Great Theoretical Ideas In Computer Science

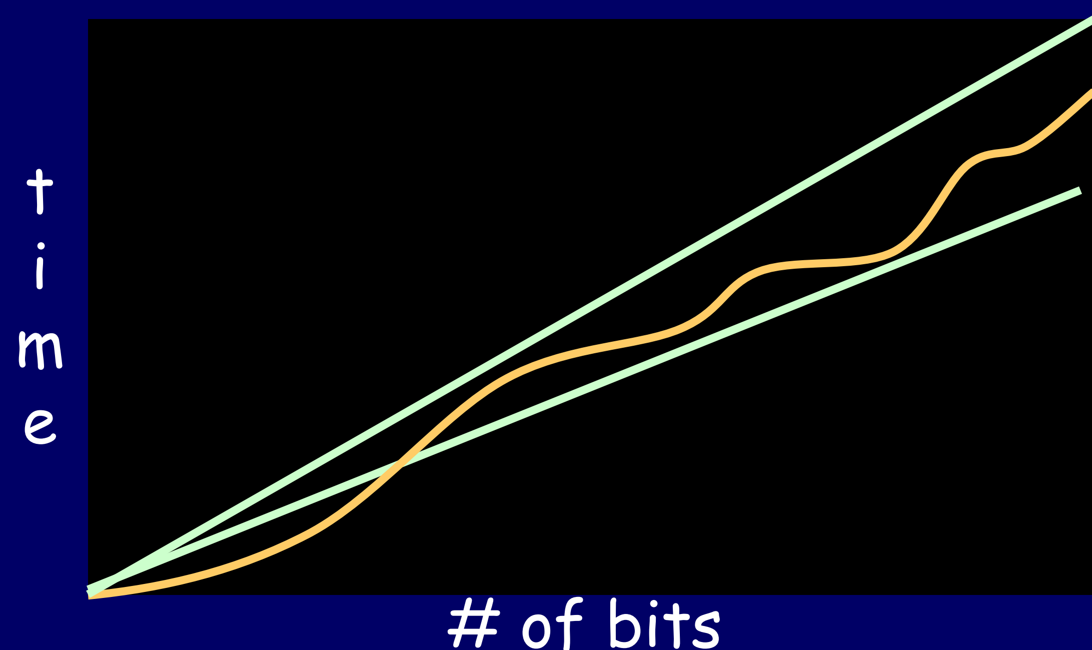
John Lafferty

CS 15-251 Fall 2006

Lecture 23 November 14, 2006

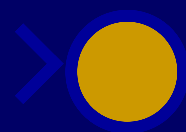
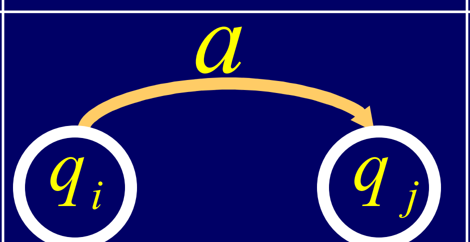
Carnegie Mellon University

On Time Versus Input Size

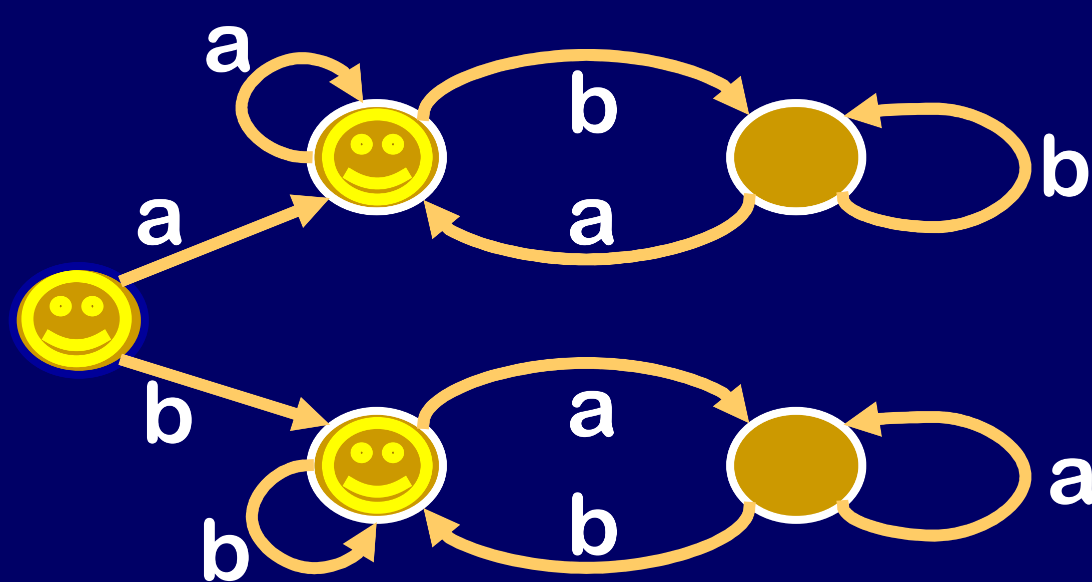


Recall From Last Time...

Finite Automaton

Finite set of states		$Q = \{q_o, q_1, q_2, \ldots, q_k\}$
A start state		q_o
A set of accepting states		$F = \left\{q_{i_1}, q_{i_2}, \ldots, q_{i_r}\right\}$
A finite alphabet	a b # x 1	Σ
State transition instructions		$\partial : Q \times \Sigma \rightarrow Q$ $\partial(q_i, a) = q_j$

Remember "ABA"?



ABA accepts only the strings with an equal number of *ab*'s and *ba*'s!

Consider the language

$$a^n b^n = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$$

i.e., a bunch of *a*'s
followed by an equal
number of *b*'s

No finite automaton accepts this language.

Theorem: $a^n b^n$ is not regular.

Proof: Assume that it is. Then $\exists$ M with k states that accepts it.

For each $0 \leq i \leq k$, let S_i be the state M is in after reading a^i.

$\exists i,j \leq k$ s.t. $S_i = S_j$, but $i \neq j$

M will do the same thing on $a^i b^i$ and $a^j b^i$.

But a valid M must reject $a^j b^i$ and accept $a^i b^i$.

$\Rightarrow\Leftarrow$

MORAL:

Finite automata can't count.

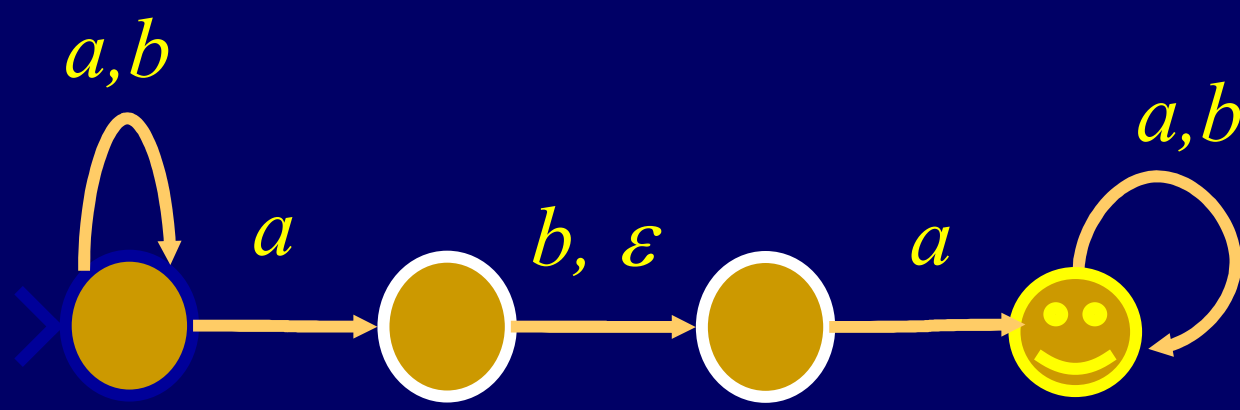
But be careful: Automata can indeed do some kinds of *implicit* counting.

Nondeterministic Finite State Automata

Two differences:

1. Allow “epsilon transitions” where a state transition is made, but no input symbol is read.

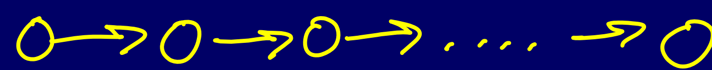
2. Allow more than one transition from a state matching the same input symbol.



$L =$ all strings containing aba or aa

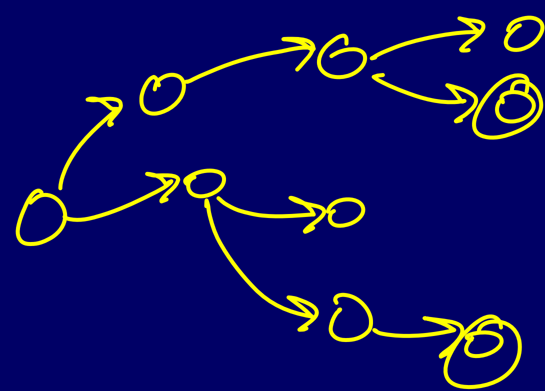
Running an NFA: Parallel Programming

DFA:



(n steps)

NFA:

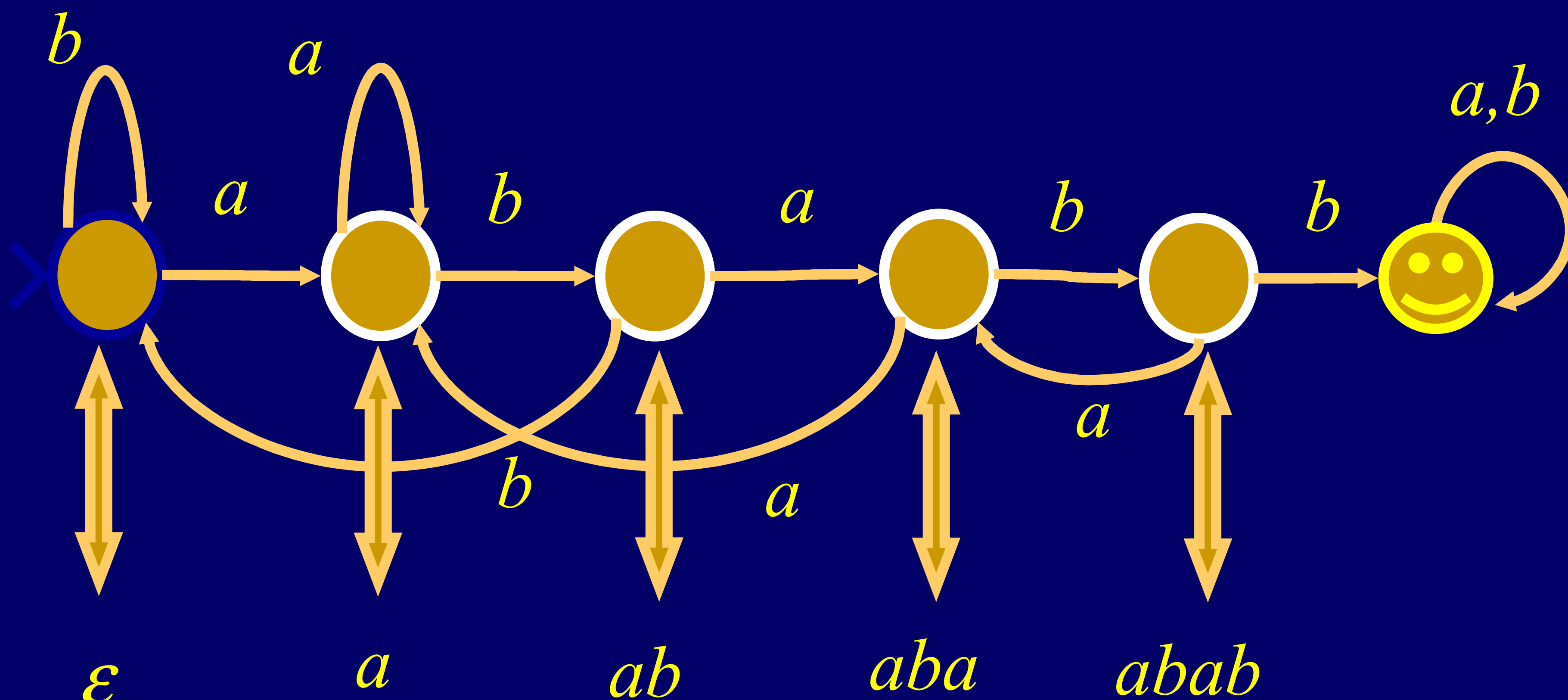


All paths evaluated in parallel. String is accepted if <u>some</u> path ends in an accept state.

NFA = DFA

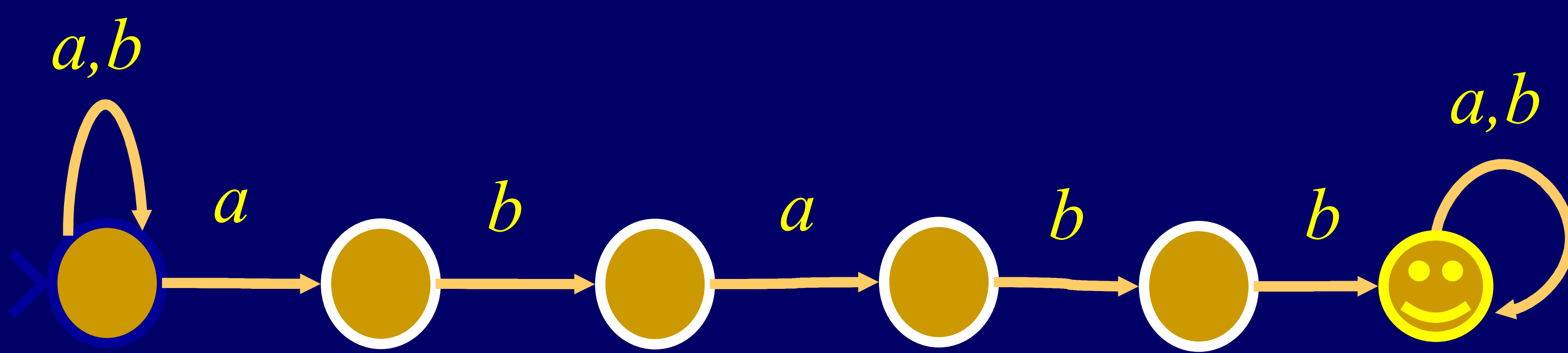
- Does this give us any more power?
- No, because any NFA can be written as an equivalent DFA.
- But they can be much more compact and convenient to work with

$L =$ all strings containing $ababb$ as a consecutive substring



Invariant: Machine is in state **s** when **s** is the longest suffix of the input (so far) that forms a prefix of *ababb.*

L = all strings containing $ababb$ as a consecutive substring



Nondeterministic machine. Equivalent to regular expression {a,b}*ababb{a,b}*

Regular Operatons on Languages

The class of regular languages is closed under union, concatenation, and *

$A \cup B = \{ s \mid s \in A \text{ or } s \in B \}$

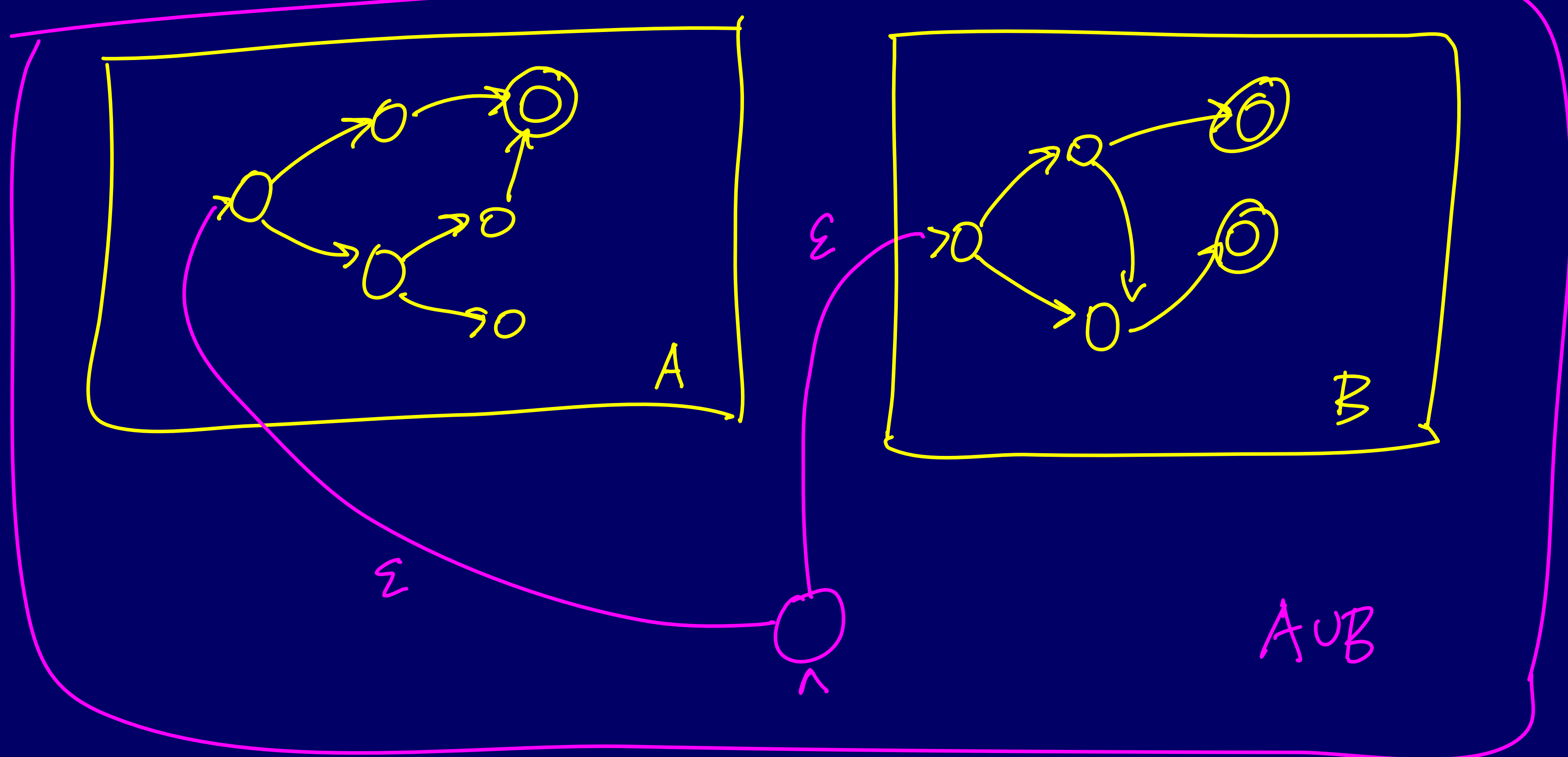
$A \circ B = \{ s \mid s = ab \text{ for } a \in A, b \in B \}$

$A^* = \{ s_1 s_2 \cdots s_k \mid s_i \in A, k \geq 0 \}$

Important to note: $\varepsilon \in A^*$

Regular Operatons: Proof by Picture

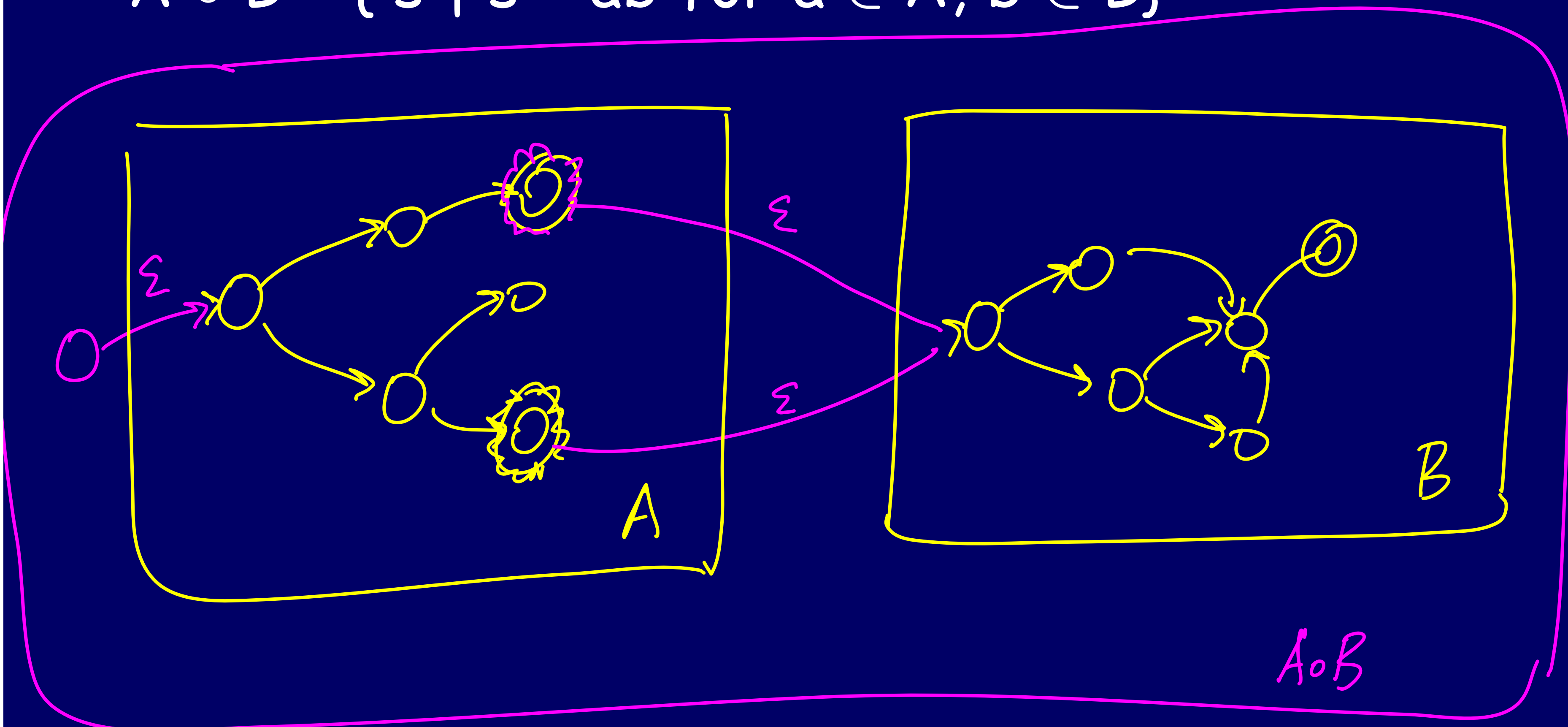
Closed under Union: $A \cup B = \{ s \mid s \in A \text{ or } s \in B \}$



Regular Operatons: Proof by Picture

Closed under concatenation:

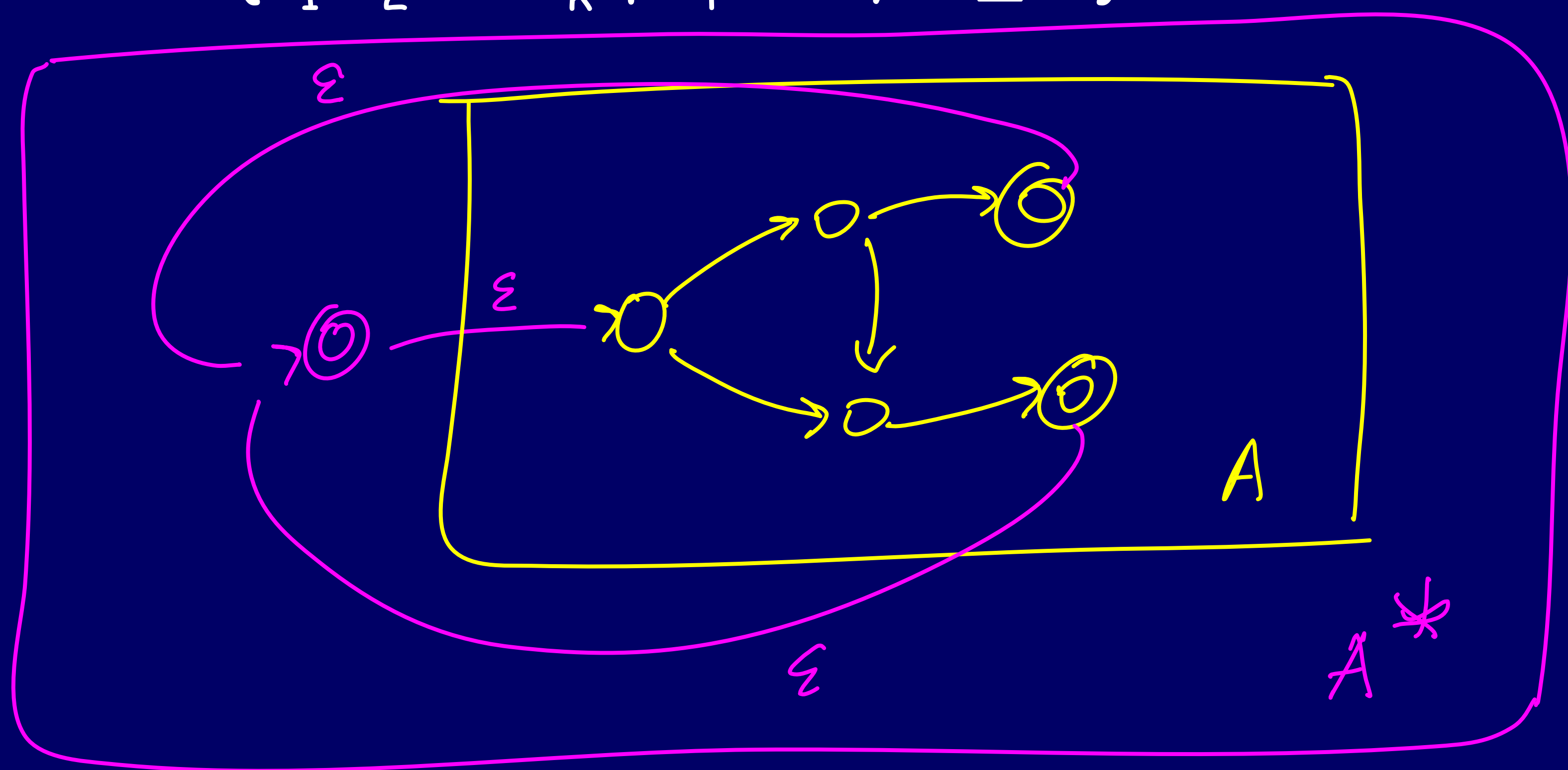
$A \circ B = \{ s \mid s = ab \text{ for } a \in A, b \in B\}$



Regular Operatons: Proof by Picture

Closed under *

$A^* = \{s_1 s_2 \cdots s_k \mid s_i \in A, k \geq 0\}$



NFA vs. DFA Summary

- Two slightly different descriptions of same set of languages
- DFAs: sequential evaluation
- NFAs: parallel evaluation
- NFA can be more compact and interpretable
- Uniqueness of state in DFA can make reasoning more transparent.

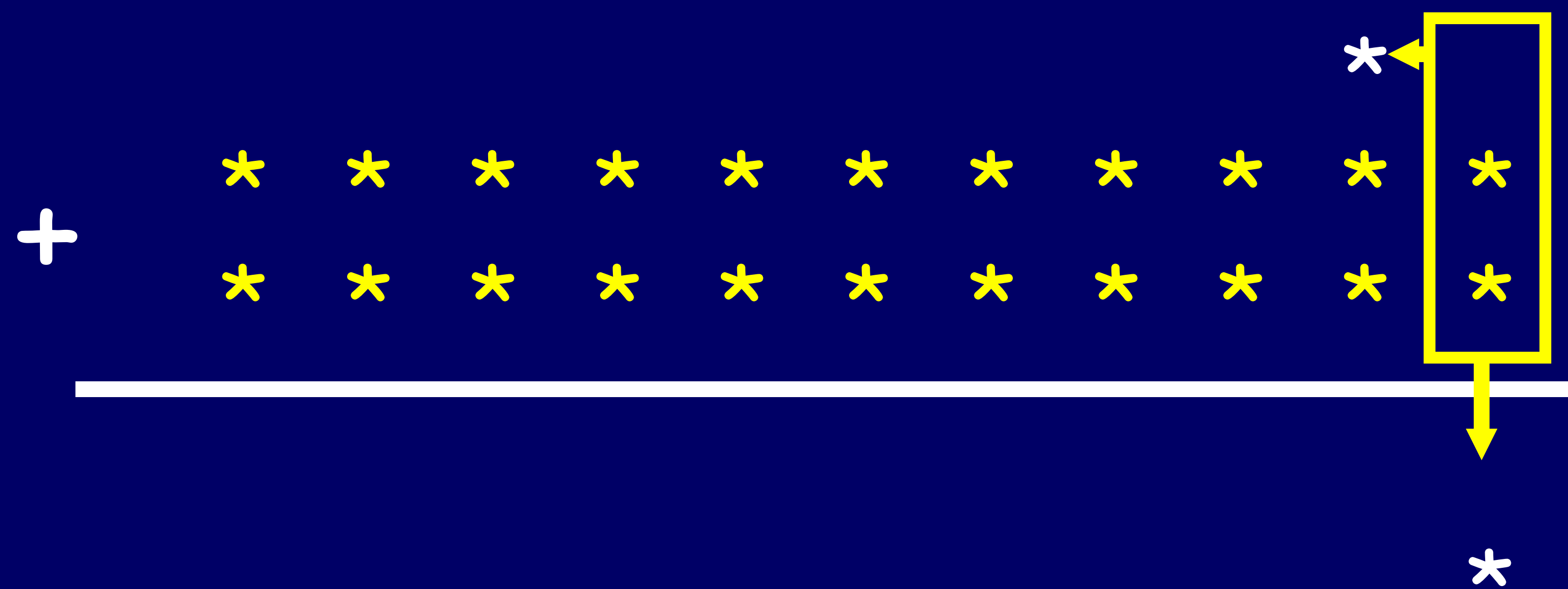
New Topic: “The Big O”

…and we don’t mean:

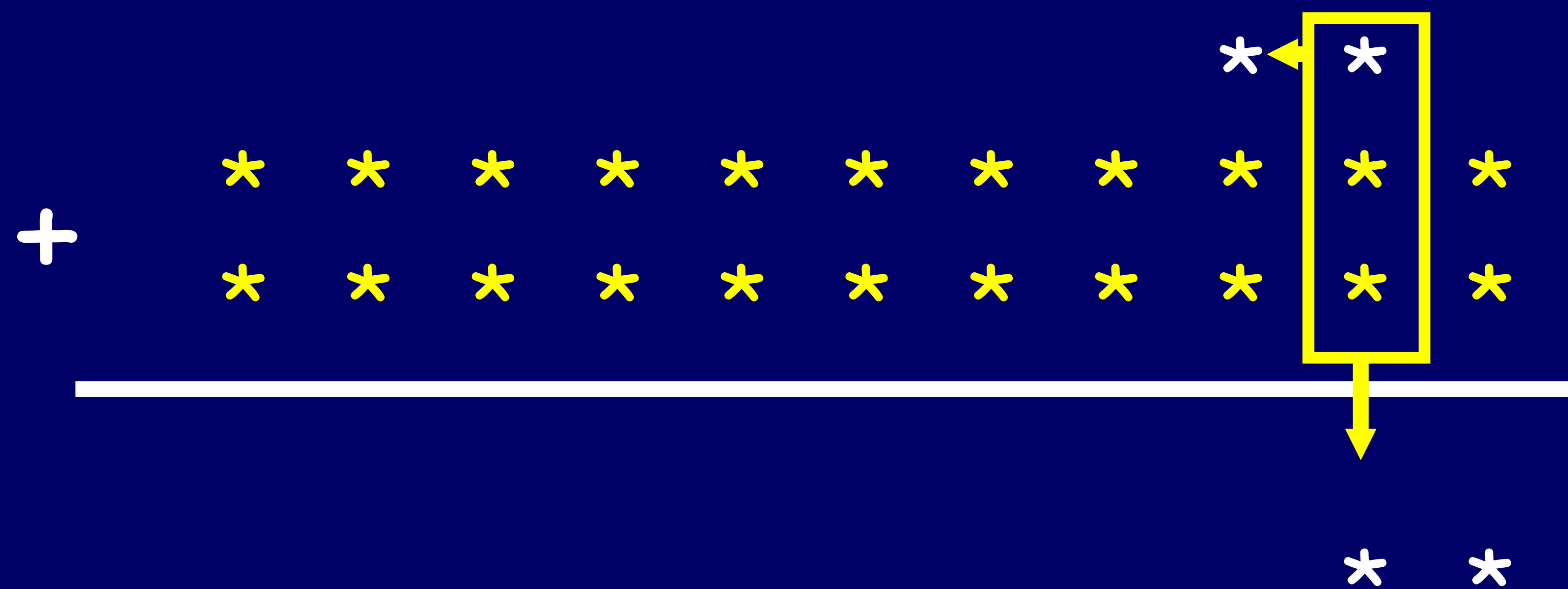
How to add 2 n-bit numbers.

```
  * * * * * * * * * * *
+ * * * * * * * * * * *
-----------------------
```

How to add 2 n-bit numbers.

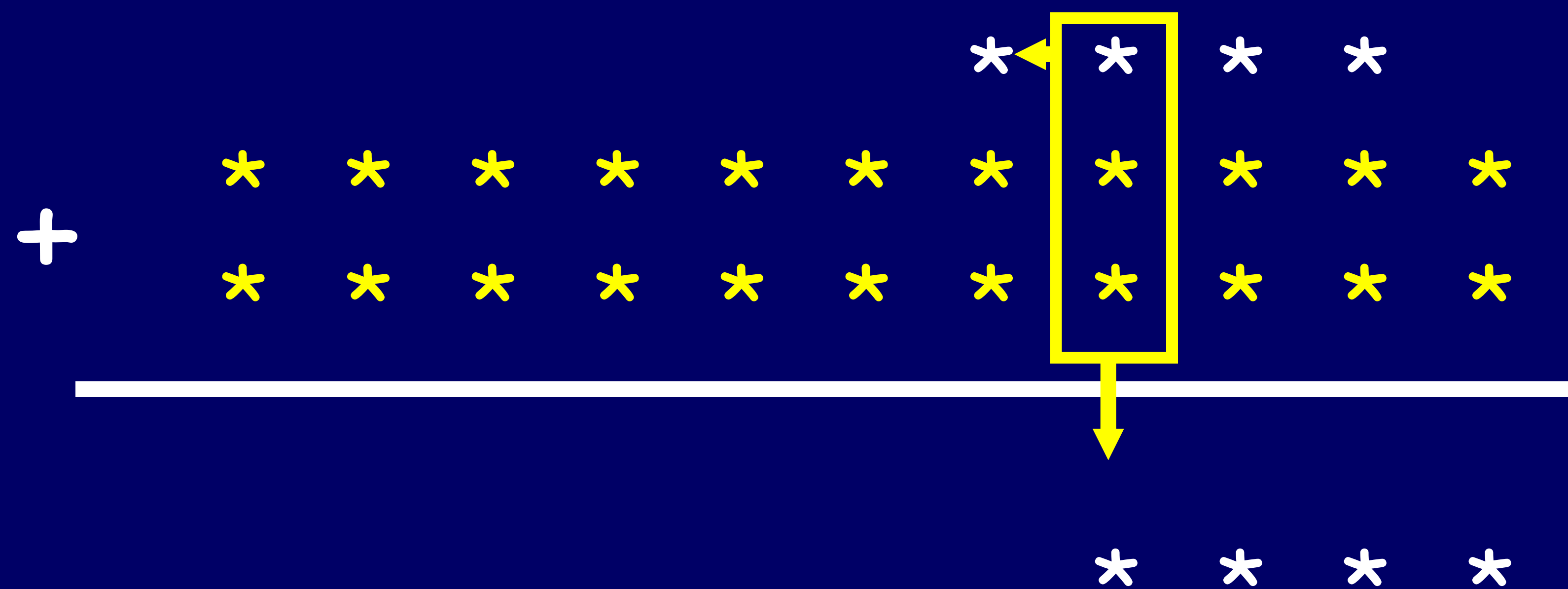


How to add 2 n-bit numbers.

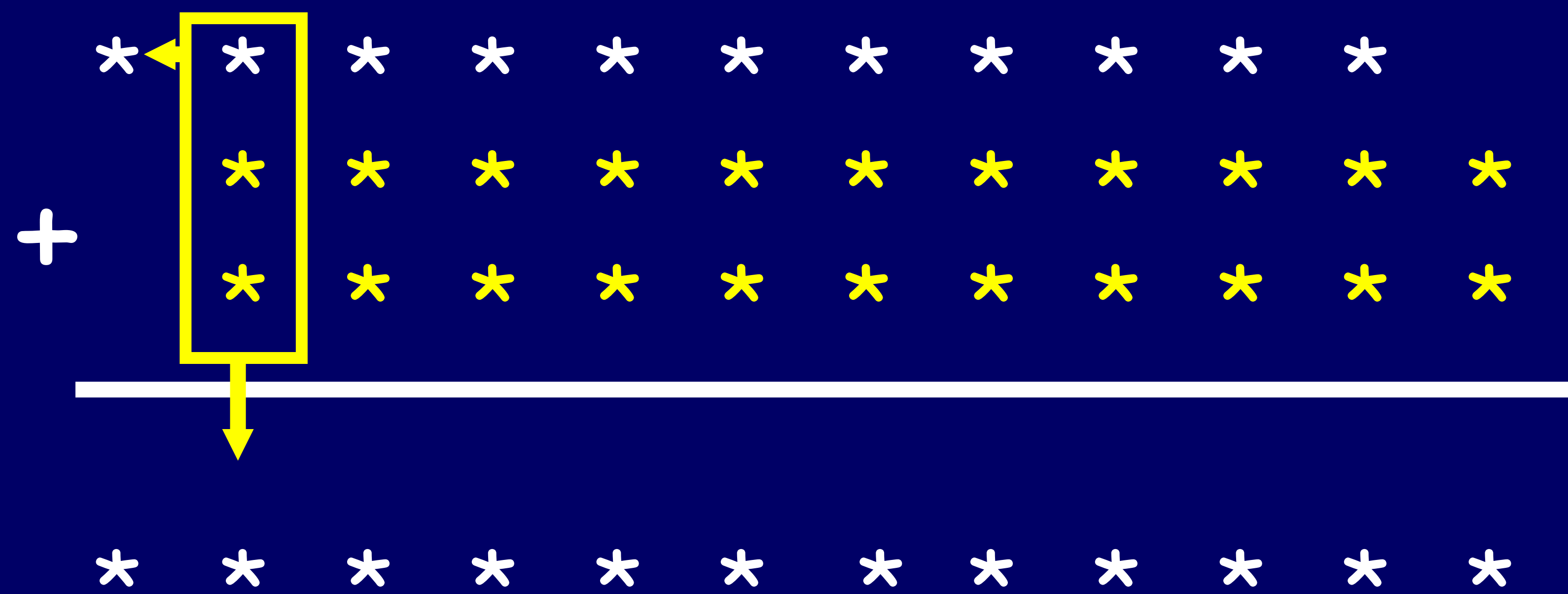


How to add 2 n-bit numbers.

How to add 2 n-bit numbers.

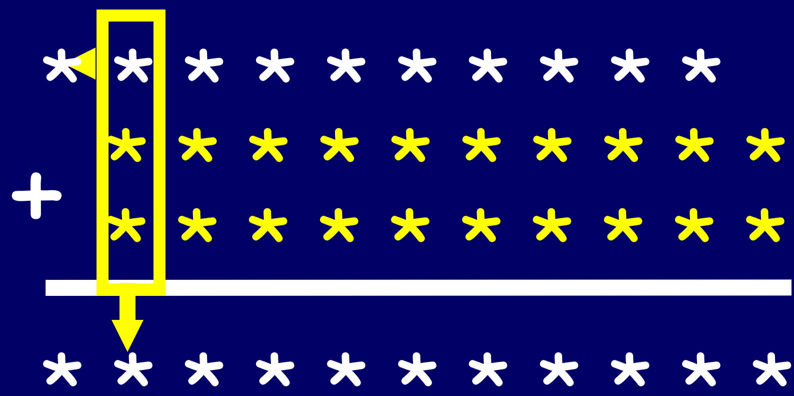


How to add 2 n-bit numbers.

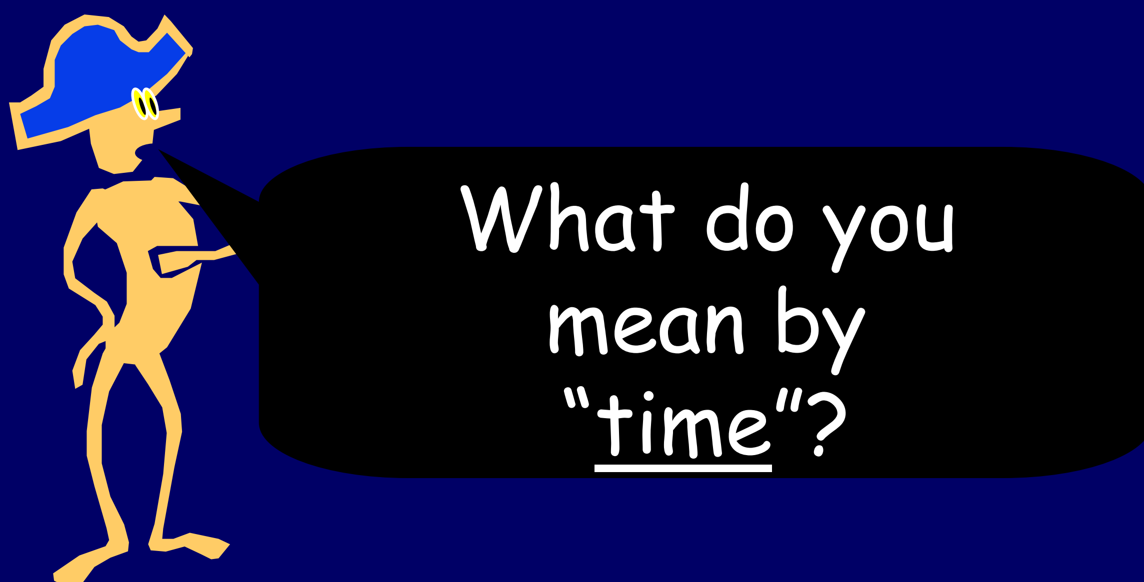


"Grade school addition"

Time complexity of grade school addition



$T(n)$ = amount of time grade school addition uses to add two n-bit numbers



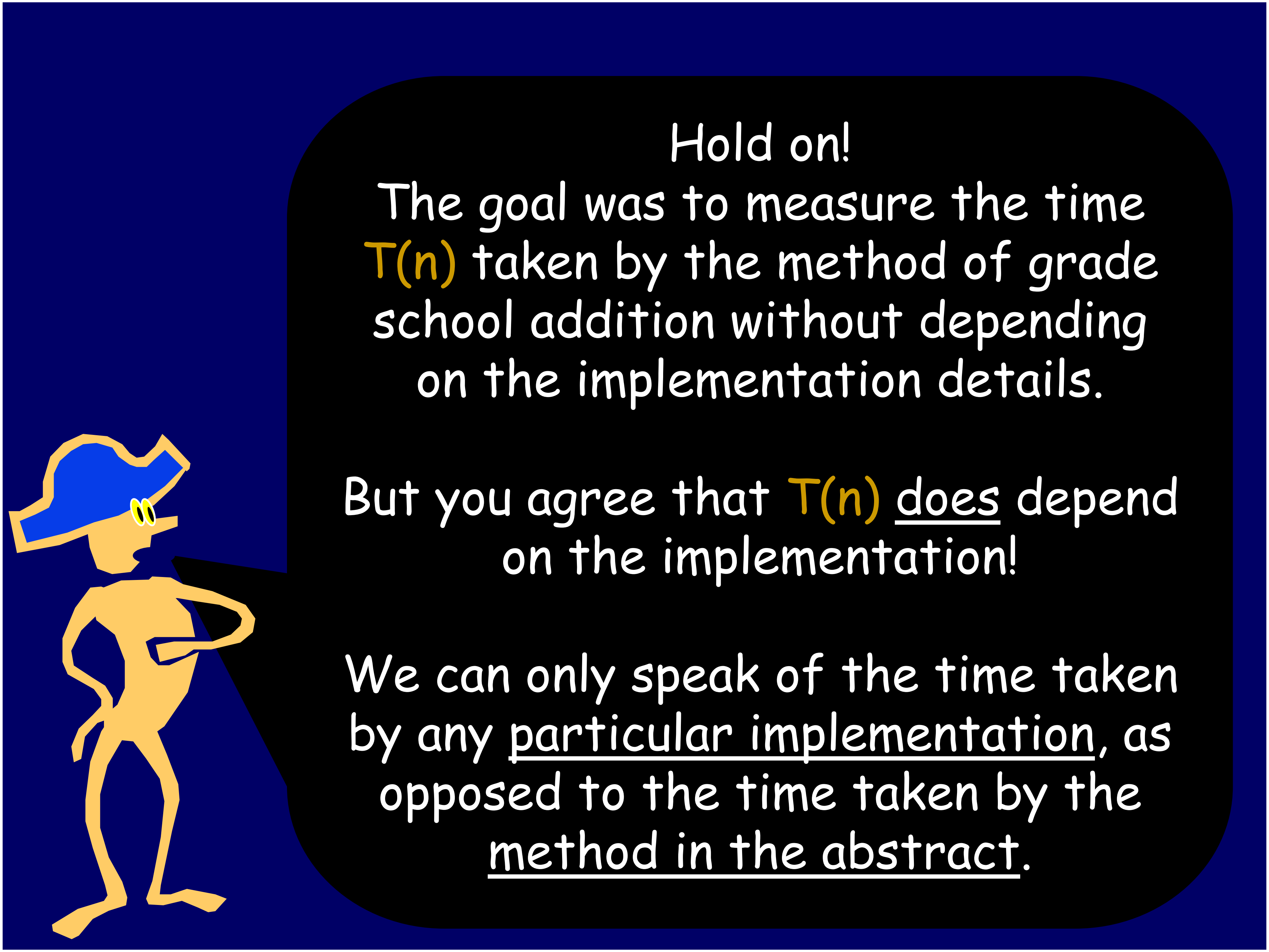
Our Goal

We want to define "<u>time</u>" in a way that
transcends implementation details
and allows us to make assertions about
grade school addition
in a very general yet useful way.

Roadblock ???

A *given* algorithm will take different amounts of time on the same inputs depending on such factors as:

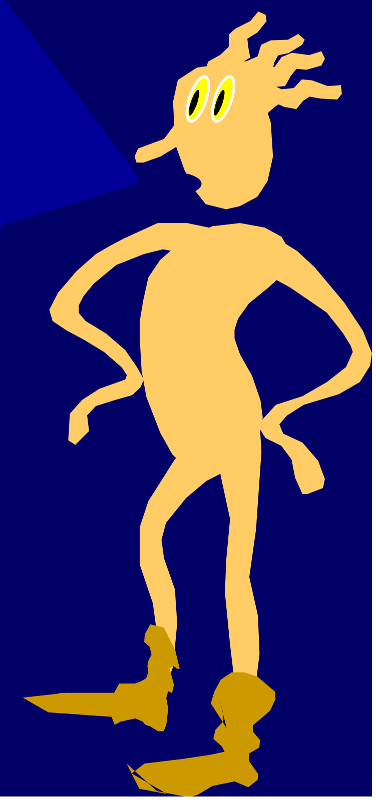
- Processor speed
- Instruction set
- Disk speed
- Brand of compiler


Hold on!
The goal was to measure the time T(n) taken by the method of grade school addition without depending on the implementation details.
But you agree that T(n) does depend on the implementation!
We can only speak of the time taken by any particular implementation, as opposed to the time taken by the method in the abstract.

Your objections are serious, Bonzo, but they are not insurmountable.

There is a very nice sense in which we can analyze grade school addition without having to worry about implementation details.

Here is how it works . . .

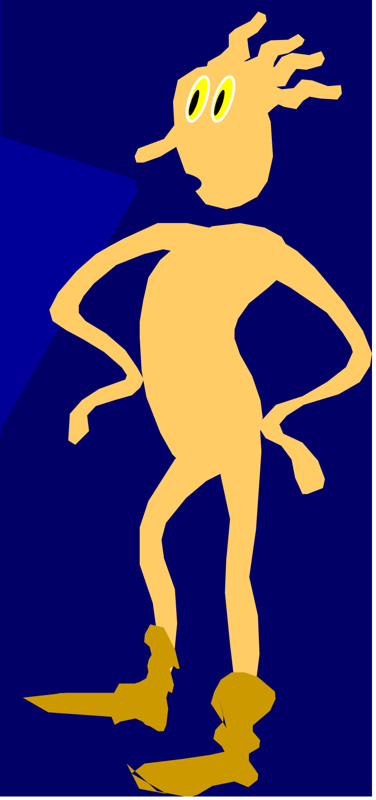


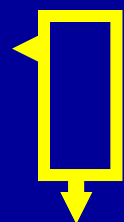
On any reasonable computer, adding 3 bits and writing down the two bit answer can be done in constant time.

Pick any particular computer M and define c to be the time it takes to perform  on that computer.

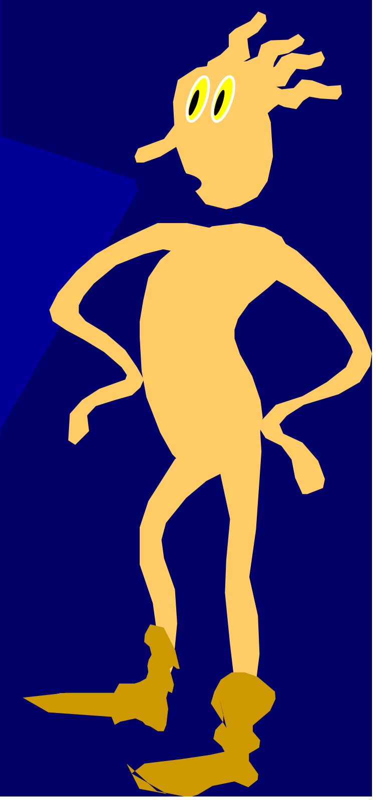
Total time to add two n-bit numbers using grade school addition: cn

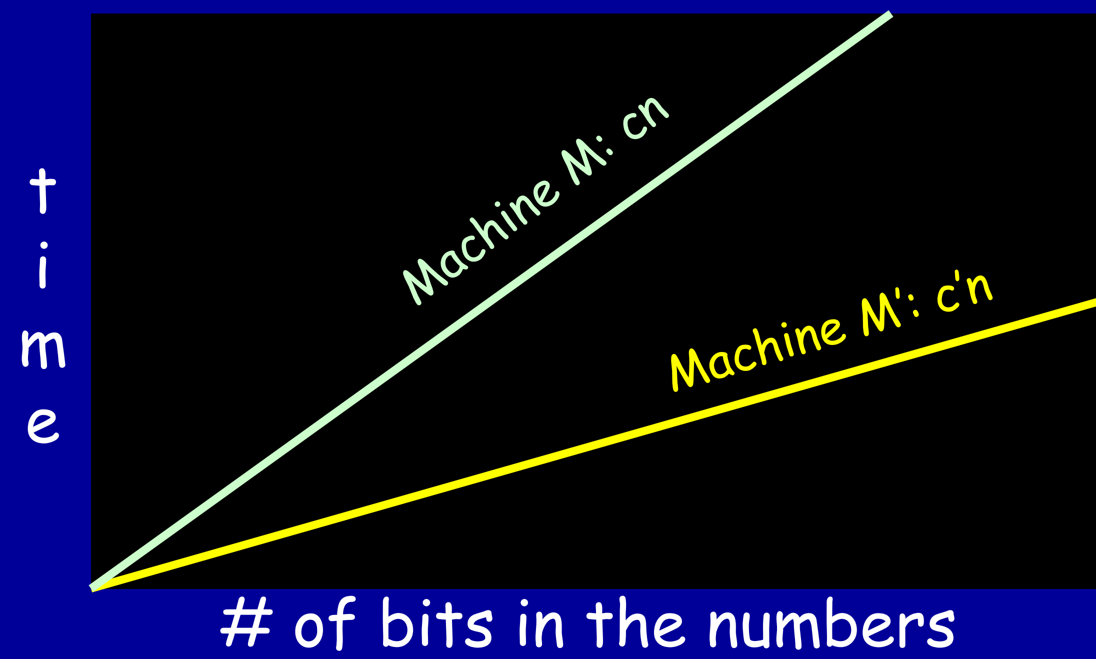
[c time for each of n columns]



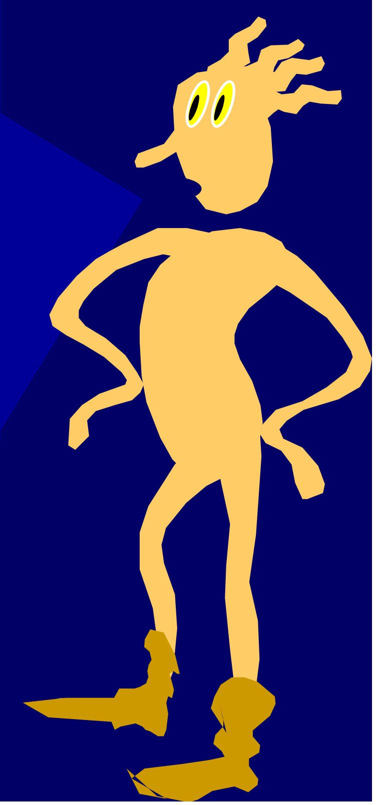
On another computer M', the time to perform  may be c'.

Total time to add two n-bit numbers using grade school addition: $c'n$
[c' time for each of n columns]



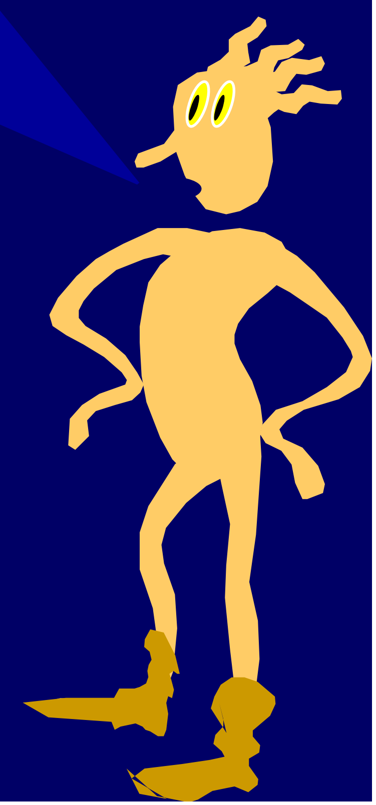


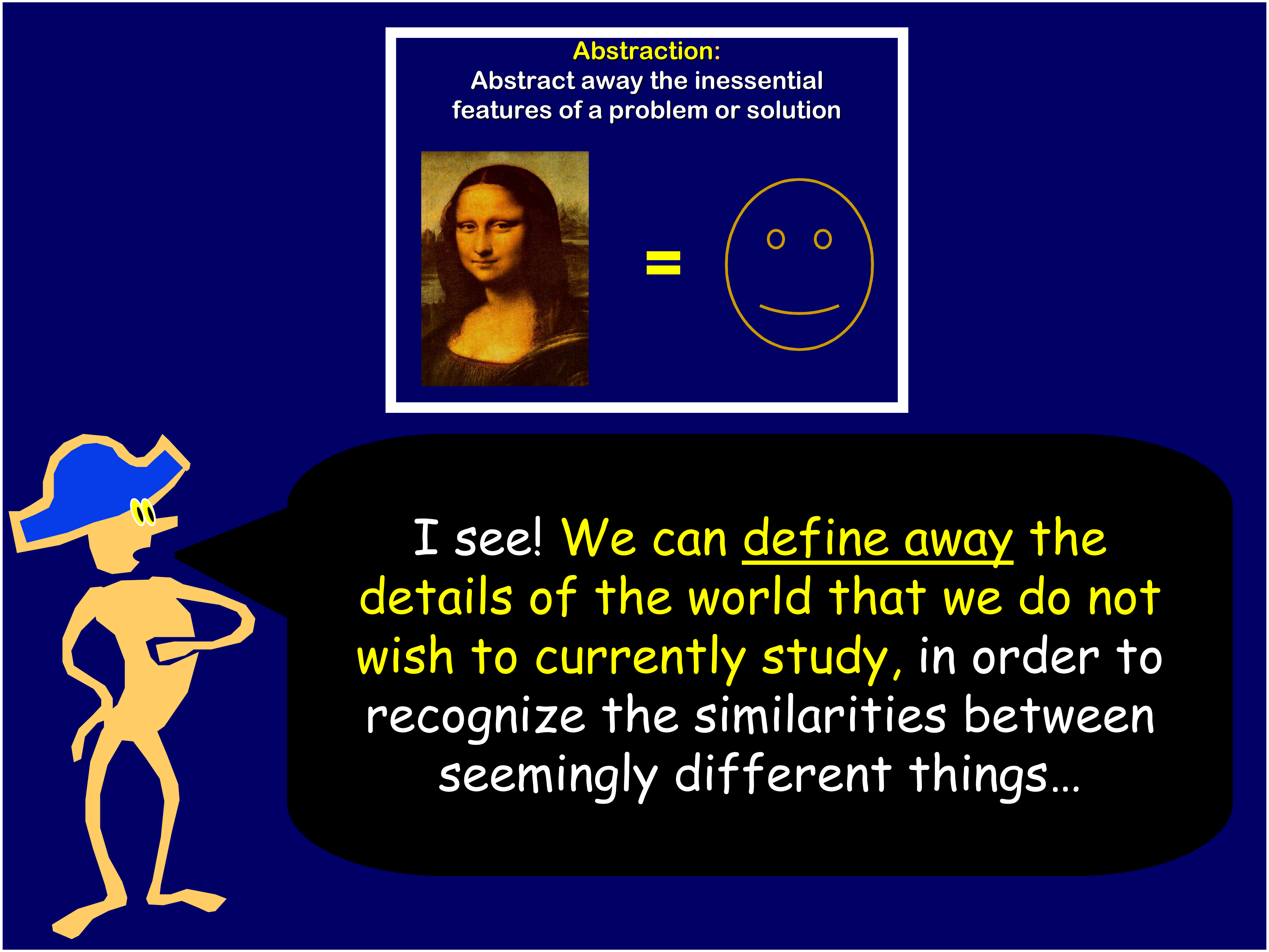
The fact that we get a line is invariant under changes of implementations. Different machines result in different slopes, but time grows linearly as input size increases.



Thus we arrive at an implementation independent insight:

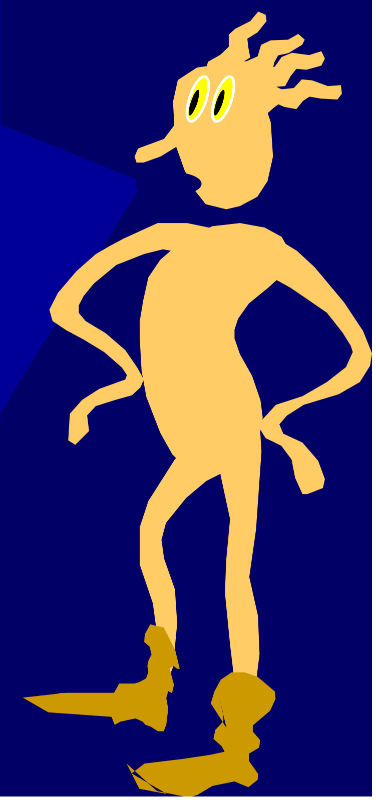
Grade School Addition is a linear time algorithm.




Abstraction:
Abstract away the inessential features of a problem or solution
=
I see! We can define away the details of the world that we do not wish to currently study, in order to recognize the similarities between seemingly different things…

Exactly, Bonzo!

This process of abstracting away
details and determining the
rate of resource usage
in terms of the problem size n
is one of the
fundamental ideas in
computer science.



Time vs Input Size

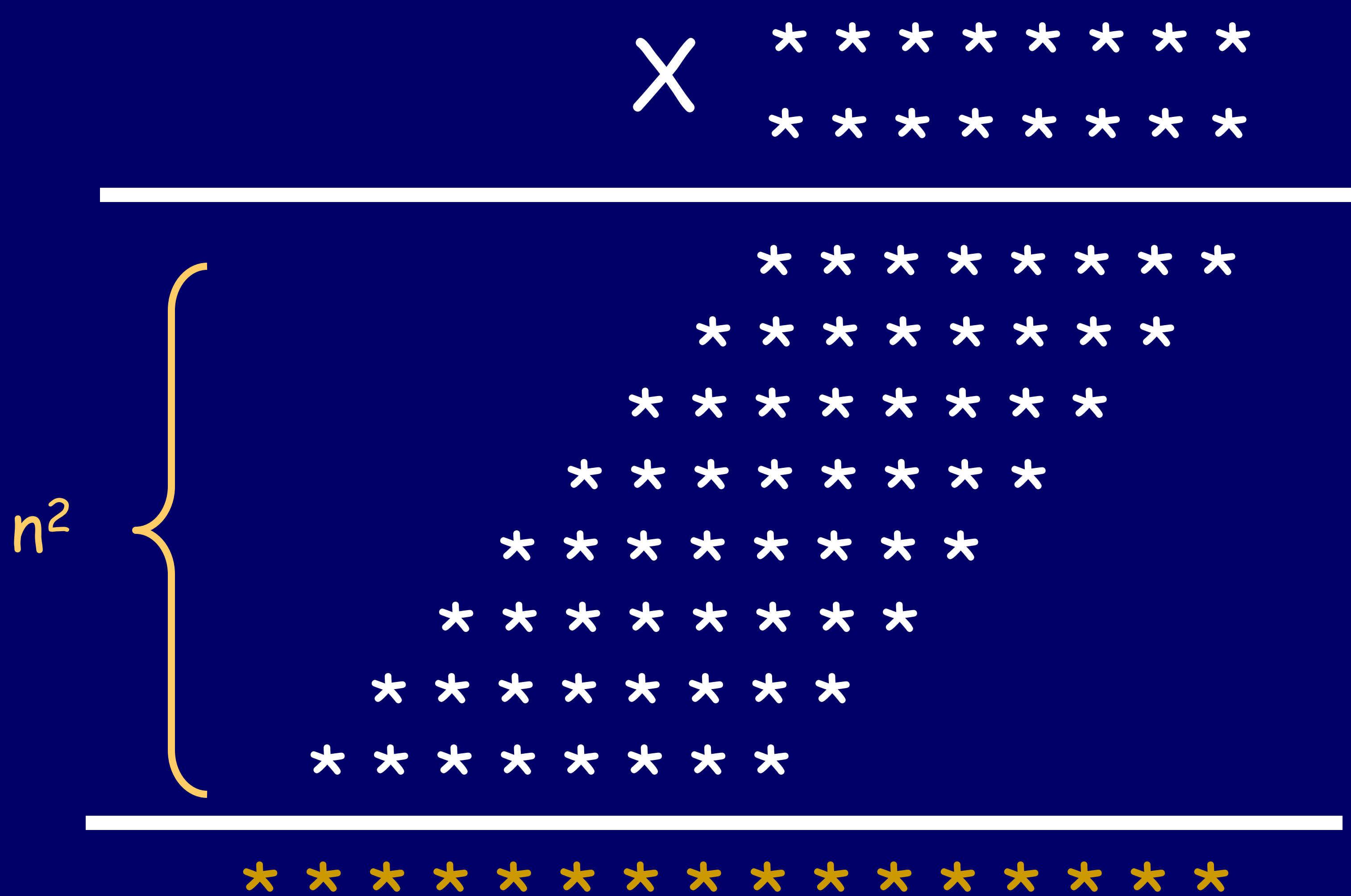
For any algorithm, define
Input Size = # of bits to specify its inputs.

Define
$TIME_n$ = the worst-case amount of time used on inputs of size n

We often ask:

What is the growth rate of $Time_n$?

How to multiply 2 n-bit numbers.



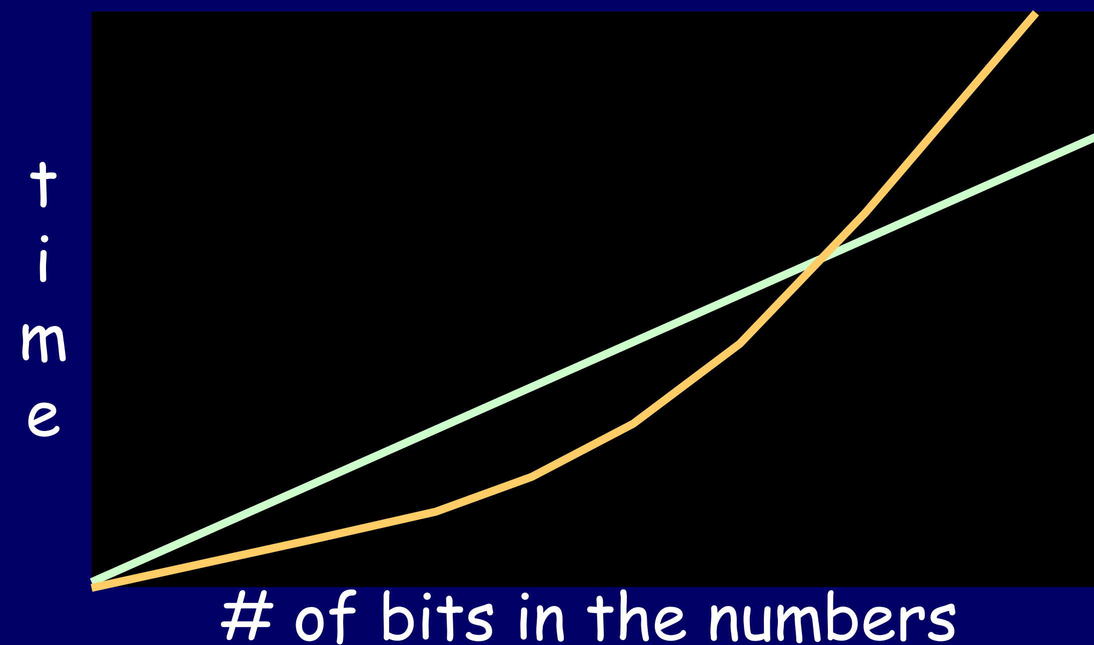
How to multiply 2 n-bit numbers.

```
          * * * * * * * *
  X       * * * * * * * *
-----------------------------
                * * * * * * * *
              * * * * * * * *
            * * * * * * * *
  n²      * * * * * * * *
        * * * * * * * *
      * * * * * * * *
    * * * * * * * *
  * * * * * * * *
-----------------------------
* * * * * * * * * * * * * * * *
```

I get it!

The total time is bounded by cn^2 (abstracting away the implementation details).

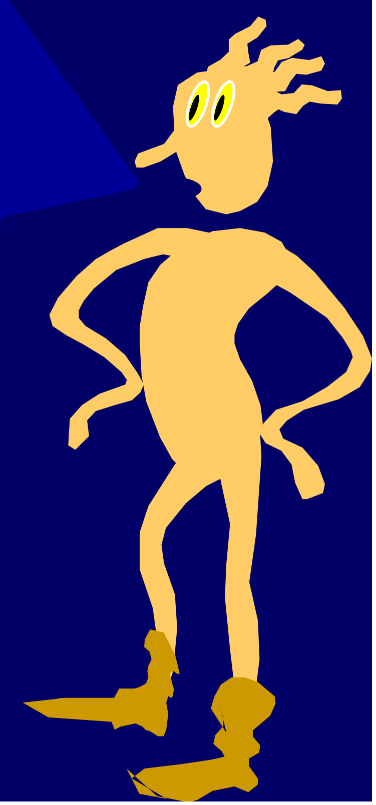
Grade School Addition: Linear time
Grade School Multiplication: Quadratic time



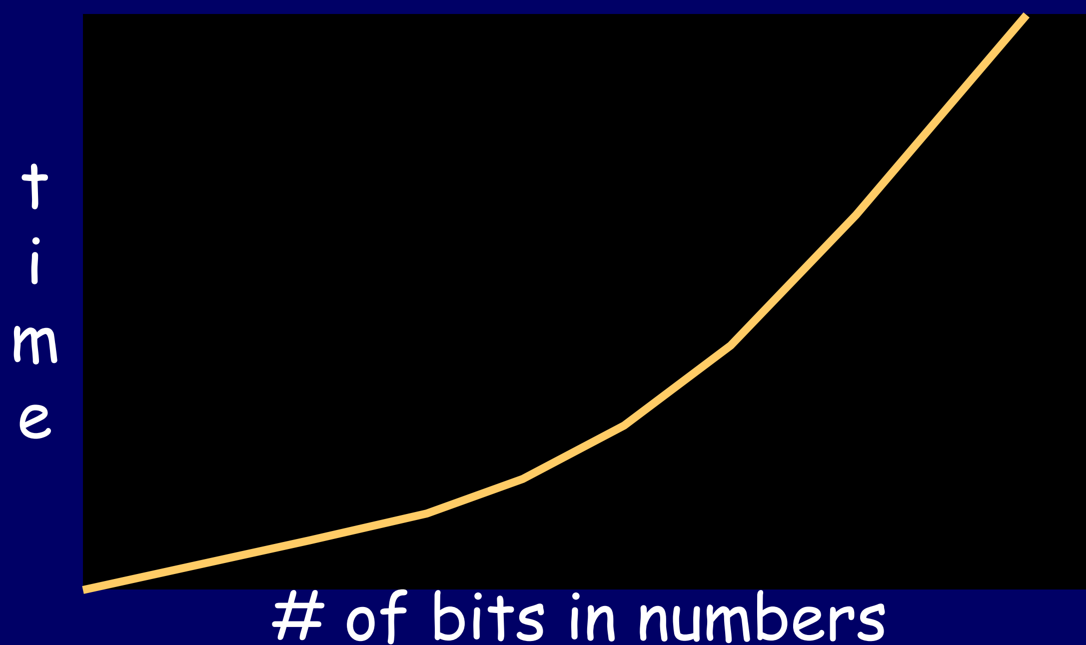
No matter how dramatic the difference in the constants, the quadratic curve will eventually dominate the linear curve

Graphic Demonstration of Running Times

http://www.cs.bell-labs.com/cm/cs/pearls/sortanim.html

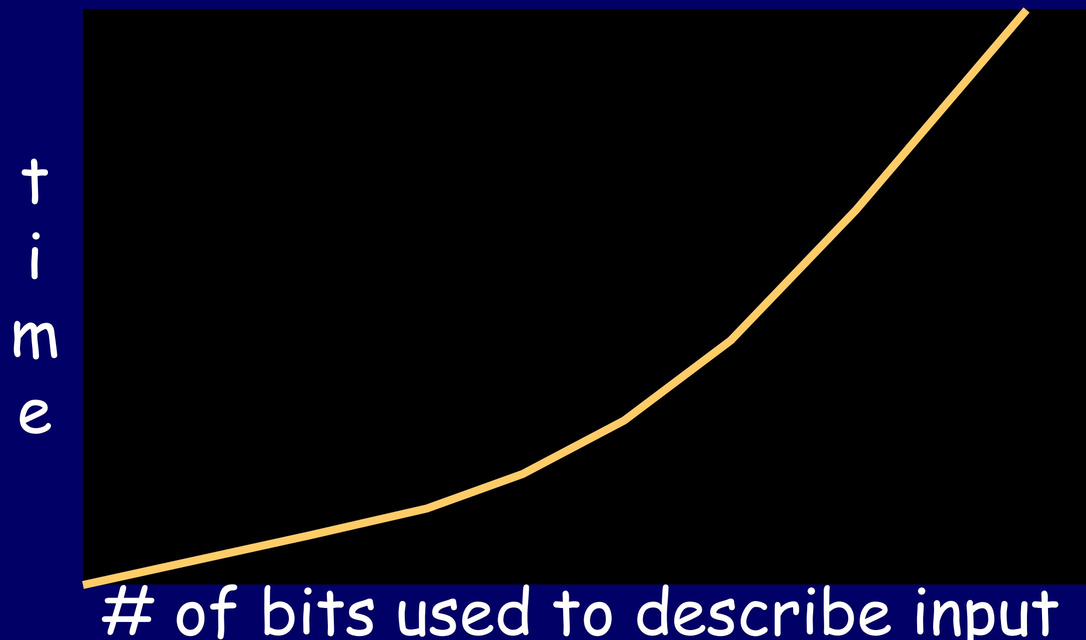

Ok, so…
How much time does it
take to square
the number n using
grade school multiplication?

Grade School Multiplication: Quadratic time



$c(\log n)^2$ time to square the number n

Time Versus Input Size



Input size is measured in <u>bits</u>, unless we say otherwise.

How much time does it take?

<u>Nursery School Addition</u>

Input: Two n-bit numbers, a and b

Output: $a + b$

Start at a and increment (by 1) b times

$T(n) = ?$

How much time does it take?

<u>Nursery School Addition</u>

Input: Two n-bit numbers, a and b

Output: $a + b$

Start at a and increment (by 1) b times

$T(n) = ?$

If $b = 000...0000$, then NSA takes almost no time.

If $b = 1111...11111$, then NSA takes $c\, n2^n$ time.

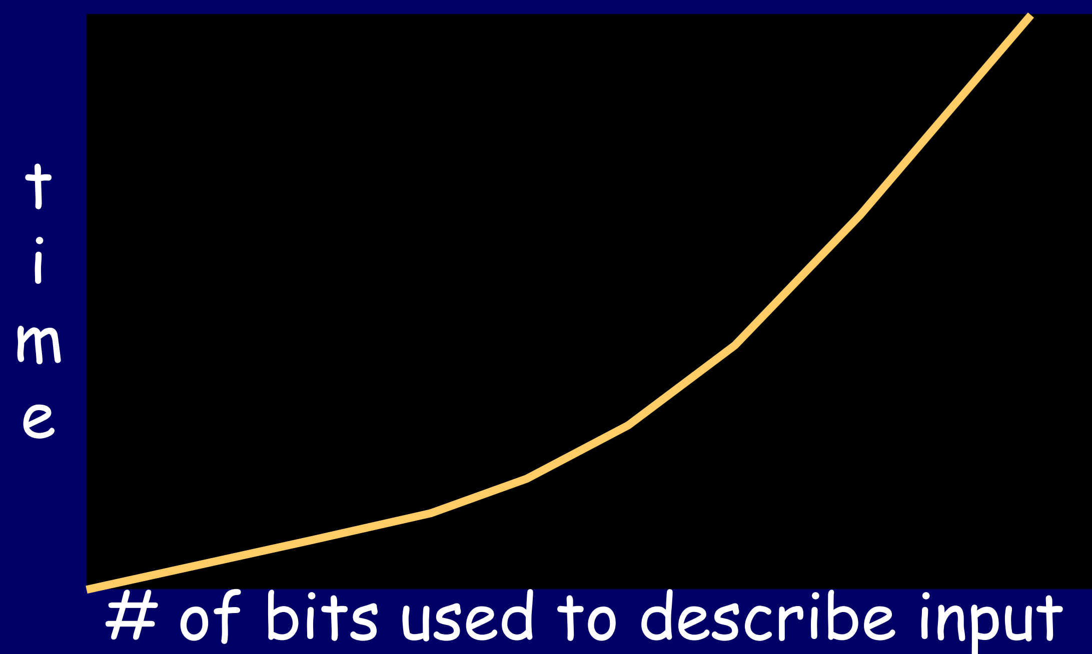
Exponential Worst Case time !!

Worst Case Time

Worst Case Time $T(n)$ for algorithm A:

$T(n) = \text{Max}_{[\text{all permissible inputs } X \text{ of size } n]}$ (Running time of algorithm A on input X).

Worst-case Time Versus Input Size



Worst Case Time Complexity

What is $T(n)$?

Kindergarten Multiplication

Input: Two n-bit numbers, a and b

Output: $a * b$

Start with a and add a, $b-1$ times

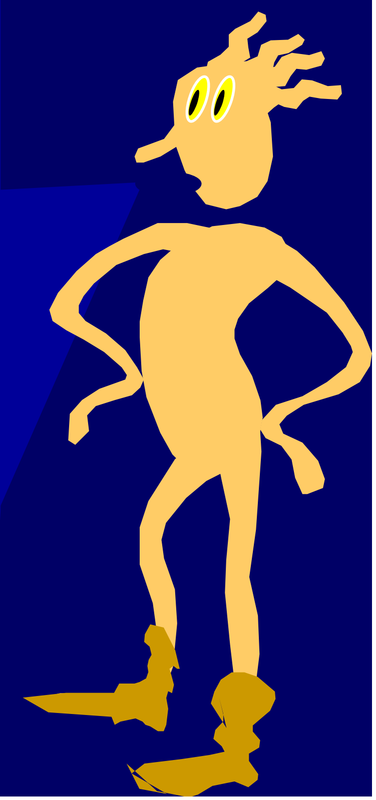
Remember, we always pick the WORST CASE input for the input size n.

Thus, $T(n) = c\, n2^n$

Exponential Worst Case time !!

Thus, Nursery School adding and Kindergarten multiplication are <u>exponential</u> time.
They scale HORRIBLY as input size grows.

Grade school methods scale <u>polynomially</u>: just linear and quadratic.
Thus, we can add and multiply fairly large numbers.

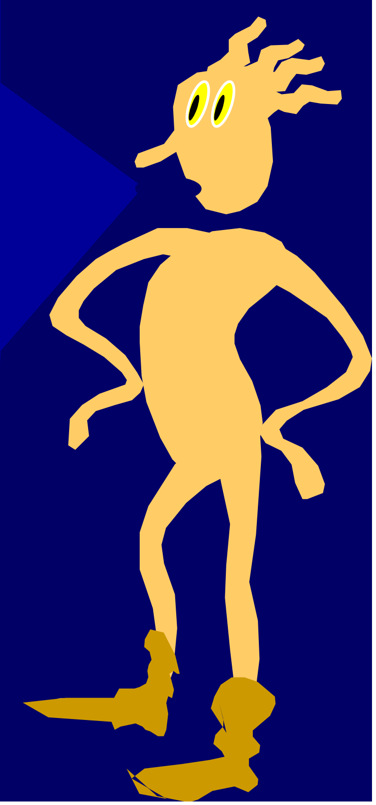


If $T(n)$ is <u>not polynomial</u>, the algorithm is <u>not efficient</u>: the run time scales too poorly with the input size.

This will be the yardstick with which we will measure "efficiency".

Multiplication is efficient, what about "reverse multiplication"?

Let's define FACTORING(N) to be any method to produce a non-trivial factor of N, or to assert that N is prime.



Factoring The Number N
By Trial Division

<u>Trial division up to $\sqrt{N}$</u>

for $k = 2$ to $\sqrt{N}$ do
    if $k \mid N$ then
    return "N has a non-trivial factor k"

return "N is prime"

c $\sqrt{N}(\log N)^2$ time if division is c $(\log N)^2$ time

On input N, trial factoring uses $c\sqrt{N}(\log N)^2$ time.

Is this efficient?

No! The input length $n = \log N$.
Hence we're using $c\,2^{n/2}\,n^2$ time.

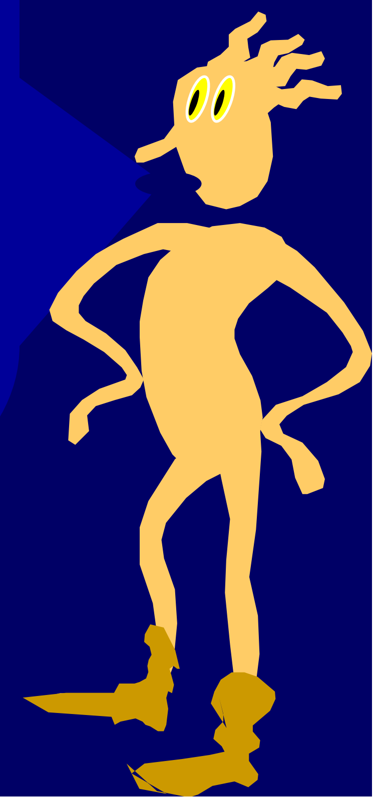
The time is EXPONENTIAL in the input length n.

Can we do better?

We know of methods for FACTORING that are sub-exponential

(about $2^{n^{1/3}}$ time)

but nothing efficient.



Useful notation to discuss growth rates

For any monotonic function f from the positive integers to the positive integers, we say

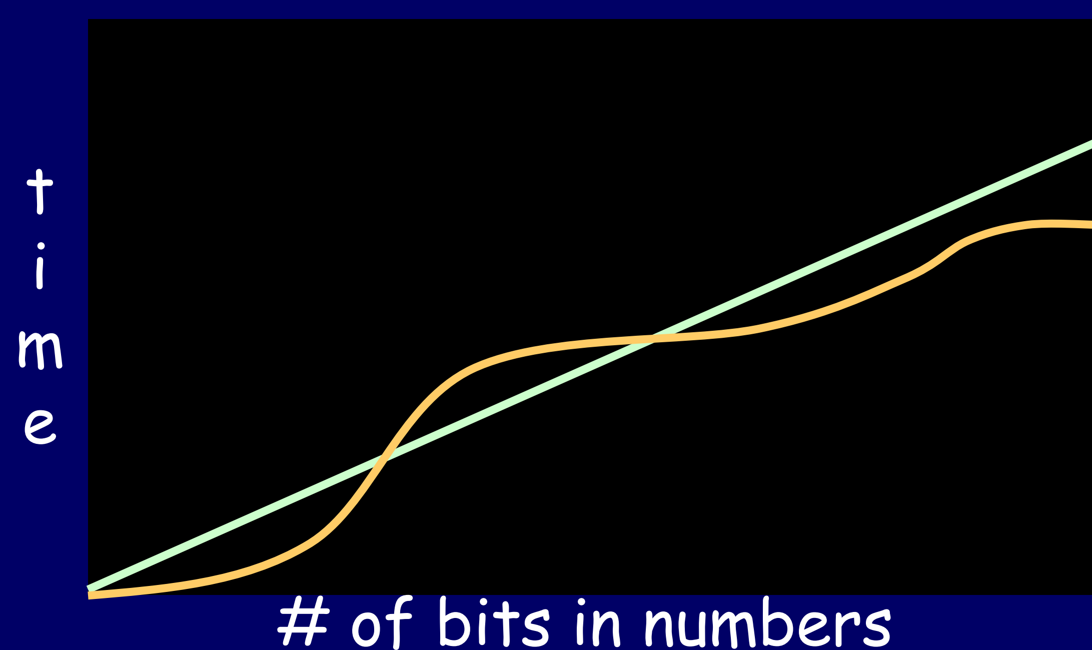
“$f = O(n)$” or “f is $O(n)$”

if

Some constant times n eventually dominates f

[Formally: there exists a constant c such that for all sufficiently large n: $f(n) \leq cn$]

$f = O(n)$ means that there is a line that can be drawn that stays above f from some point on.



Other useful notation: Ω

For any monotonic function f from the positive integers to the positive integers, we say

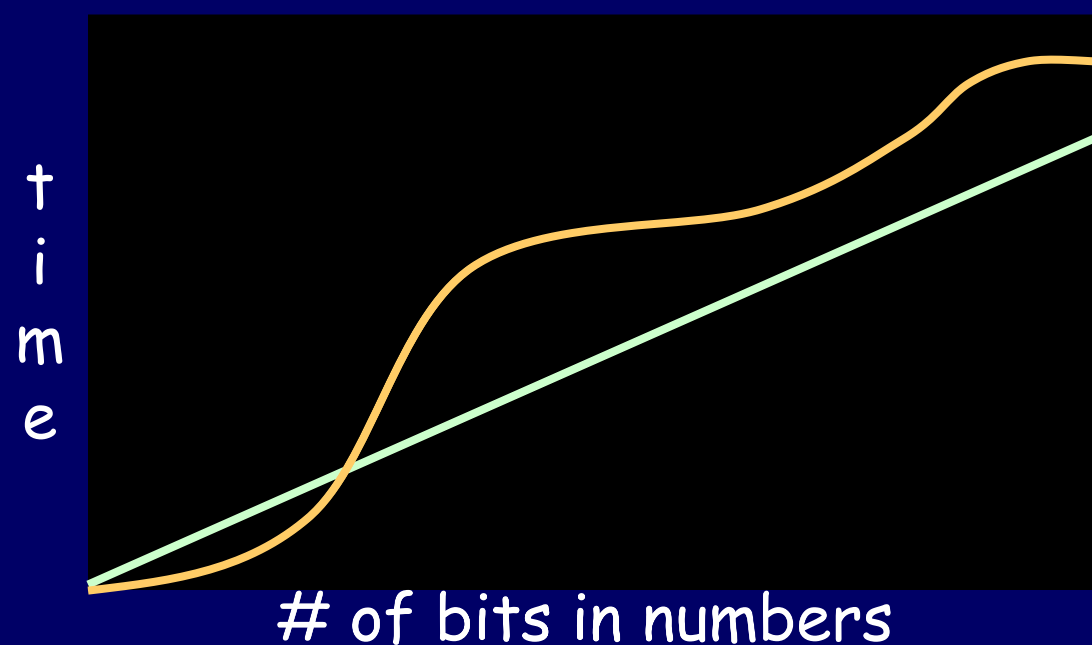
"$f = \Omega(n)$" or "f is $\Omega(n)$"

if:

f eventually dominates some constant times n

[Formally: there exists a constant c such that for all sufficiently large n: $f(n) \geq cn$]

$f = \Omega(n)$ means that there is a line that can be drawn that stays <u>below</u> f from some point on



Yet more useful notation: Θ

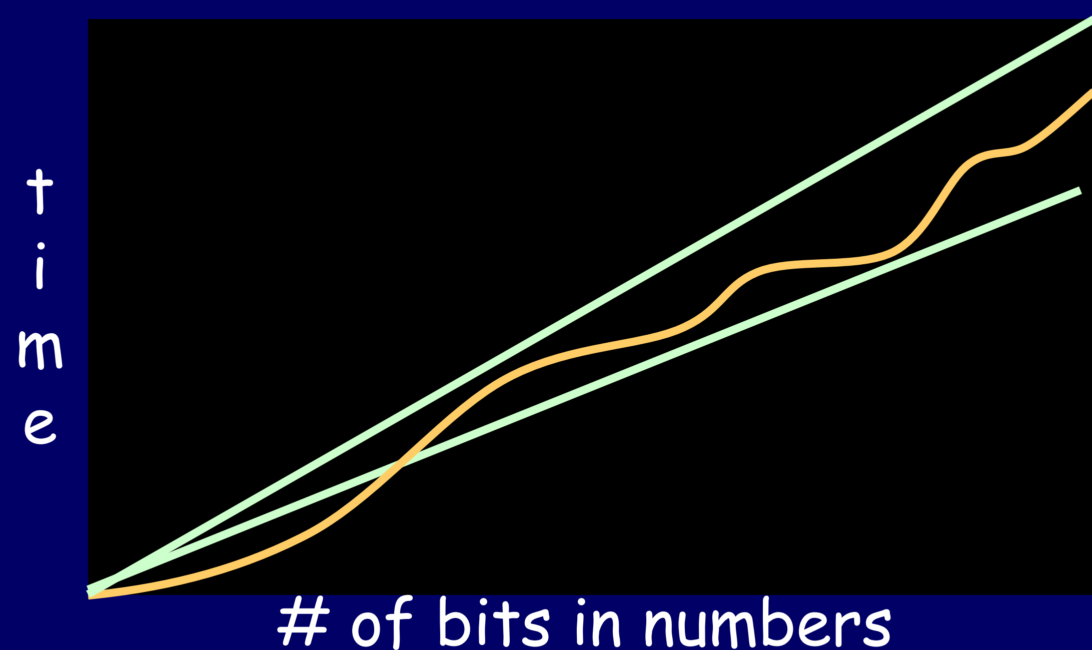
For any monotonic function f from the positive integers to the positive integers, we say

"$f = \Theta(n)$" or "f is $\Theta(n)$"

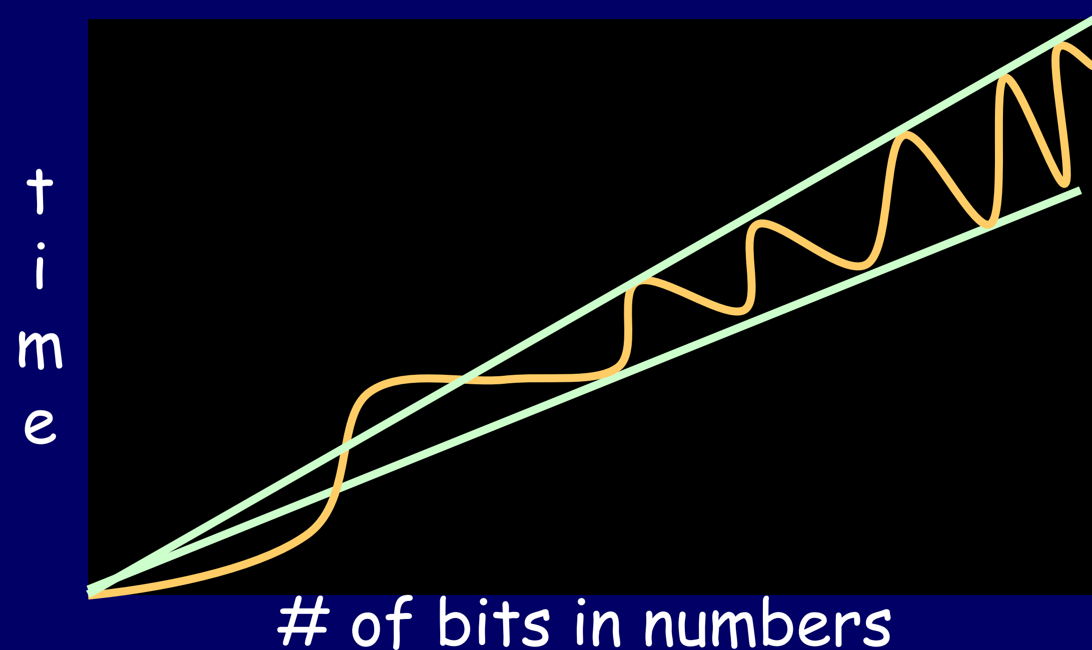
if:

$f = O(n)$ <u>and</u> $f = \Omega(n)$

$f = \Theta(n)$ means that f can be sandwiched between two lines from some point on.



$f = \Theta(n)$ means that f can be sandwiched between two lines from some point on.



Useful notation to discuss growth rates

For any two monotonic functions f and g from the positive integers to the positive integers, we say

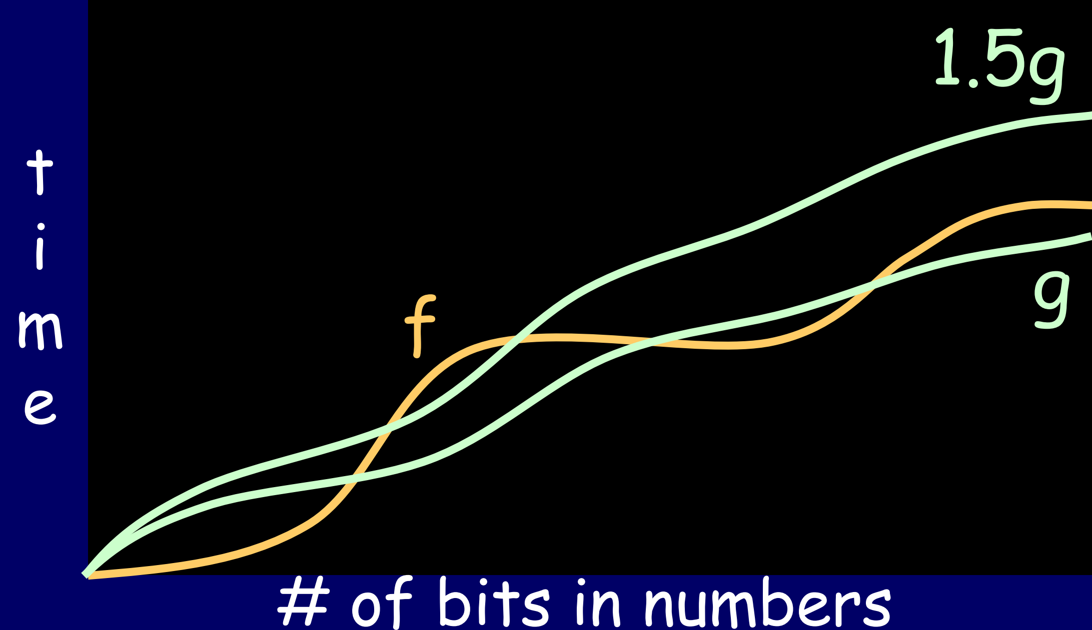
“$f = O(g)$” or “f is $O(g)$”

if

Some constant times g eventually dominates f

[Formally: there exists a constant c such that for all sufficiently large n: $f(n) \leq c\, g(n)$]

$f = O(g)$ means that there is some constant c such that $c\ g(n)$ stays above $f(n)$ from some point on.



More useful notation: Ω

For any two monotonic functions f and g from the positive integers to the positive integers, we say

"$f = \Omega(g)$" or "f is $\Omega(g)$"

if:

f eventually dominates some constant times g

[Formally: there exists a constant c such that for all sufficiently large n: $f(n) \geq c\, g(n)$]

Yet more useful notation: Θ

For any two monotonic functions f and g from the positive integers to the positive integers, we say

"$f = \Theta(g)$" or "f is $\Theta(g)$"

if:

$f = O(g)$ <u>and</u> $f = \Omega(g)$

- $n = O(n^2)$?
  - YES

Take $c = 1$.
For all $n \geq 1$, it holds that $n \leq cn^2$

Quickies

- $n = O(n^2)$?
  - YES
- $n = O(\sqrt{n})$?
  - NO

Suppose it were true that $n \le c \sqrt{n}$
for some constant c and large enough n

Cancelling, we would get $\sqrt{n} \le c$.
Which is false for $n > c^2$

Quickies

Quickies

- $n = O(n^2)$?
  - YES
- $n = O(\sqrt{n})$?
  - NO
- $3n^2 + 4n + \pi = O(n^2)$?
  - YES
- $3n^2 + 4n + \pi = \Omega(n^2)$?
  - YES
- $n^2 = \Omega(n \log n)$?
  - YES
- $n^2 \log n = \Theta(n^2)$

$$\left.\begin{array}{l} 3n^2 + 4n + \pi = O(n^2) \\ 3n^2 + 4n + \pi = \Omega(n^2) \end{array}\right\} 3n^2 + 4n + \pi = \Theta(n^2)$$

Two statements in one!

$$n^2 \log n = \Theta(n^2)$$

$$n^2 \log n = O(n^2)$$

NO

$$n^2 \log n = \Omega(n^2)$$

YES

Names For Some Growth Rates

Linear Time: $T(n) = O(n)$

Quadratic Time: $T(n) = O(n^2)$

Cubic Time: $T(n) = O(n^3)$

Polynomial Time:

for some constant k, $T(n) = O(n^k)$.

Example: $T(n) = 13n^5$

Large Growth Rates

Exponential Time:

for some constant k, $T(n) = O(k^n)$

Example: $T(n) = n2^n = O(3^n)$

Almost Exponential Time:

for some constant k, $T(n) = 2^{\text{kth root of n}}$

Example: $\mathrm{T}(\mathrm{n}) = 2^{\sqrt{n}}$

Small Growth Rates

Logarithmic Time: $T(n) = O(\log n)$
Example: $T(n) = 15\log_2(n)$

Polylogarithmic Time:
for some constant k, $T(n) = O(\log^k(n))$

Note: These kind of algorithms can't possibly read all of their inputs.

Binary Search

A very common example of logarithmic time is looking up a word in a sorted dictionary.

Some Big Ones

<u>Doubly Exponential Time</u> means that for some constant k

$$\mathrm{T}(\mathrm{n}) = 2^{2^{kn}}$$

<u>Triply Exponential</u>

$$\mathrm{T}(\mathrm{n}) = 2^{2^{2^{kn}}}$$

And so forth.

Faster and Faster: 2STACK

2STACK(0) = 1

2STACK(n) = $2^{\text{2STACK(n-1)}}$

2STACK(1) = 2
2STACK(2) = 4
2STACK(3) = 16
2STACK(4) = 65536

2STACK(5) $\geq 10^{80}$
= atoms in universe

2STACK(n) = $$\underbrace{2^{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}}_{\text{"tower of n 2's"}}$$

And the inverse of 2STACK: log*

2STACK(0) = 1

2STACK(n) = $2^{2STACK(n-1)}$

2STACK(1) = 2
2STACK(2) = 4
2STACK(3) = 16
2STACK(4) = 65536

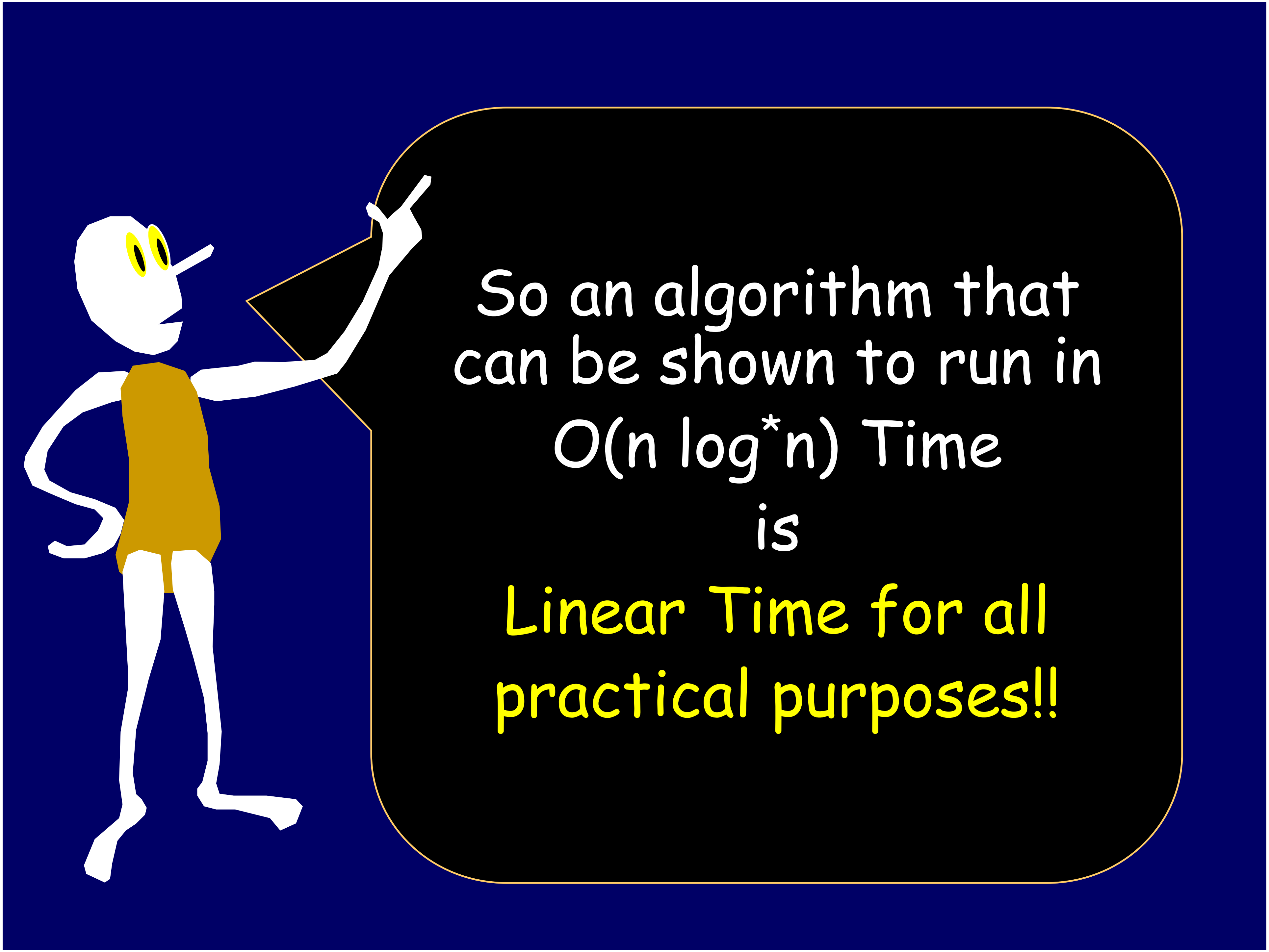
2STACK(5) $\geq 10^{80}$
= atoms in universe

log*(n) = # of times you have to apply the log function to n to make it ≤ 1

And the inverse of 2STACK: log*

$2STACK(0) = 1$	$\log^*(1) = 0$
$2STACK(n) = 2^{2STACK(n-1)}$	
$2STACK(1) = 2$	$\log^*(2) = 1$
$2STACK(2) = 4$	$\log^*(4) = 2$
$2STACK(3) = 16$	$\log^*(16) = 3$
$2STACK(4) = 65536$	$\log^*(65536) = 4$
$2STACK(5) \geq 10^{80}$ = atoms in universe	$\log^*(\text{atoms}) = 5$

$\log^*(n)$ = # of times you have to apply the log function to n to make it ≤ 1


So an algorithm that can be shown to run in O(n log*n) Time
is
Linear Time for all practical purposes!!

Ackermann's Function

$A(0, n) = n + 1$ for $n \geq 0$

$A(m, 0) = A(m - 1, 1)$ for $m \geq 1$

$A(m, n) = A(m - 1, A(m, n - 1))$ for $m, n \geq 1$

A(4,2) > # of particles in universe

A(5,2) can't be written out as decimal in this universe

Inverse Ackermann function

$A(0, n) = n + 1$ for $n \geq 0$

$A(m, 0) = A(m - 1, 1)$ for $m \geq 1$

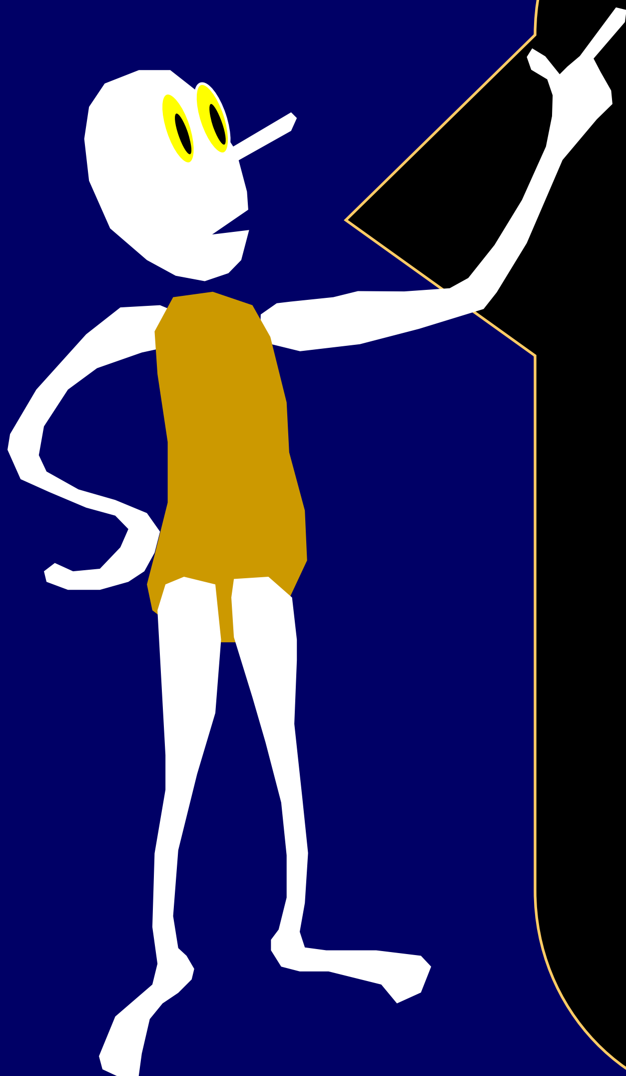
$A(m, n) = A(m - 1, A(m, n - 1))$ for $m, n \geq 1$

Define: $A'(k) = A(k,k)$

Inverse Ackerman $\alpha(n)$ is the inverse of A'

Practically speaking: $n \times \alpha(n) \leq 4n$

Are these just mathematical games, freaks?



The inverse Ackermann function – in fact, $\Theta(n\,\alpha(n))$ arises in the seminal paper of

D. D. Sleator and R. E. Tarjan. *A data structure for dynamic trees*. Journal of Computer and System Sciences, 26(3):362-391, 1983.

Busy Beavers

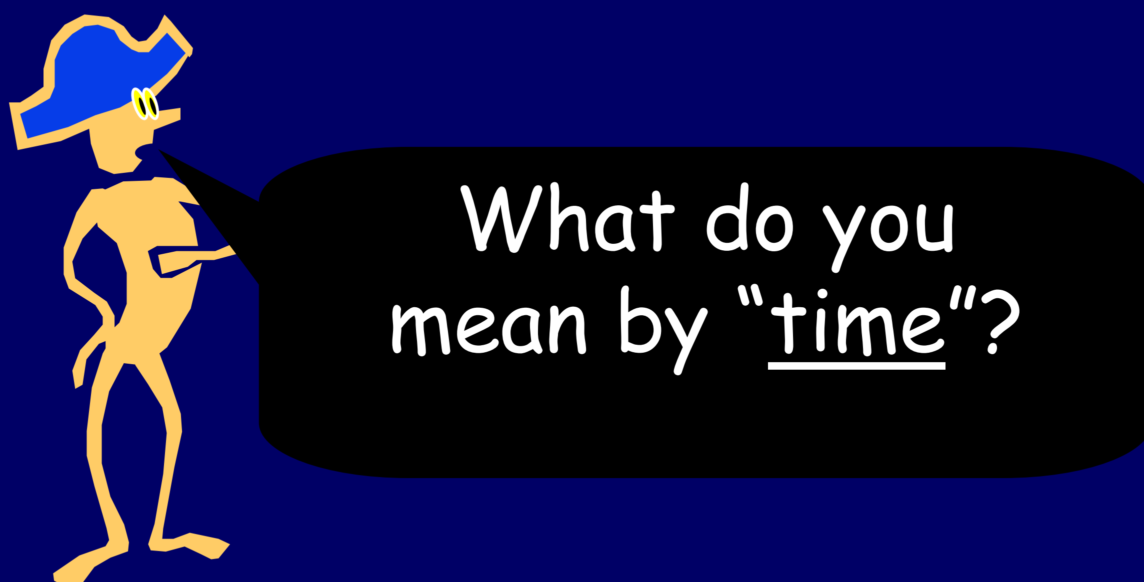
Near the end of the course we will define the BUSYBEAVER function: it will make Ackermann look like dust.


But we digress…
Let us get back
to the discussion about
“time” from the
beginning of
today’s class…

Time complexity of grade school addition

```
  * * * * * * * * * * *
    * * * * * * * * * *
+   * * * * * * * * * *
  ---------------------
  * * * * * * * * * * *
```

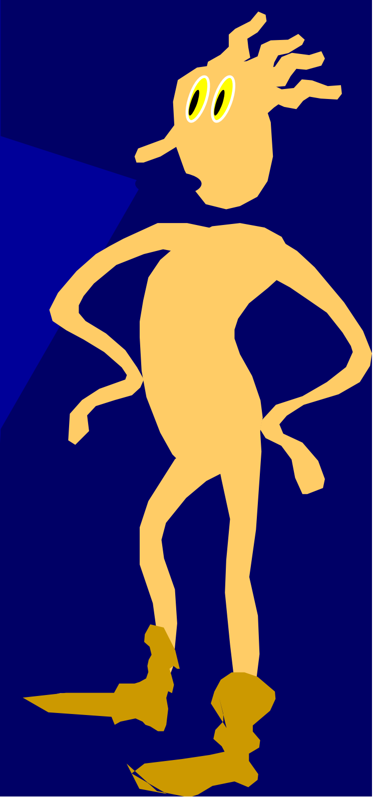
$T(n)$ = amount of time grade school addition uses to add two n-bit numbers

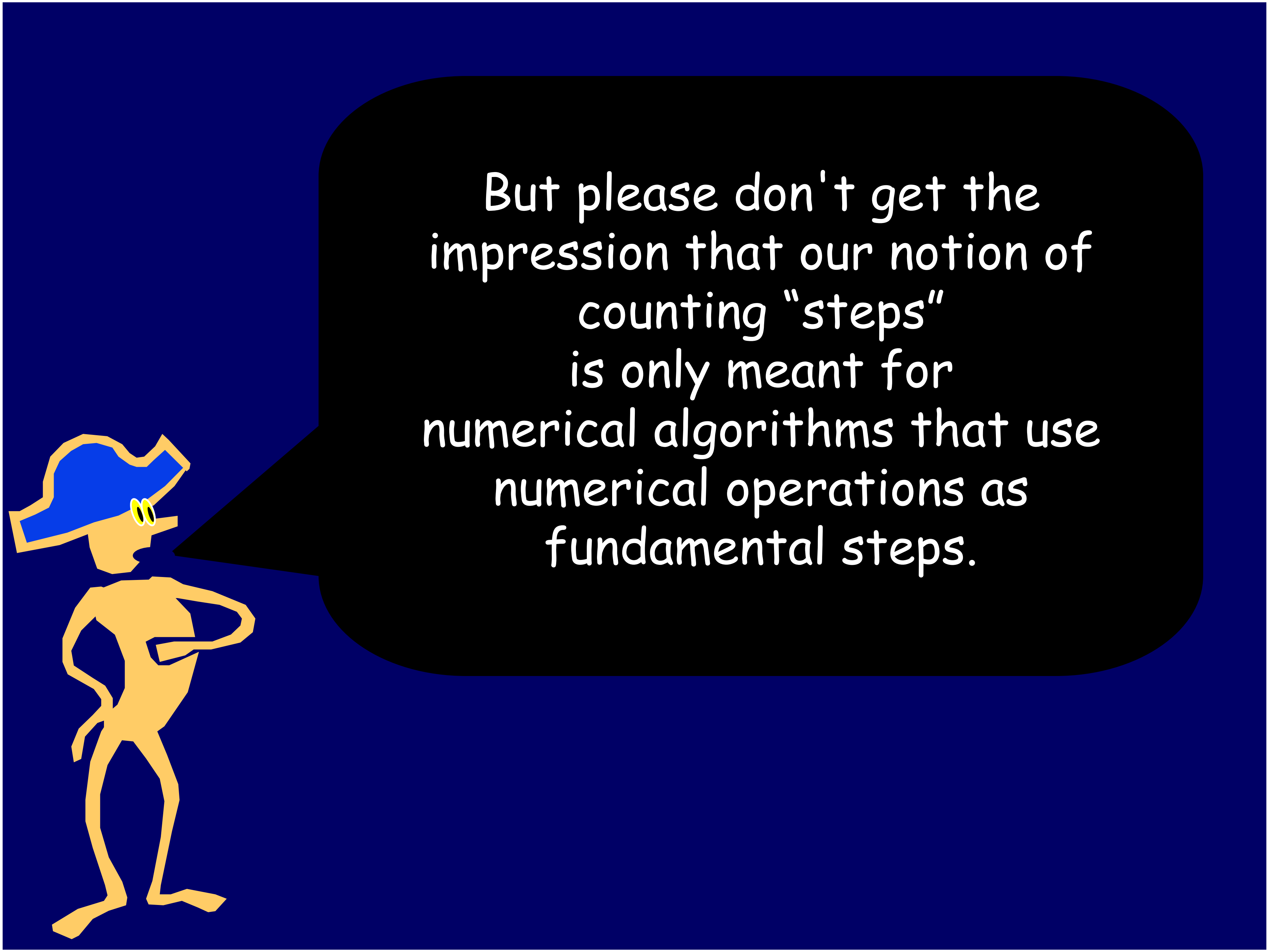


On any reasonable computer, adding 3 bits and writing down the two bit answer can be done in constant time.

Pick any particular computer A and define c to be the time it takes to perform  on that computer.

Total time to add two n-bit numbers using grade school addition: cn
[c time for each of n columns]

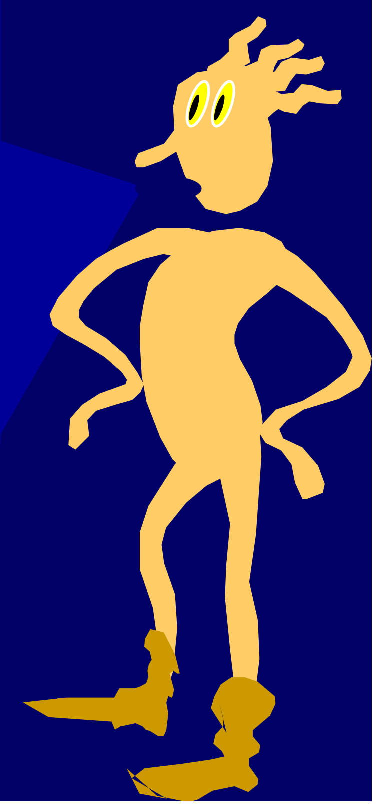



But please don't get the impression that our notion of counting “steps” is only meant for numerical algorithms that use numerical operations as fundamental steps.

Here is a general framework in which to reason about "time".

Suppose you want to evaluate the running time $T(n)$ of your favorite algorithm BOB.

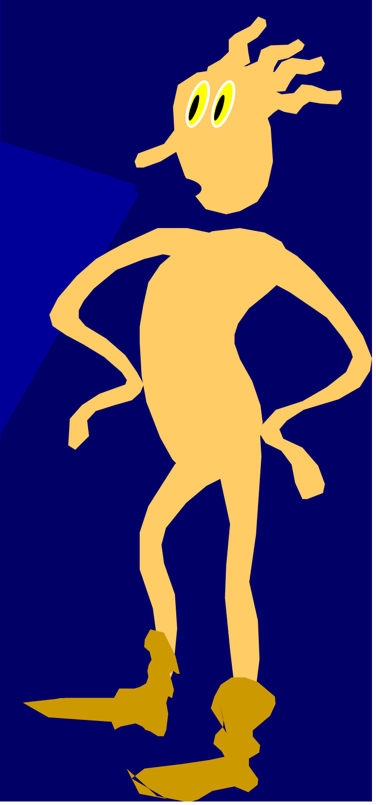
You want to ask:
how much "time" does BOB take when given an input X?

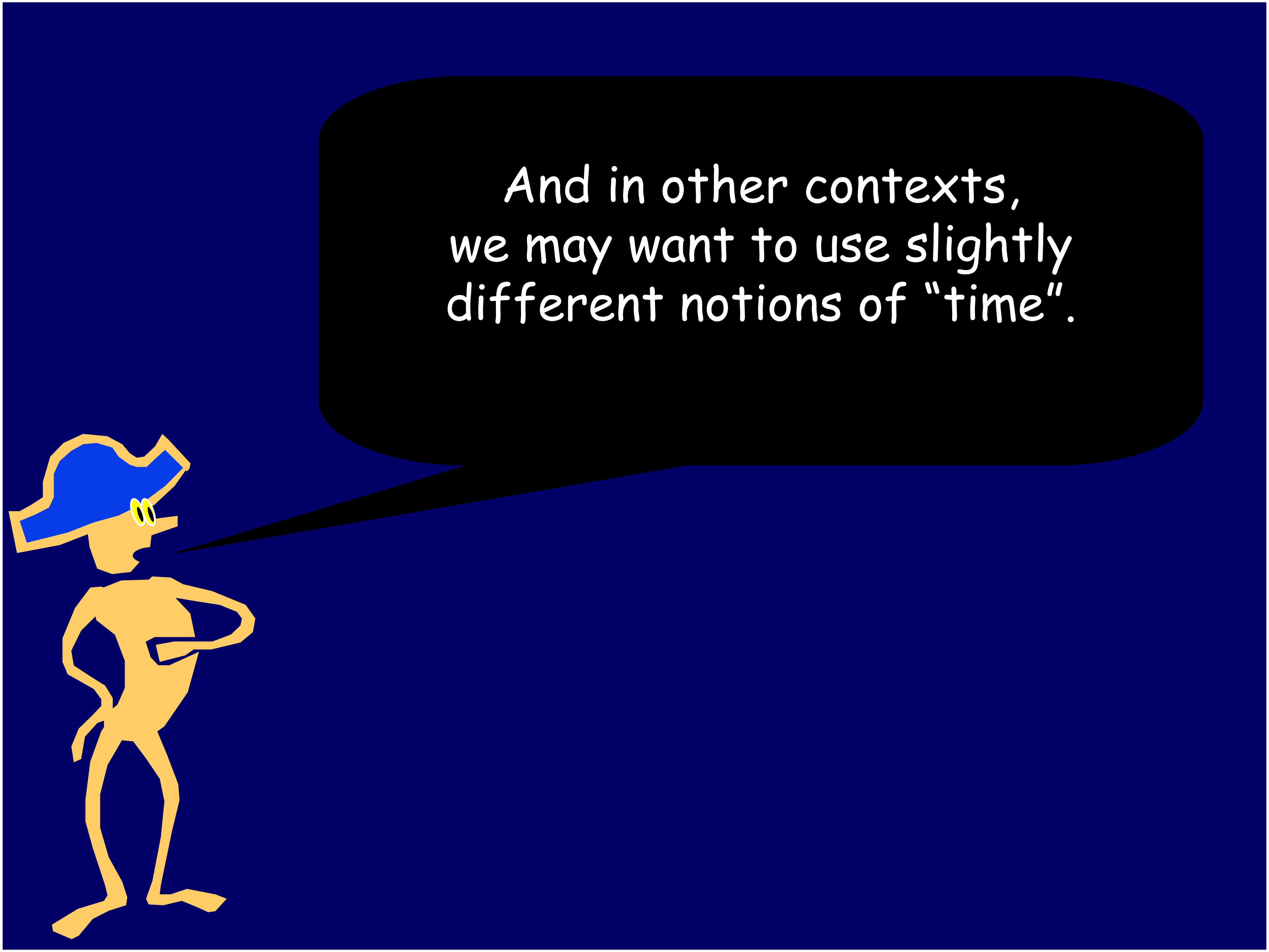


For concreteness, consider an implementation of the algorithm BOB in machine language for some processor.

Now, "time" can be measured as the number of instructions executed when given input X.

And $T(n)$ is the worst-case time on all permissible inputs of length n.

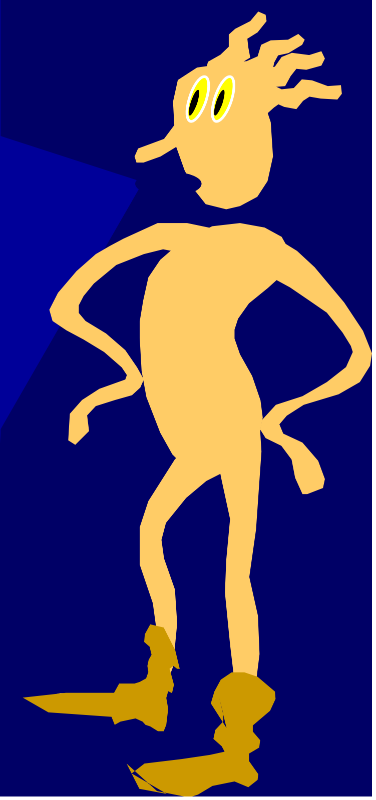



And in other contexts, we may want to use slightly different notions of “time”.

Sure.

You can measure “time” as
the number of elementary “steps”
defined in any other way,
provided each such “step”
takes <u>constant</u> time
in a reasonable implementation.

<u>Constant</u>: independent of the
length n of the input.



So instead, I can count the number of <u>JAVA bytecode</u> instructions executed when given the input X.

Or, when looking at grade school addition and multiplication, I can just count the number of additions

Time complexity of grade school addition

```
  * * * * * * * * * * *
+   * * * * * * * * * * *
    * * * * * * * * * * *
  -----------------------
  * * * * * * * * * * * *
```

T(n) = The amount of time grade school addition uses to add two n-bit numbers

We saw that T(n) was linear.

$T(n) = \Theta(n)$

Time complexity of grade school multiplication

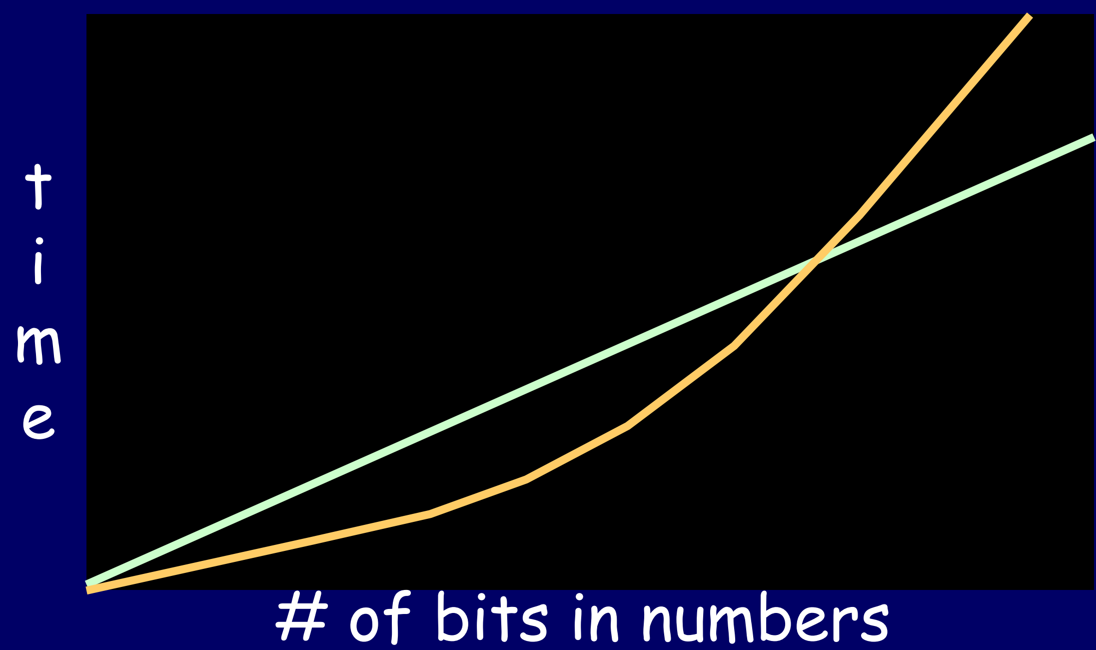
```
      X  * * * * * * * *
         * * * * * * * *
   ------------------------
         * * * * * * * *
       * * * * * * * *
     * * * * * * * *
n²  * * * * * * * *
   * * * * * * * *
  * * * * * * * *
 * * * * * * * *
* * * * * * * *
   ------------------------
* * * * * * * * * * * * * * * *
```

T(n) = The amount of time grade school multiplication uses to add two n-bit numbers

We saw that T(n) was quadratic.

$T(n) = \Theta(n^2)$

Grade School Addition: Linear time
Grade School Multiplication: Quadratic time



No matter how dramatic the difference in the constants the quadratic curve will eventually dominate the linear curve

Neat! We have demonstrated that as the inputs get large enough, multiplication is a harder problem than addition.

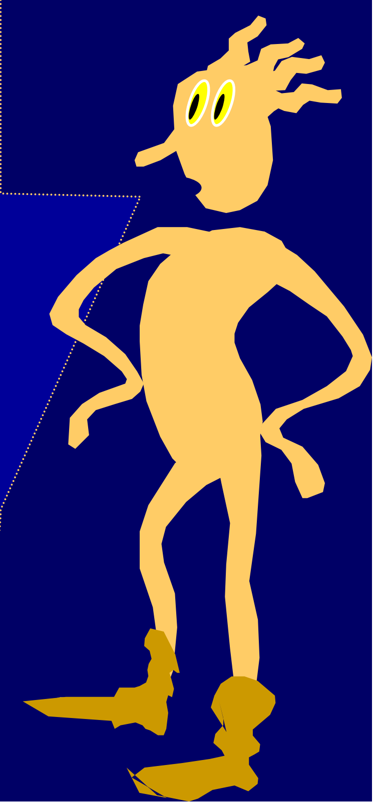
Mathematical confirmation of our common sense.

Is Bonzo correct?

Don't jump to conclusions!

We have argued that grade school multiplication uses more time than grade school addition. This is a comparison of the complexity of two <u>specific</u> algorithms.

To argue that multiplication is an inherently harder problem than addition we would have to show that "the best" addition algorithm is faster than "the best" multiplication algorithm.



Next Class

Will Bonzo be able to add two numbers faster than $\Theta(n)$?

Can Odette ever multiply two numbers faster than $\Theta(n^2)$?

Tune in on Thursday, same time, same place…