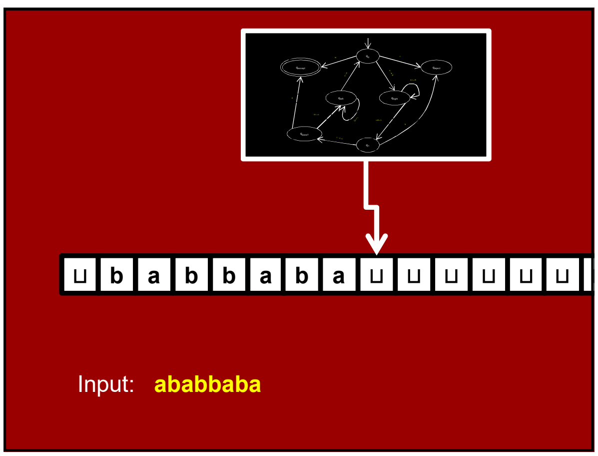

⊔ b a b b a b a ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

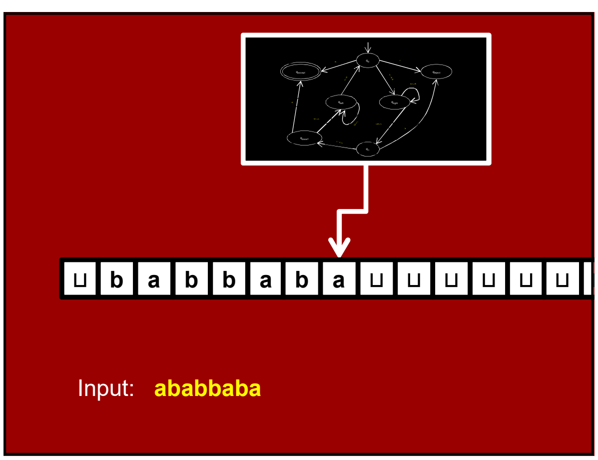

⊔ b a b b a b a ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

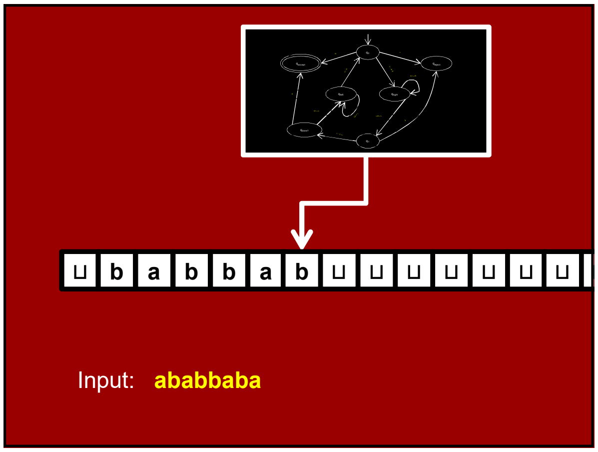

⊔ b a b b a b a ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

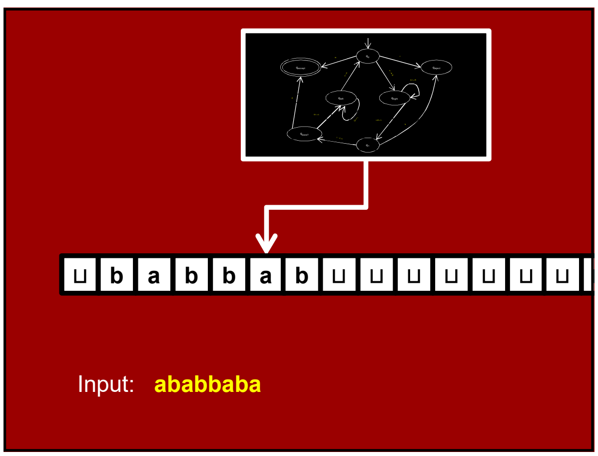

⊔ b a b b a b a ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

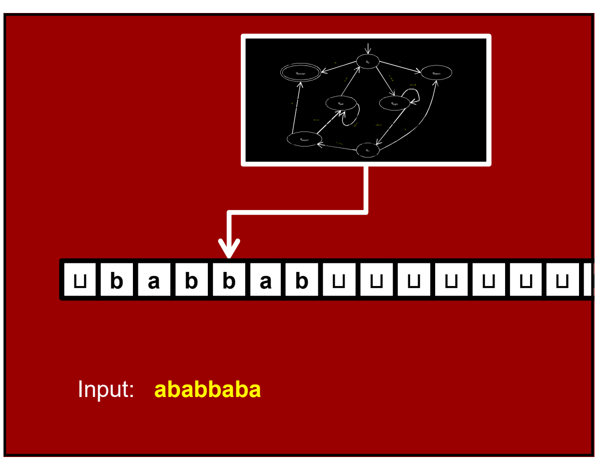

⊔ b a b b a b a ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

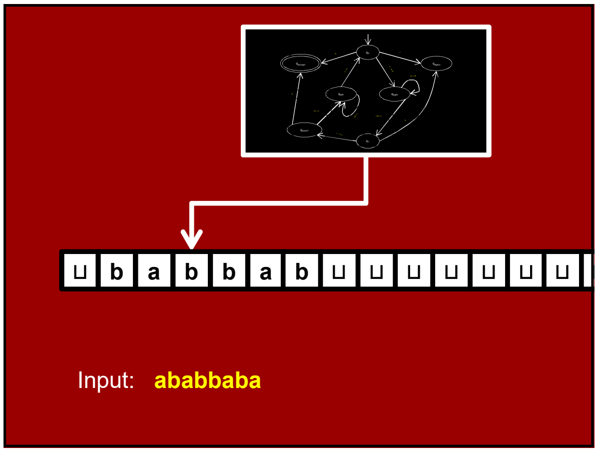

⊔ b a b b a b a ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

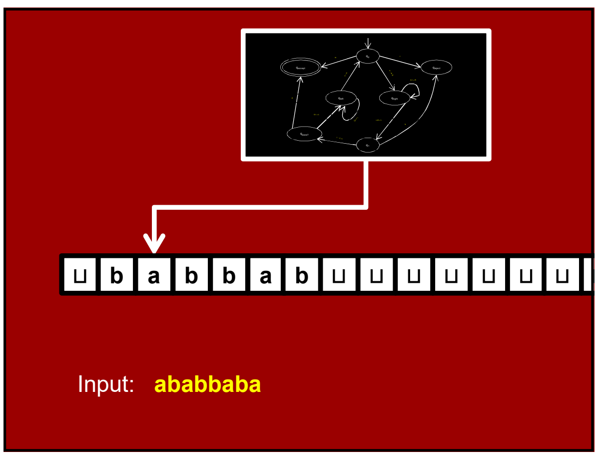

⊔ b a b b a b a ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

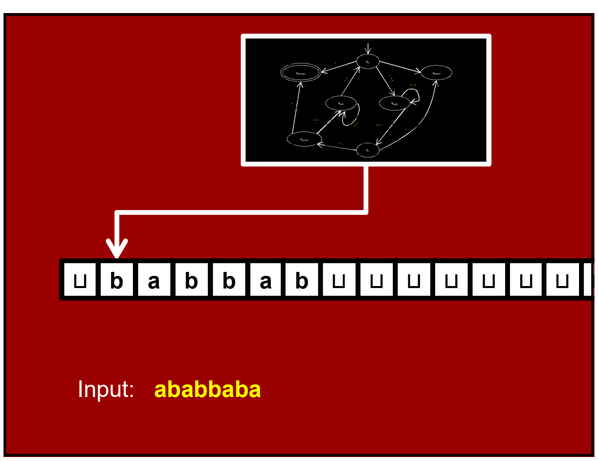

⊔ b a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

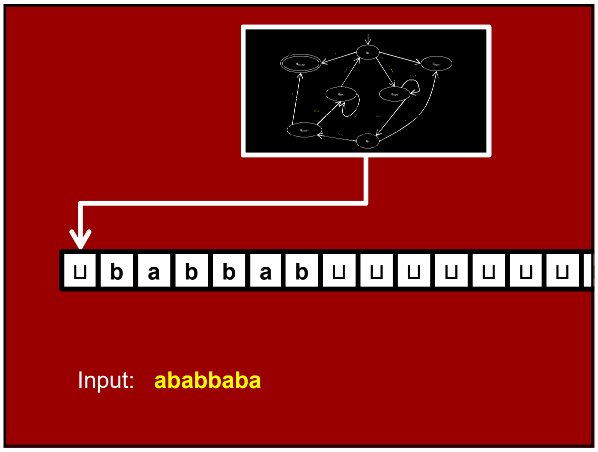

⊔ b a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

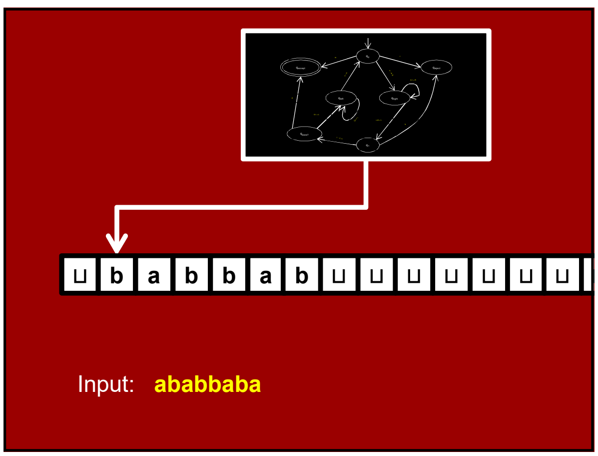

⊔ b a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

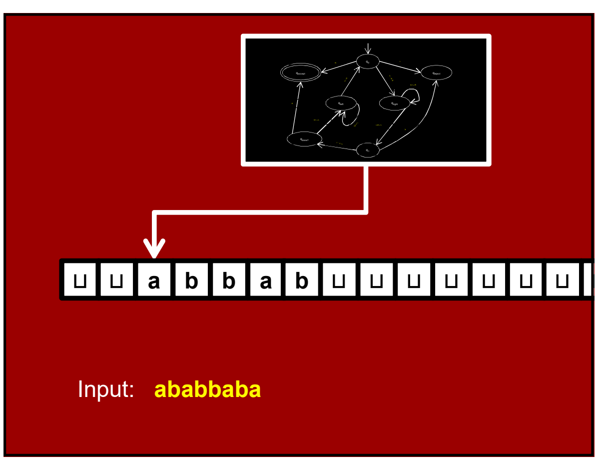

⊔ b a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

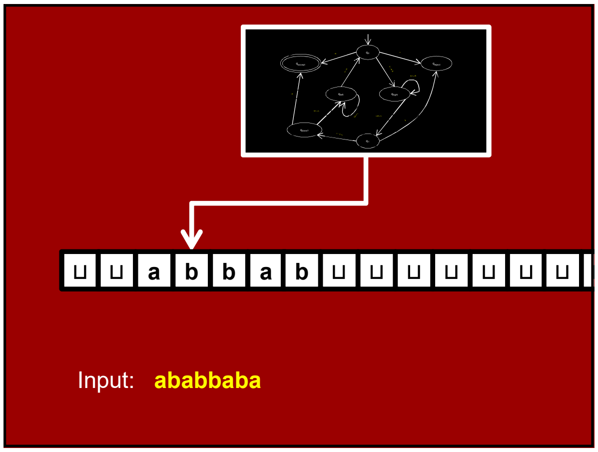

⊔ b a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

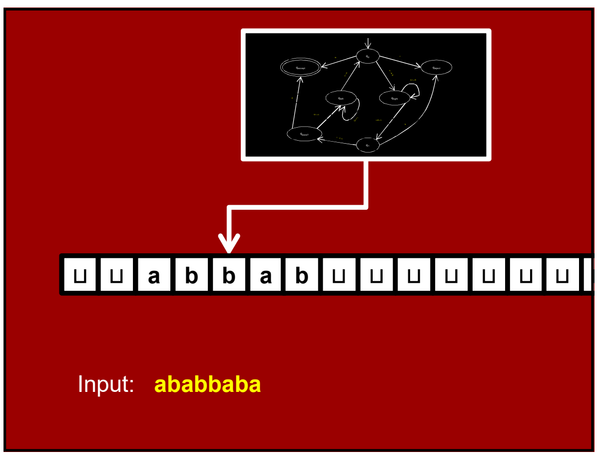

⊔ b a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

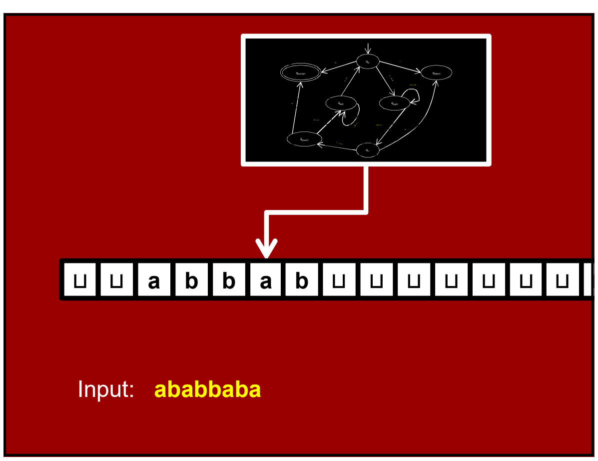

⊔ b a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

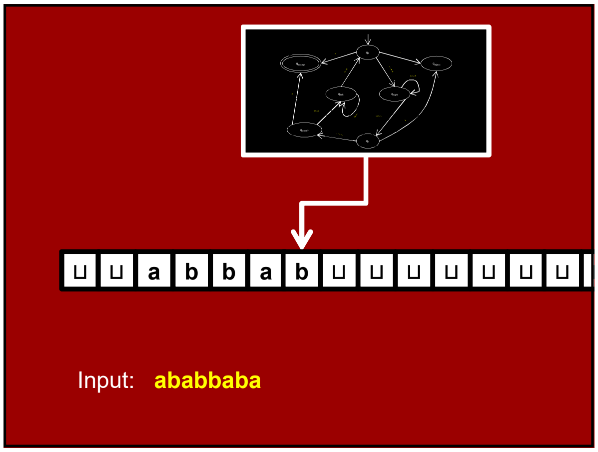

⊔ b a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

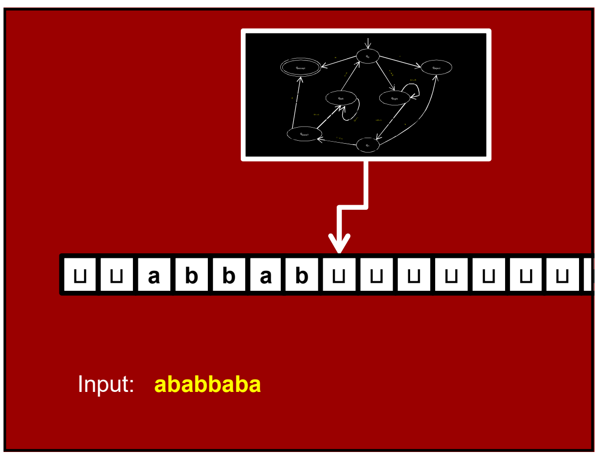

⊔ ⊔ a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

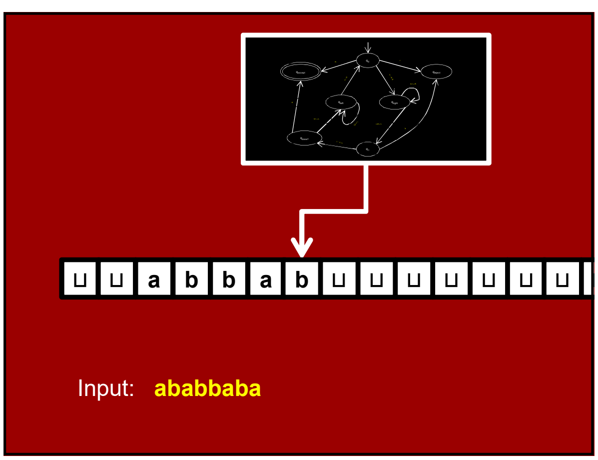

⊔ ⊔ a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba


⊔ ⊔ a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba


⊔ ⊔ a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba


⊔ ⊔ a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba


⊔ ⊔ a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba


⊔ ⊔ a b b a b ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
Input: ababbaba

Running time of deciding PALINDROME

Number of steps to decide if $x \in$ PALINDROME…

Depends on n, the length of x.

If x really is a palindrome, # of TM steps is:

$(n+1) + n + (n-1) + \cdots 3 + 2 + 1$

$= \frac{(n+1)(n+2)}{2} = \frac{1}{2}n^2 + \frac{3}{2}n + 1$

If x isn't a palindrome, it depends.
Could take as few as $n+1$ steps.

Great Idea #2:

Measure running time as a
worst-case
function of the input length.

Defining running time

The running time of algorithm A is a function $T_A : \mathbb{N} \to \mathbb{N}$, defined by

$$T_A(n) = \max_{\substack{\text{instances } x \\ \text{of length } n}} \{\# \text{ steps } A \text{ takes on } x\}$$

(worst case)

(When A is clear, we often just write $T(n)$.)

Defining running time

The running time of algorithm A is a function $T_A : \mathbb{N} \to \mathbb{N}$, defined by

$$T_A(n) = \max_{\substack{\text{instances } x \\ \text{of length } n}} \{\#\text{ steps A takes on } x\}$$

E.g., our PALINDROME TM had running time…

$$T(n) = \tfrac{1}{2}n^2 + \tfrac{3}{2}n + 1$$

Why worst case?

Well, we're not dogmatic about it.

Average (random) case, "typical" case, "smoothed analysis", all interesting too.

Pros of worst-case analysis:

- An ironclad guarantee.
- Matches our worst-case notion of an algorithm solving a problem.
- Hard to define what a 'typical' instance is.
- Random inputs are often not representative of typical inputs.
- Most straightforward way to do analysis.

Great Idea #3:

When it comes to running time, focus on the "big picture": how it **scales** as a function of n.

Our Palindrome TM had running time

$T(n) = \frac{1}{2}n^2 + \frac{3}{2}n + 1$

Our Palindrome TM had running time

$T(n) = \frac{1}{2}n^2 + \frac{3}{2}n + 1$

- Analogous to "too many significant figures"
- We'll soon study algorithms at a higher level (like, in C, or pseudocode), where it's not even exactly clear what counts as "1" time step
- Even for slightly more complicated algorithms, it's nearly impossible to calculate so precisely

Our Palindrome TM had running time

$T(n) = \frac{1}{2}n^2 + \frac{3}{2}n + 1$

We want to use the right level of abstraction!

The key takeaway of this T(n):
it's "quadratic"; that is, proportional to n^2.

This leads us to…

Great Idea #4:

Big-O notation

INFORMAL Definition

"As n gets large, T(n) is proportional to n^2." — T(n) is $\Theta(n^2)$

"As n gets large, T(n) is at most proportional to n^2." — T(n) is $O(n^2)$

"As n gets large, T(n) is at least proportional to n^2." — T(n) is $\Omega(n^2)$

INFORMAL Definition

roughly	$=$	$\Theta(\cdot)$
roughly	$\leq$	$O(\cdot)$
roughly	$\geq$	$\Omega(\cdot)$

Examples

roughly $=$ $\Theta(\cdot)$

roughly $\leq$ $O(\cdot)$

roughly $\geq$ $\Omega(\cdot)$

Examples

$\frac{1}{2}n^2 + \frac{3}{2}n + 1$ is $\Theta(n^2)$

roughly $\leq$ $O(\cdot)$

roughly $\geq$ $\Omega(\cdot)$

Examples

$\frac{1}{2}n^2 + \frac{3}{2}n + 1$ is $\Theta(n^2)$

$\frac{1}{2}n^2 + \frac{3}{2}n + 1$ is $O(n^3)$
is $O(n^2)$, too

roughly $\geq$ $\Omega(\cdot)$

Examples

$\frac{1}{2}n^2 + \frac{3}{2}n + 1$ is $\Theta(n^2)$

$\frac{1}{2}n^2 + \frac{3}{2}n + 1$ is $O(n^3)$
is $O(n^2)$, too

$\frac{1}{2}n^2 + \frac{3}{2}n + 1$ is $\Omega(n)$
is $\Omega(n^2)$, too

Examples

$6n^2 - 2n + 5$ is $\Theta(n^2)$
is not $\Theta(n^3)$

$2n^2 - 11$ is $O(n^3)$
is $O(n^2)$, too
is not $O(n)$

$1000n^2$ is $\Omega(n)$
is $\Omega(n^2)$, too

Formal definition of $O(n^2)$

Definition: $T(n)$ is $O(n^2)$ if and only if

∃ positive real C,
∃ positive real n_0,
such that $\forall\, n \geq n_0$ it holds that $T(n) \leq Cn^2$.

"Once n is large enough…

…T(n) is at most a constant factor times n^2."

Definition: T(n) is $O(n^2)$ if and only if

∃ positive real C,
∃ positive real n_0,
such that $\forall\, n \geq n_0$ it holds that $T(n) \leq Cn^2$.

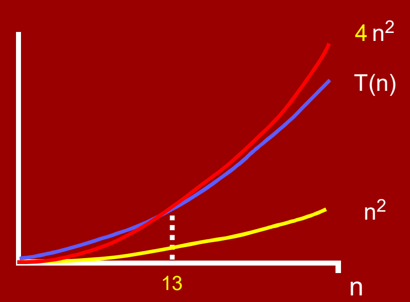
Example: $T(n) = 3n^2 + 10n + 30$ is $O(n^2)$

Why? Take $C = 4$. Take $n_0 = 13$.

Now if $n \geq 13$, then $10n + 30 \leq 10n + 3n$
$= 13n \leq n^2$

and so $T(n) = 3n^2 + 10n + 30 \leq 3n^2 + n^2 = 4n^2$.

Example: $T(n) = 3n^2 + 10n + 30$ is $O(n^2)$



T(n) is $O(n^2)$ because $T(n) \leq 4n^2$ for $n \geq 13$

Formal definition of $O(n^3)$

Definition: T(n) is $O(n^3)$ if and only if

∃ positive real C,
∃ positive real n_0,
such that $\forall\, n \geq n_0$ it holds that $T(n) \leq Cn^3$.

"Once n is large enough…

…T(n) is at most a constant factor times n^3."

Formal definition of O(g(n))

Definition: T(n) is O(g(n)) if and only if

∃ positive real C,

∃ positive real n_0,

such that ∀ n ≥ n_0 it holds that $T(n) \leq C \cdot g(n)$

"Once n is large enough…

…T(n) is at most a constant factor times g(n)."

Formal definition of $\Omega(n^2)$

Definition: T(n) is $\Omega(n^2)$ if and only if

∃ positive real c > 0,

∃ positive real n_0,

such that ∀ n ≥ n_0 it holds that $T(n) \geq cn^2$

"Once n is large enough…

…T(n) is at least a constant factor times n^2."

Formal definition of Ω(g(n))

Definition: T(n) is Ω(g(n)) if and only if

∃ positive real c > 0,

∃ positive real n_0,

such that ∀ n ≥ n_0 it holds that $T(n) \geq c \cdot g(n)$

"Once n is large enough…

…T(n) is at least a constant factor times g(n)."

Formal definition of Θ(g(n))

Definition: T(n) is Θ(g(n)) if and only if

T(n) is O(g(n))

and

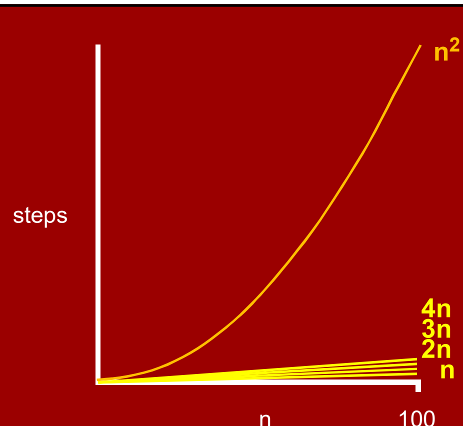
T(n) is Ω(g(n)).

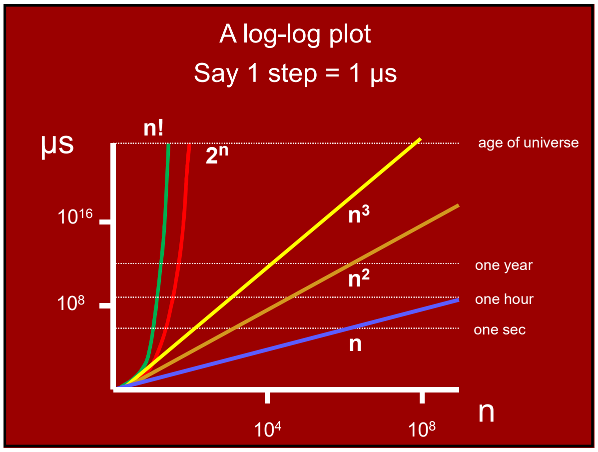
"Once n is large enough…

…T(n) is within a constant factor of g(n)."

Common run-time scaling

Θ(log n)	"logarithmic"	2× input size ⇒ run time +1
Θ(n)	"linear"	doubling the input size ⇒ doubling the running time
$\Theta(n^2)$	"quadratic"	2× input size ⇒ 4× run time
$\Theta(n^3)$	"cubic"	2× input size ⇒ 8× run time
$\Theta(n^c)$	"polynomial"	2× input size ⇒ constant × run time
$\Theta(2^n)$	"exponential"	2× input size ⇒ run time **squares**





Some functions, each O(·) of the next

1	n	2^n
$\log(\log^* n)$	$n \log n$	3^n
$\log^* n$	n^2	$n!$
$\log \log n$	n^3	n^n
$\log n$	n^{100}	2^{2^n}
$\sqrt{n}$	$n^{\log n}$	$\underbrace{2^{2^{2^{\cdots^2}}}}_{n \text{ times}}$
$n / \log n$		

inverse function of ($\log^* n$ → $\underbrace{2^{2^{2^{\cdots^2}}}}_{n \text{ times}}$)

Some functions, each O(·) of the next

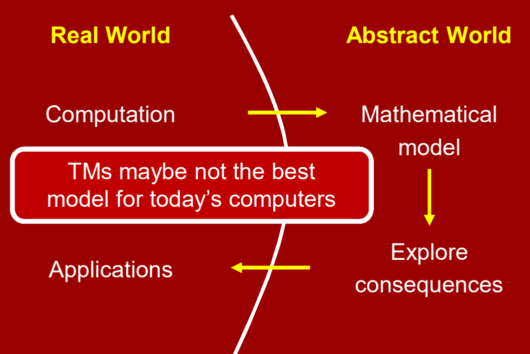
fastest known alg. for MULTIPLICATION → $n \cdot (\log n) \cdot 8^{\log^* n}$ (between n and $n \log n$)

	n	2^n
$\log(\log^* n)$	$n \log n$	3^n
$\log^* n$	n^2	$n!$
$\log \log n$	n^3	n^n
$\log n$	n^{100}	2^{2^n}
$\sqrt{n}$	$n^{\log n}$	$\underbrace{2^{2^{2^{\cdots^2}}}}_{n \text{ times}}$
$n / \log n$		

Great Idea #5:

The computation model **does** make a difference when counting running time.

A picture from Lecture 1



Suppose you are the proofreader. You have to check if there's a mistake…

"Tacit, I hate gas (aroma of evil), masonry, tramps, a wasp martyr. Remote liberal ceding is idle – if... heh-heh," Sam X. Xmas murmured in an undertone to tow-trucker Edwards. "Alas. Simple – hot." To didos, no tracks, Ed decided. "Or – eh – trucks abob."

(…160 pages…)

"Bob, ask Curt. He rode diced desk carton. So did Otto help Miss Alas draw Derek-cur. Two tote? Not red Nun. A nide. Rum. Rum Sam X. Xmas. Heh, heh. Field, I sign. I declare bile to merry tramps. A wasp martyr? No, Sam – live foam or a sage Tahiti Cat."

TwoFingersPalindromeTest(S,n)

```
               // ACCEPT iff string
               // S[1]...S[n] is a palindrome
  lo ← 1
  hi ← n
  while (lo < hi)
    if S[lo] ≠ S[hi] then REJECT
    lo ← lo + 1
    hi ← hi − 1
  end while
  ACCEPT
```

Poll: what *should* the running time be?

TwoFingersPalindromeTest(S,n)

```
               // ACCEPT iff string
               // S[1]...S[n] is a palindrome
  lo ← 1
  hi ← n
  while (lo < hi)
    if S[lo] ≠ S[hi] then REJECT
    lo ← lo + 1
    hi ← hi − 1
  end while
  ACCEPT
```

can you really access far-apart memory cells in “1” step?

TwoFingersPalindromeTest(S,n)

```
               // ACCEPT iff string
               // S[1]...S[n] is a palindrome
  lo ← 1
  hi ← n
  while (lo < hi)
    if S[lo] ≠ S[hi] then REJECT
    lo ← lo + 1
    hi ← hi − 1
  end while
  ACCEPT
```

storing `hi` requires $\log_2 n$ bits;

does decrementing take 1 step? $\Theta(\log n)$ steps?

```
TwoFingersPalindromeTest(S,n)
               // ACCEPT iff string
               // S[1]...S[n] is a palindrome
  lo ← 1
  hi ← n
  while (lo < hi)
    if S[lo] ≠ S[hi] then REJECT
    lo ← lo + 1
    hi ← hi - 1
  end while
  ACCEPT
```

```
TwoFingersPalindromeTest(S,n)
               // ACCEPT iff string
               // S[1]...S[n] is a palindrome
  lo ← 1
  hi ← n
  while (lo < hi)
    if S[lo] ≠ S[hi] then REJECT
    lo ← lo + 1
    hi ← hi - 1
  end while
  ACCEPT
```

This "feels like" it has running time $\Theta(n)$…

```
TwoFingersPalindromeTest(S,n)
               // ACCEPT iff string
               // S[1]...S[n] is a palindrome
  lo ← 1
  hi ← n
  while (lo < hi)
    if S[lo] ≠ S[hi] then REJECT
    lo ← lo + 1
    hi ← hi - 1
  end while
  ACCEPT
```

Next lecture: We'll discuss a model where this has running time $\Theta(n)$.

Today: Just want to point these issues out…

Great Idea #6:

Intrinsic complexity
& beating brute force

Intrinsic complexity

Given a *problem*, e.g., PALINDROME,
we can ask about its **intrinsic complexity**:

How fast is its **fastest** algorithm?

(Up to $\Theta(\cdot)$, and fixing the model of computation!)

PALINDROME:

We know an $O(n)$ algorithm, `TwoFingers`.
Could there be a faster one? E.g., $O(\sqrt{n})$?

Theorem:

Any alg. solving PALINDROME uses $\geq n-1$ steps.

Proof sketch: Suppose algorithm A solves it using $\leq n-2$ steps.
Let x be the string aaaa···a (n times), which is a palindrome.
When A runs with input x there must be distinct $1 \leq j_1, j_2 \leq n$
such that A never reads $I[j_1]$ or $I[j_2]$. (Why?)
Let x′ be the same as x except that $x[j_1]=b$ and $x[j_2]=c$.
When A runs on x′ it has same behavior as when it runs on x. (Why?)
But A accepts x and rejects x′ (why?), a contradiction.

PALINDROME:

We know an O(n) algorithm, `TwoFingers`.

Could there be a faster one? E.g., $O(\sqrt{n})$?

Theorem:

Any alg. solving PALINDROME uses ≥ n−1 steps.

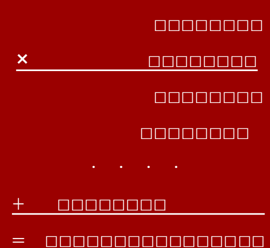
Conclusion:

The intrinsic time complexity of PALINDROME is **linear**;

$\Theta(n)$ time is necessary and sufficient.

MULTIPLICATION:

In grade school you learn an $O(n^2)$ algorithm.



MULTIPLICATION:

In grade school you learn an $O(n^2)$ algorithm.

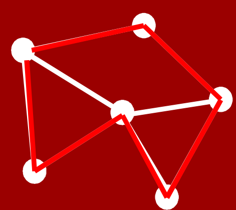
Easy to show ≥ n steps are required: you at least have to write down the answer!

Is there a faster algorithm?

Yes! A much faster one, we'll see next time...

HAMILTONIAN-CYCLE:

Instance: A connected graph.
Notation: Let 'n' = # of vertices.
Solution: Yes/No: Is there a "tour" visiting each vertex exactly once?



HAMILTONIAN-CYCLE:

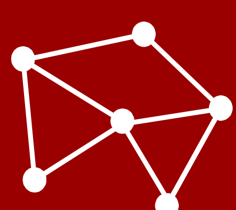
Brute-force alg: Try all tours
≈ n! steps

[Held-Karp'70]: Dynamic programming
≈ 2^n steps

[Björklund'10]: Clever algebraic brute-force
≈ 1.657^n steps

EULERIAN-CYCLE:

Instance: A connected graph.
Notation: Let 'n' = # of vertices.
Solution: Yes/No: Is there a "tour" visiting each **edge** exactly once?



EULERIAN-CYCLE:

Algorithm $\mathbb{E}$:
Check if every vertex is attached to an *even* number of other vertices.
If so, output Yes. Else output No.

Euler's Theorem: Alg. $\mathbb{E}$ solves EULERIAN-CYCLE.

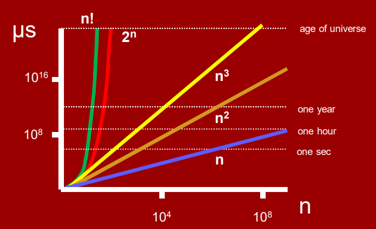
Time: $T_{\mathbb{E}}(n) = O(n^2)$.

Great Idea #7:

Polynomial time.

I.e., time $O(n^c)$ for some constant c.

There is something truly **magical** about the notion of polynomial time.



There is an enormous efficiency chasm between polynomial and exponential time.

HAMILTONIAN-CYCLE:

Seems to require exponential time.
We have no 'good' understanding of which graphs have Hamiltonian cycles.

EULERIAN-CYCLE:

Polynomial time.
Euler's Theorem 'explains' Eulerian cycles.

There is an enormous efficiency chasm between polynomial and exponential time.

HAMILTONIAN-CYCLE:

Seems to require exponential time.
We have no 'good' understanding of which graphs have Hamiltonian cycles.

EULERIAN-CYCLE:

Polynomial time.
Euler's Theorem 'explains' Eulerian cycles.

There is an enormous **understanding chasm** between polynomial and exponential time.

Common progress paradigm for a problem

Brute force algorithm: Exponential time

what we care about most in 15-251 ↓ usually the 'magic' happens here

Algorithmic breakthrough: Polynomial time

what we care about more in 15-451 ↓

Blood, sweat, and tears: Nearly linear time

Does "polynomial time" imply "efficient"?

$\Theta(n)$	Efficient (unless the constant is insane…)
$\Theta(n \log n)$	Efficient.
$\Theta(n^2)$	Kind of efficient.
$\Theta(n^3)$	Barely efficient?
$\Theta(n^{100})$	Not efficient. But it almost never arises.

(Efficient / Kind of efficient:) Distinction depends on your exact model.

It's a **negatable** benchmark:

"**Not** polynomial time" pretty much implies "**not** efficient".

Polynomial time

50 years of computer science experience shows it's a very compelling definition:

- A *necessary* first step towards truly efficient algorithms, associated with "beating brute-force"
- Very robust to notion of what is an elementary step.
- Easy to work with: Plug a poly-time subroutine into a poly-time algorithm: still poly-time.
- Empirically, it seems that most natural problems with poly-time algorithms also have efficient-in-practice algorithms.

Great Idea #8:
The Strong Church−Turing Thesis

All 'reasonable' models of step-counting for 'algorithms' are polynomially equivalent.

The Strong Church−Turing Thesis

Suggested by decades of computer science experience.

E.g., it's not hard to show that Turing Machines can simulate "C / python-style" algorithms/step-counting with at most polynomial slowdown, & vice versa.

The Strong Church−Turing Thesis

Challenger from the 1970s:

Randomized computation.

Give the model the ability to generate random bits.

In light of research from 1980s…

We believe (can't prove) that the Strong Church−Turing Thesis holds true even with randomized computation.

The Strong Church–Turing Thesis

Challenger from the 1980s:

Quantum computation (Lecture 24).

Allow “qubits” in quantum superposition.

In light of research from 1990s…

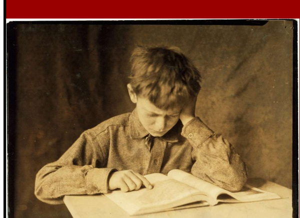
We believe (can’t prove) that the Strong Church–Turing Thesis **is not true**.

Great Idea #8: The Strong Church–Turing Thesis

All ‘reasonable’ models of step-counting for ‘algorithms’ are polynomially equivalent.

Sometimes Great Ideas are wrong!
Challenge all ideas!

Study Guide



Definitions:

- Running time complexity.
- Big O, Θ, Ω

Practice:

- Analyzing time complexity of TMs
- Proving T(n) is O(g(n)) or Θ(g(n)), Ω(g(n))
- Proving T(n) is **not** O(g(n)),