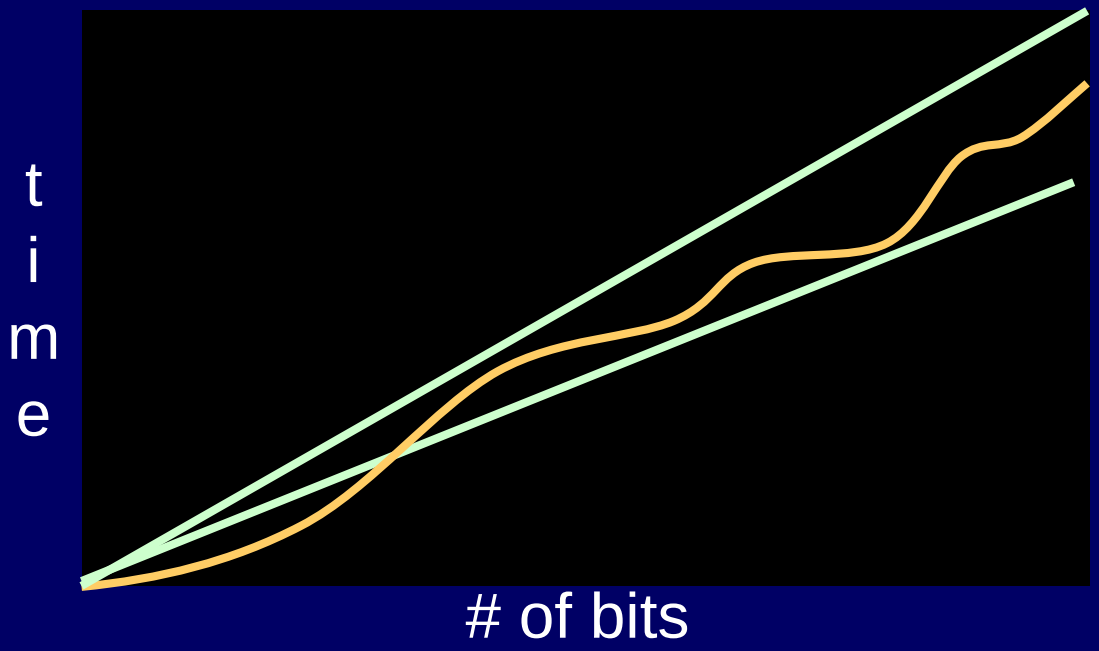
Great Theoretical Ideas In Computer Science

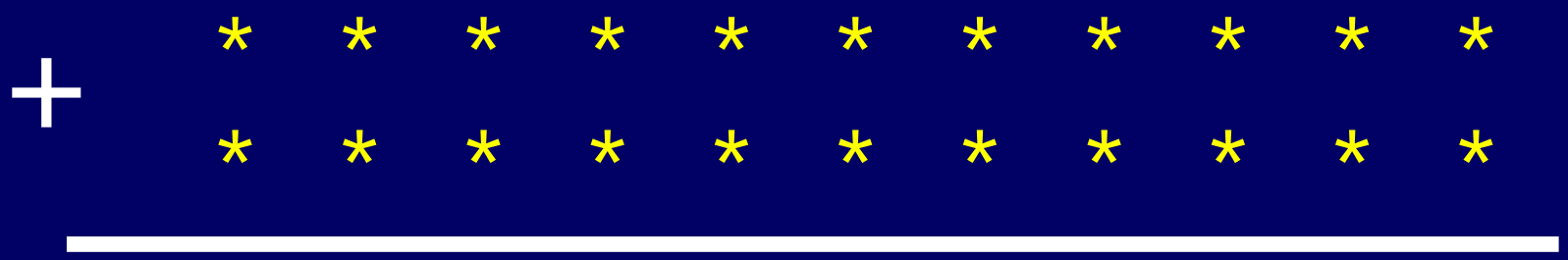
Steven Rudich CS 15-251 Spring 2004

Lecture 15 March 2, 2004 Carnegie Mellon University

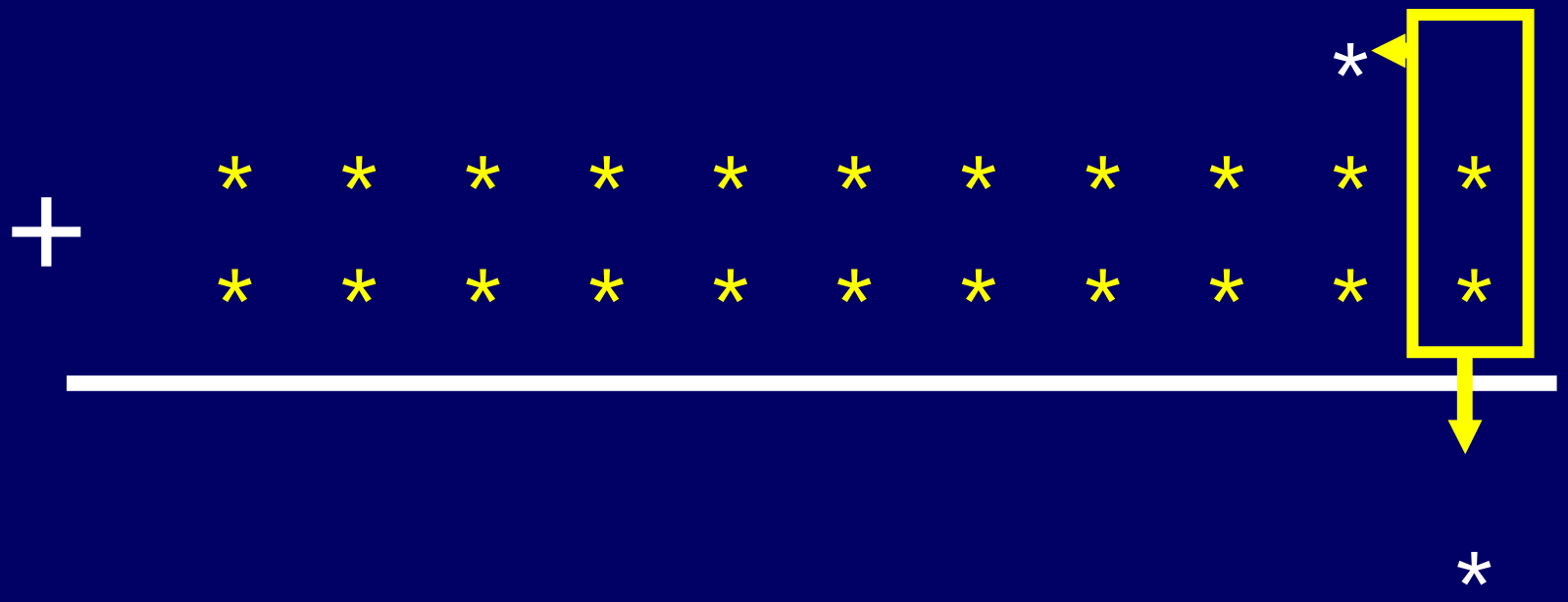
On Time Versus Input Size



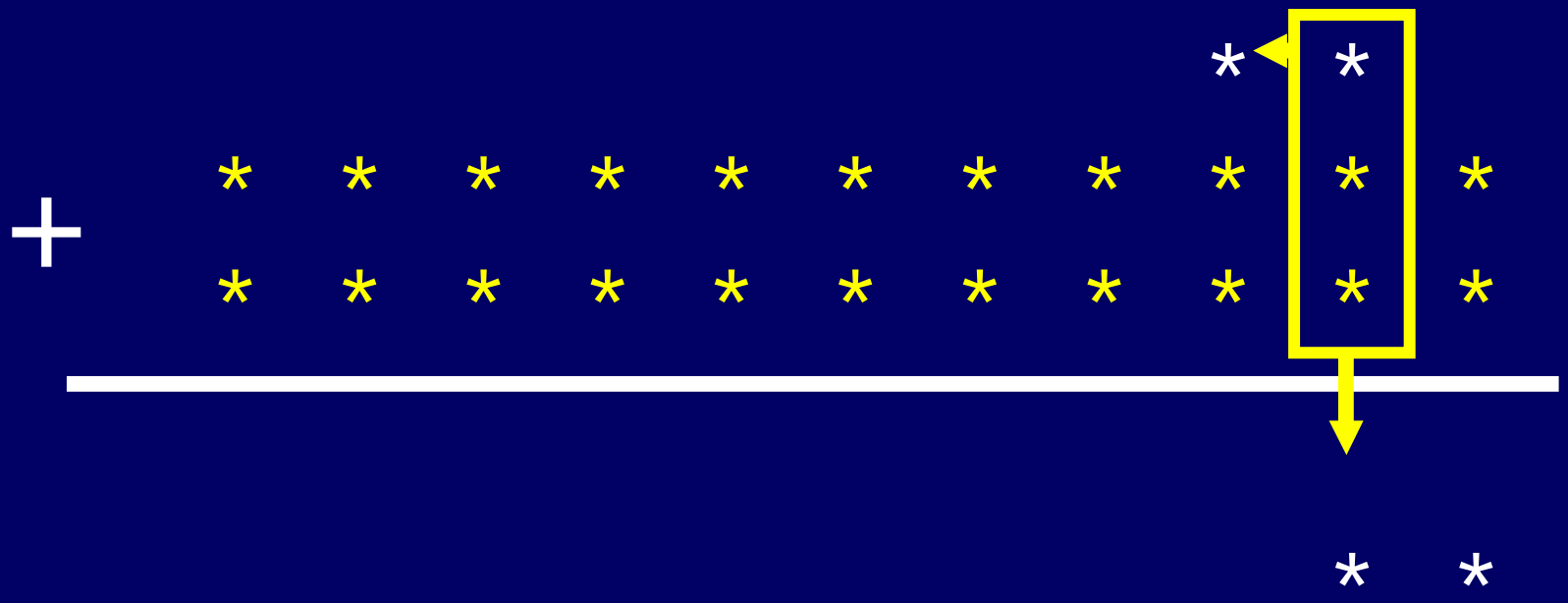
How to add 2 n-bit numbers.



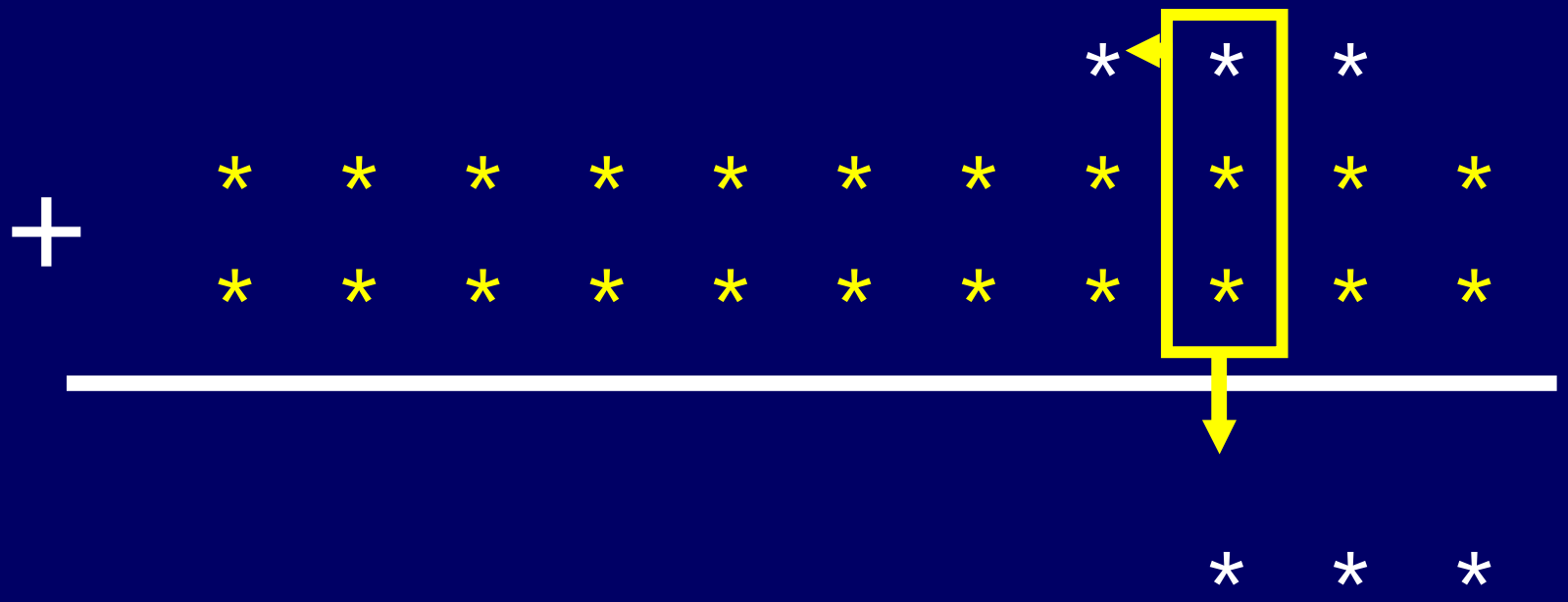
How to add 2 n-bit numbers.



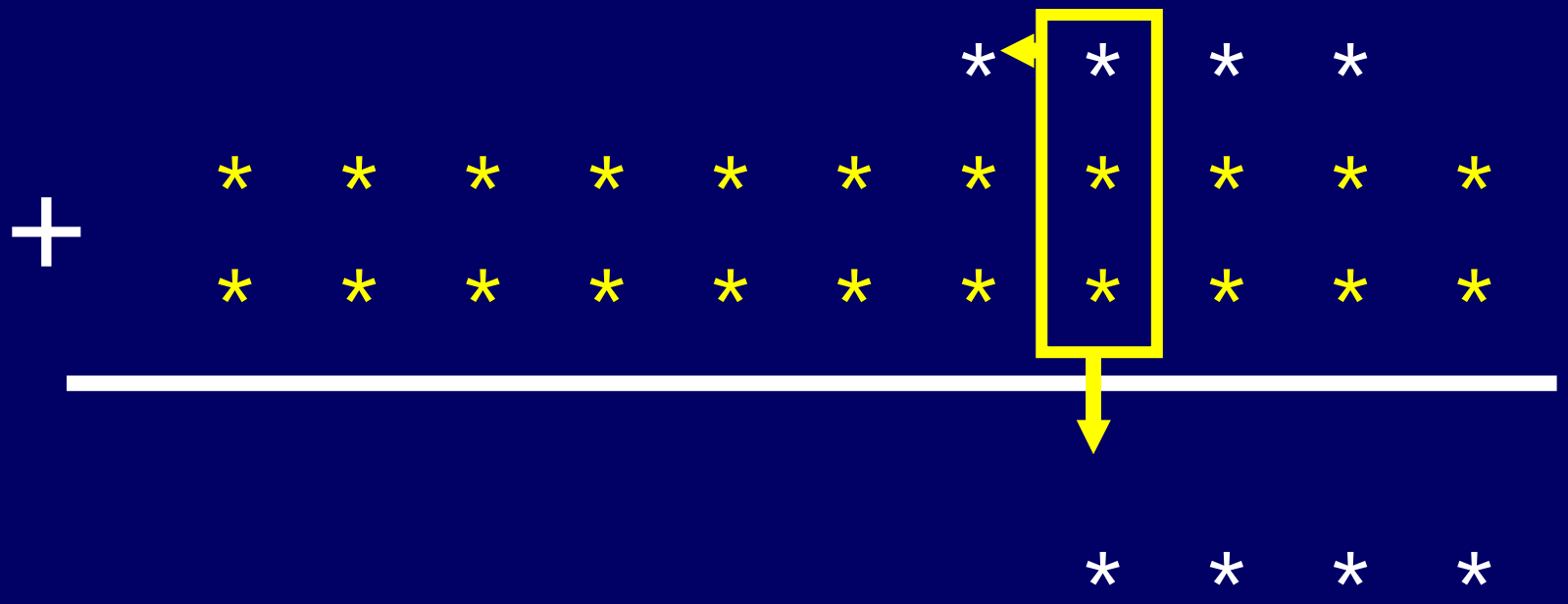
How to add 2 n-bit numbers.



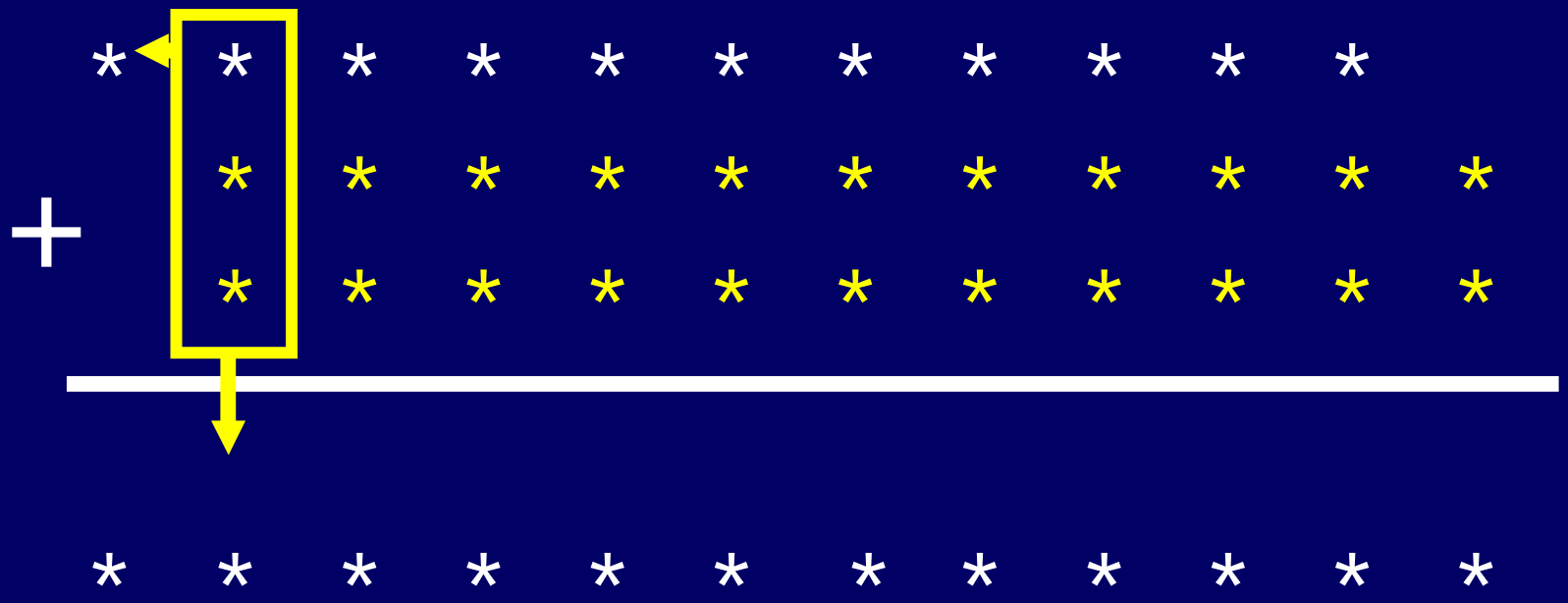
How to add 2 n-bit numbers.



How to add 2 n-bit numbers.



How to add 2 n-bit numbers.



Time complexity of grade school addition

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

What do you mean by “time”?

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

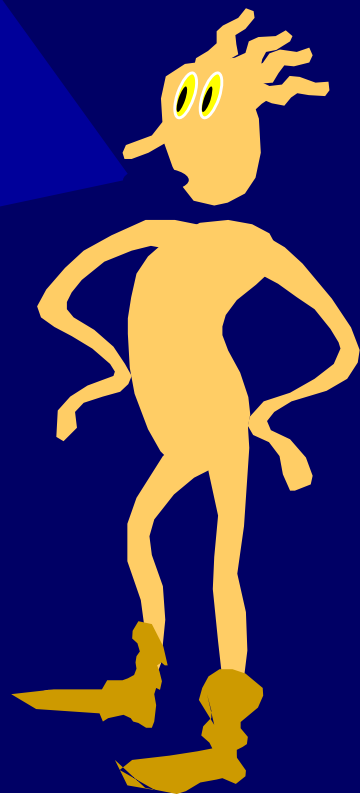
Our Goal

We want to define TIME in a sense that transcends implementation details and allows us to make assertions about grade school addition in a very general way.

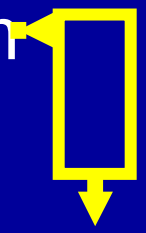
Hold on! You just admitted that it makes no sense to measure the time, T(n), taken by the method of grade school addition since the time depends on the implementation details. We will have to speak of the time taken by a particular implementation, as opposed to the time taken by the method in the abstract.

Don’t jump to conclusions!
Your objections are serious, but not insurmountable. There is a very nice sense in which we can analyze grade school addition without ever having to worry about implementation details.

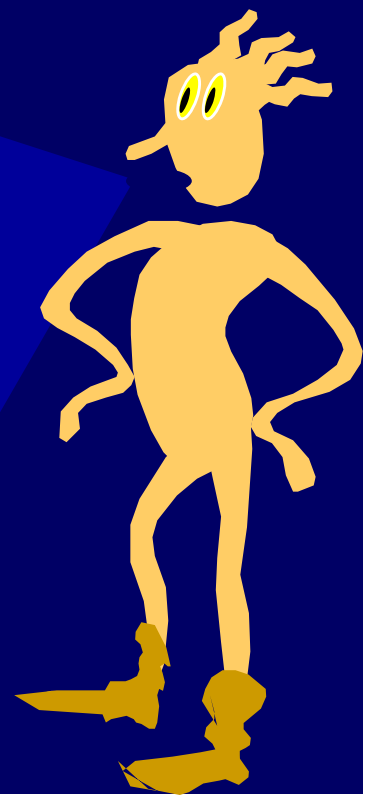
Here is how it works . . .



On any reasonable computer adding 3 bits and writing down the two bit answer can be done in constant time. Pick any particular computer A and define c to be the time it takes to perform on that computer.

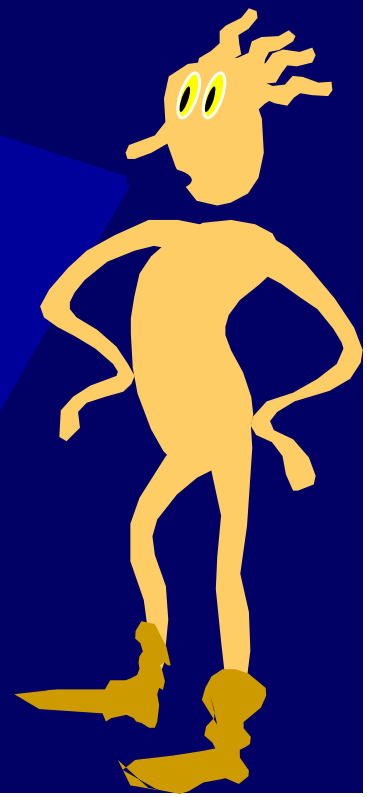


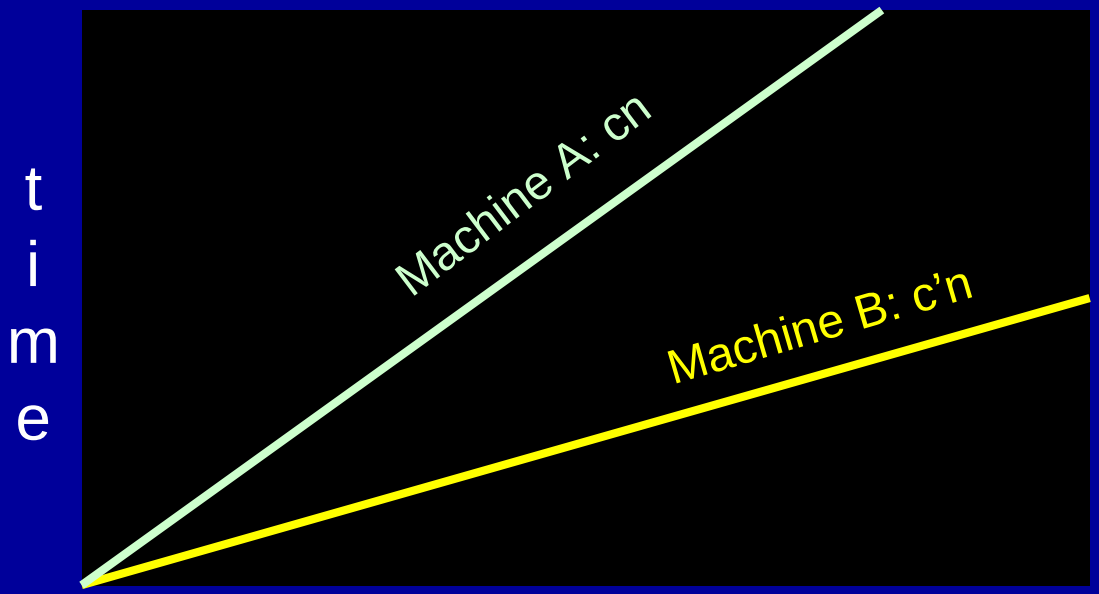
Total time to add two n-bit numbers using grade school addition: cn
[c time for each of n columns]



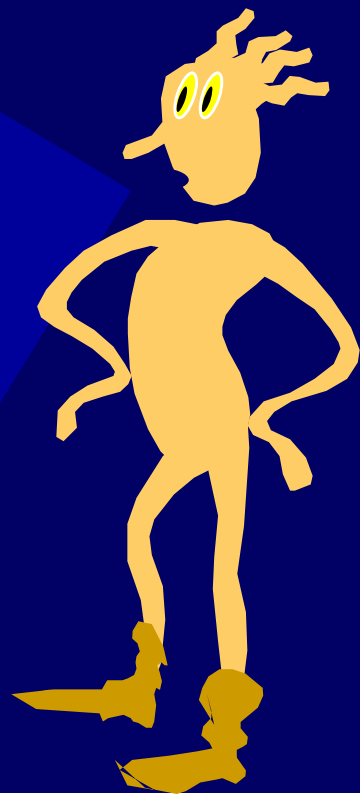
Implemented on another computer B the running time will be c'n where c' is the time it takes to perform on that computer.

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



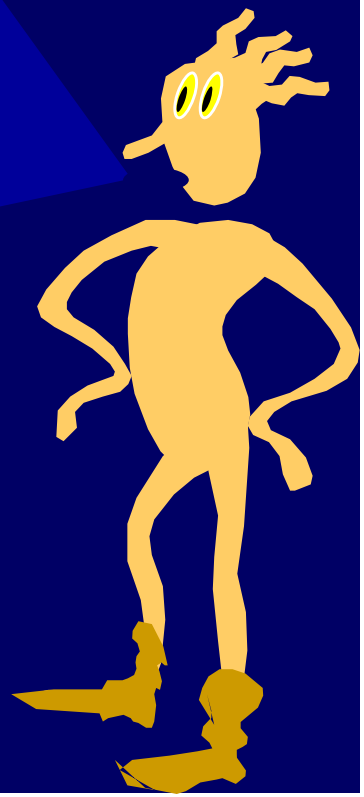


The fact that we get a line is invariant under changes of implementations. Different machines result in different slopes, but time grows linearly as input size increases.



Thus we arrive at an implementation independent insight: Grade School Addition is a linear time algorithm.

Determining the growth rate of the resource curve as the problem size increases is one of the fundamental ideas of computer science.



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..

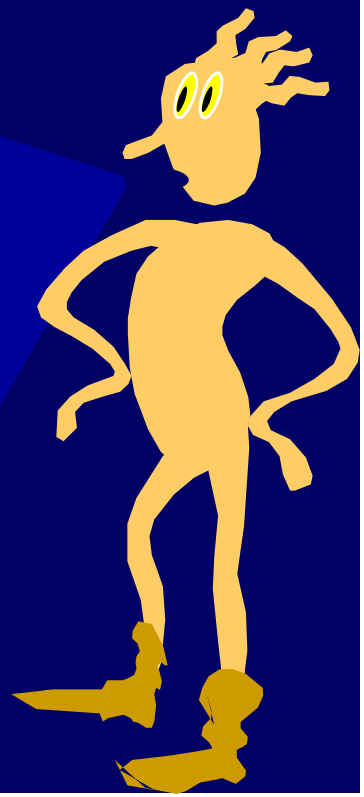
TIME vs INPUT SIZE

For any algorithm, define
INPUT SIZE = # of bits to specify inputs,

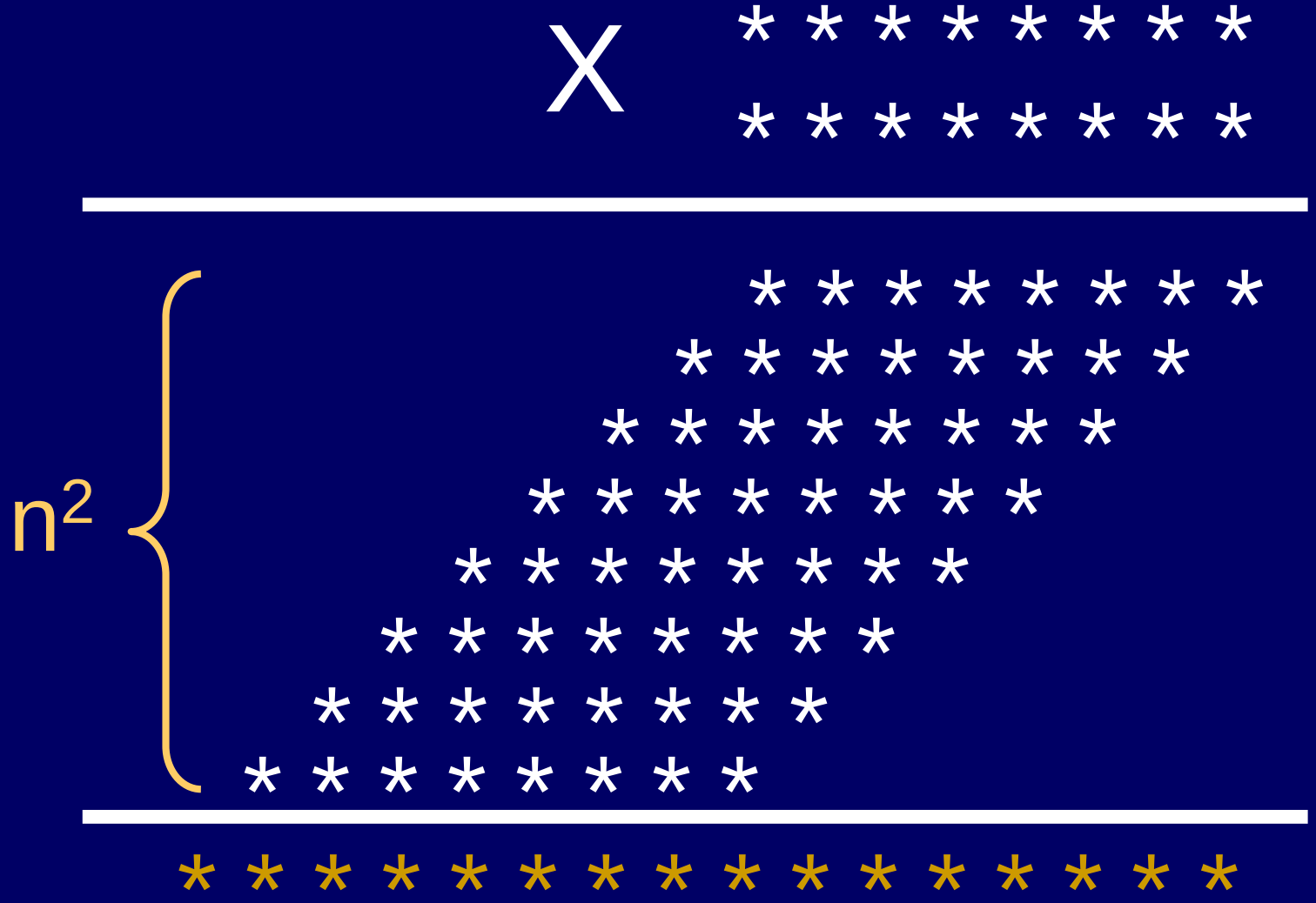
Define
$TIME_n$ = the worst-case amount of time used on inputs of size n.

We Often Ask:

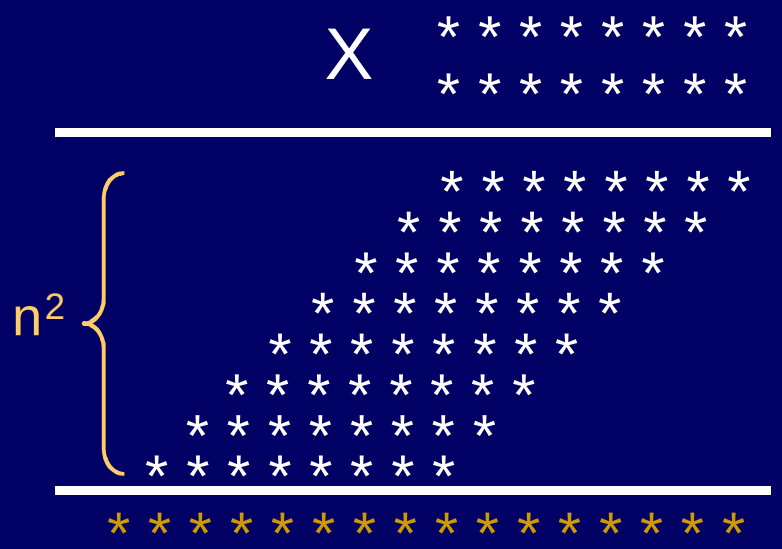
What is the GROWTH RATE of $Time_n$?



How to multiply 2 n-bit numbers.

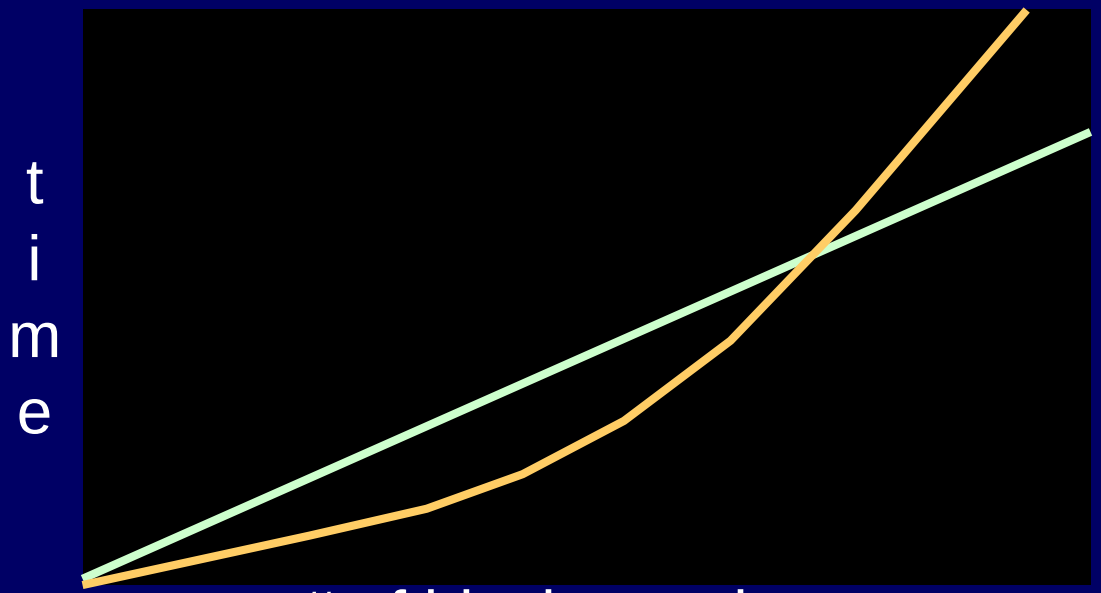


How to multiply 2 n-bit numbers.



I get it!
The total time is bounded by cn^2.

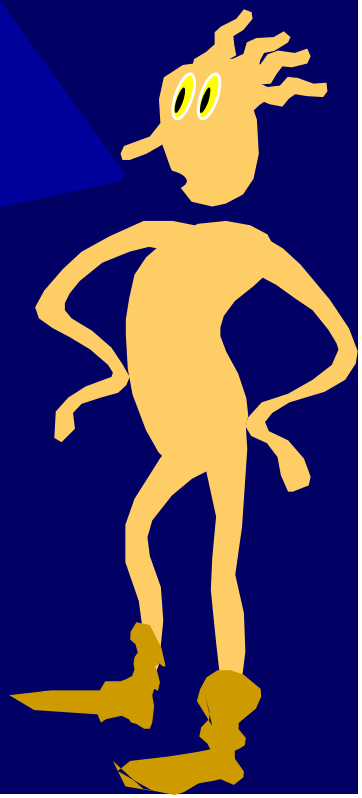
Grade School Addition: Linear time
Grade School Multiplication: Quadratic time



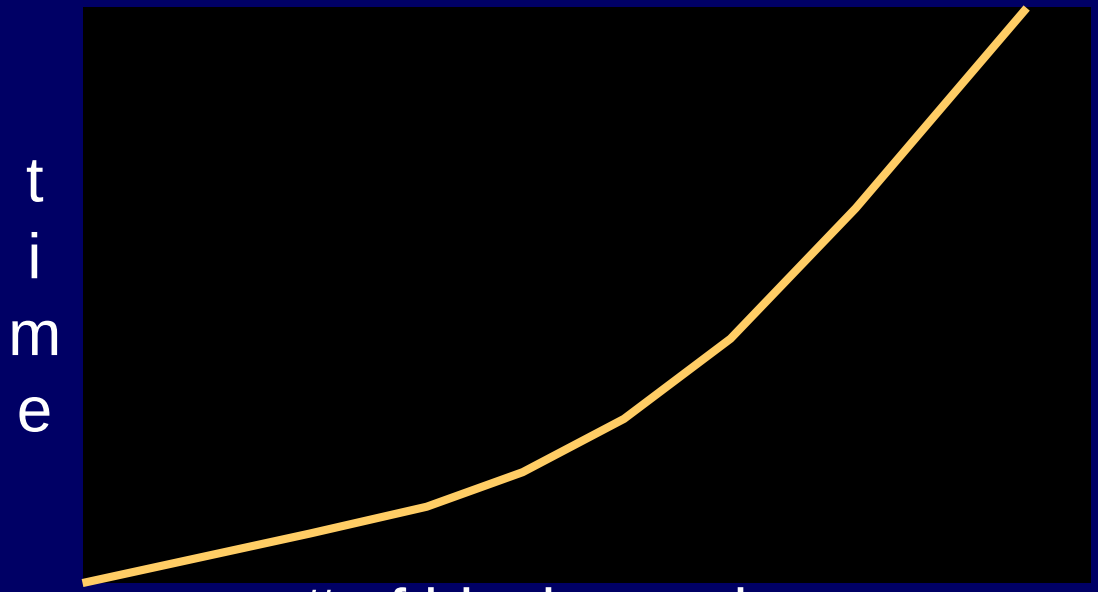
No matter how dramatic the difference in the constants the quadratic curve will eventually dominate the linear curve

Ok, so…

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

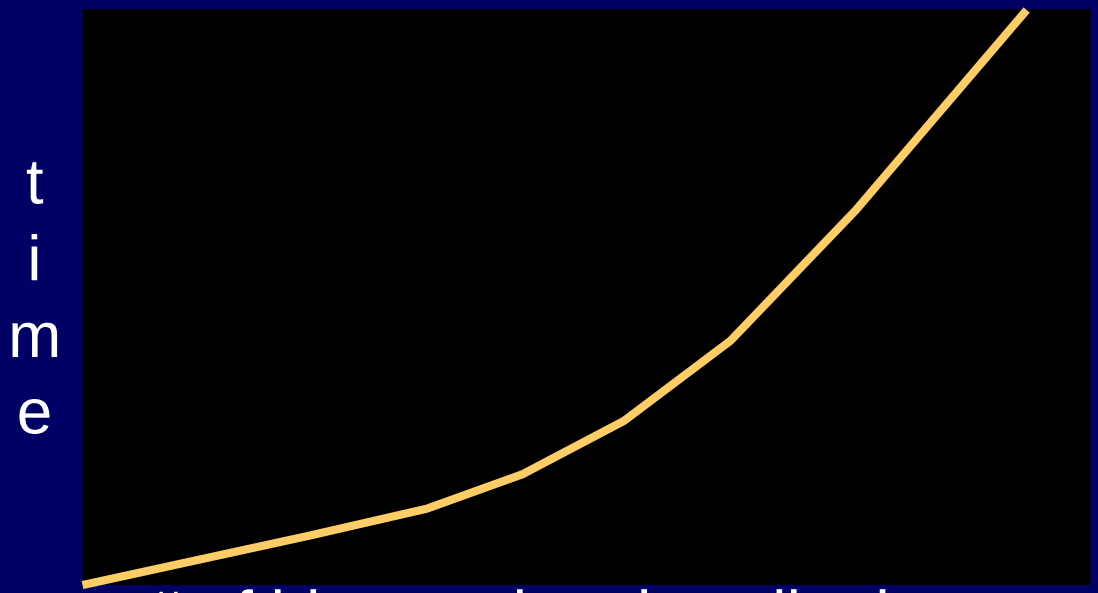


Grade School Multiplication: Quadratic time



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

Time Versus Input Size



Input size is measured in bits, unless we say otherwise.

How much time does it take?

Nursery School Addition

INPUT: Two n-bit numbers, a and b

OUTPUT: a + b

Start at a and add 1, b times

T(n) = ?

What is T(n) ?

Nursery School Addition

INPUT: Two n-bit numbers, a and b

OUTPUT: a + b

Start at a and add 1, b times

If b=000.0000, then NSA takes almost no time. If b = 111111.11, then NSA takes c $n2^n$ time.

What is T(n) ?

Nursery School Addition

INPUT: Two n-bit numbers, a and b

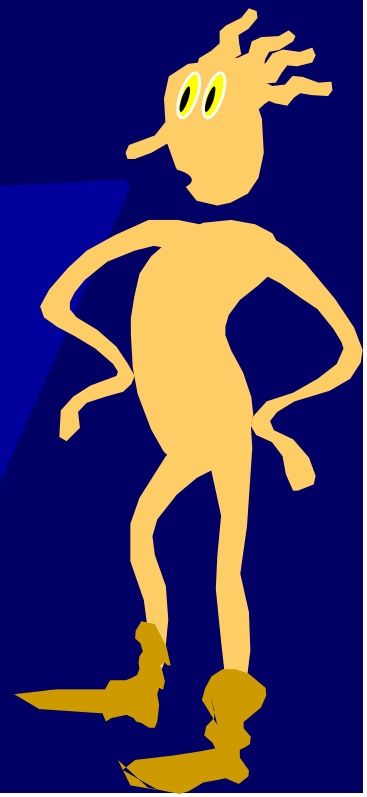
OUTPUT: a + b

Start at a and add 1, b times

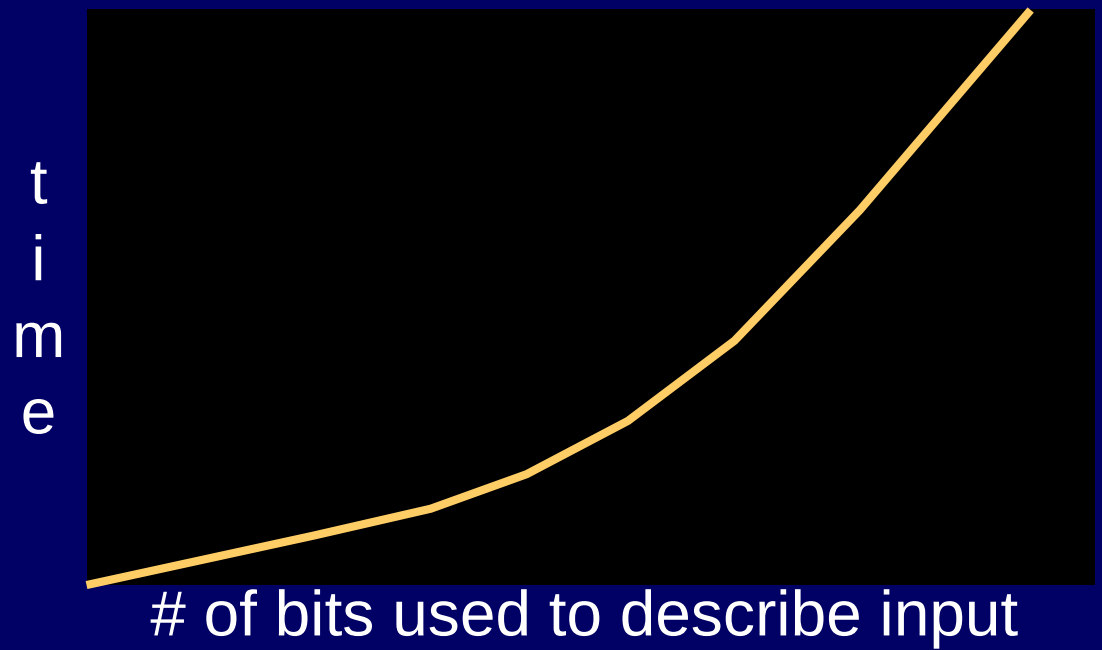
Worst case time is $c\, n2^n$ ß Exponential!

Worst-case Time T(n) for algorithm A means that we define a measure of input size n, and we define

$$T(n) = \max_{\text{all permissible inputs } X \text{ of size } n} \text{running time of algorithm A on } X.$$



Worst-case Time Versus Input Size



Worst Case Time Complexity

What is T(n)?

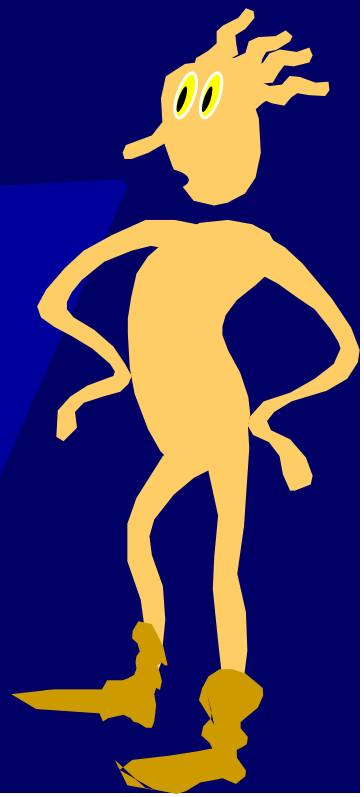
Kindergarden Multiplication
INPUT: Two n-bit numbers, a and b
OUTPUT: a * b

Start at a and add a, b-1 times

We always pick the WORST CASE for the input size n.
Thus, $T(n) = c\, n2^n$ ß Exponential

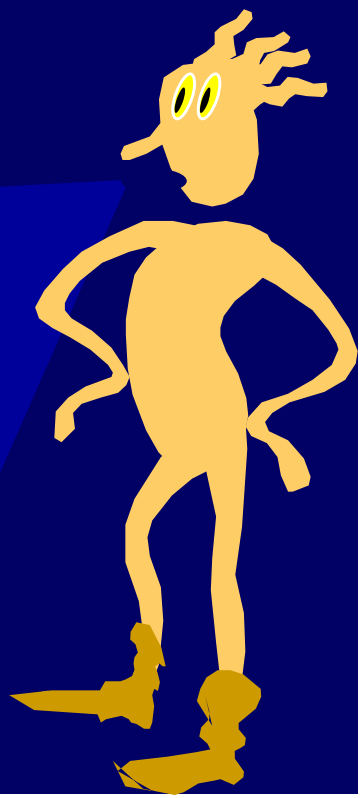
Thus, Nursery School adding and multiplication are exponential time. They SCALE HORRIBLY as input size grows.

Grade school methods scale polynomially – just linear and quadratic. Thus, we can add and multiply fairly large numbers.



Multiplication is efficient, what about reverse multiplication?

Let’s define FACTORING n to any method to produce a non-trivial factor of n, or to assert that n is prime.



Factoring The Number n By Trial Division

Trial division up to $\sqrt{n}$

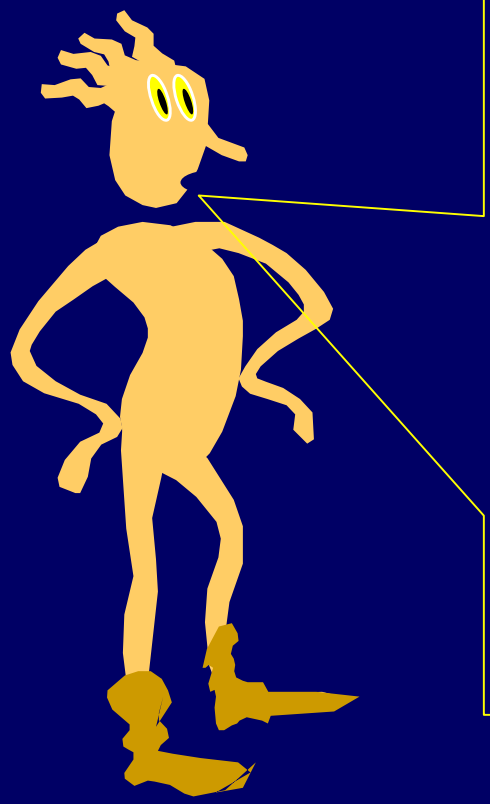
```
for k = 2 to √n do
  if k | n then
    return n "has a non-trivial factor" k

return n "is prime"
```

$O(\sqrt{n}\,(\log n)^2)$ time if division is $O((\log n)^2)$

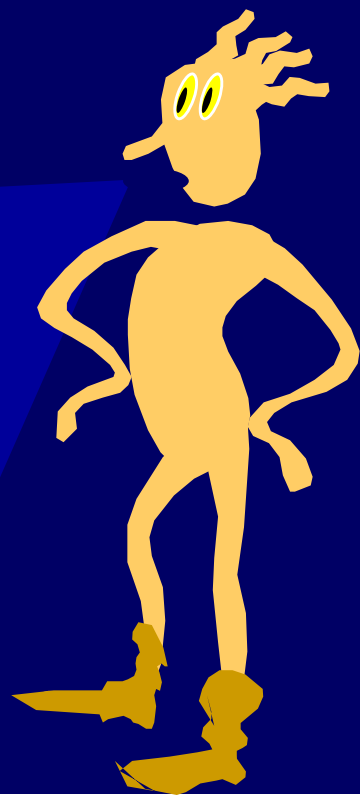
On input n, trial factoring uses $O(\sqrt{n}\,(\log n)^2)$ time. Is that efficient?

No! The input length is log(n). Let k = log n. In terms of k, we are using $2^{k/2}\,k^2$ time.

The time is EXPONENTIAL in the input length.



We know methods of FACTORING that are sub-exponential (about $2^{\text{cube root of k}}$), but nothing efficient.



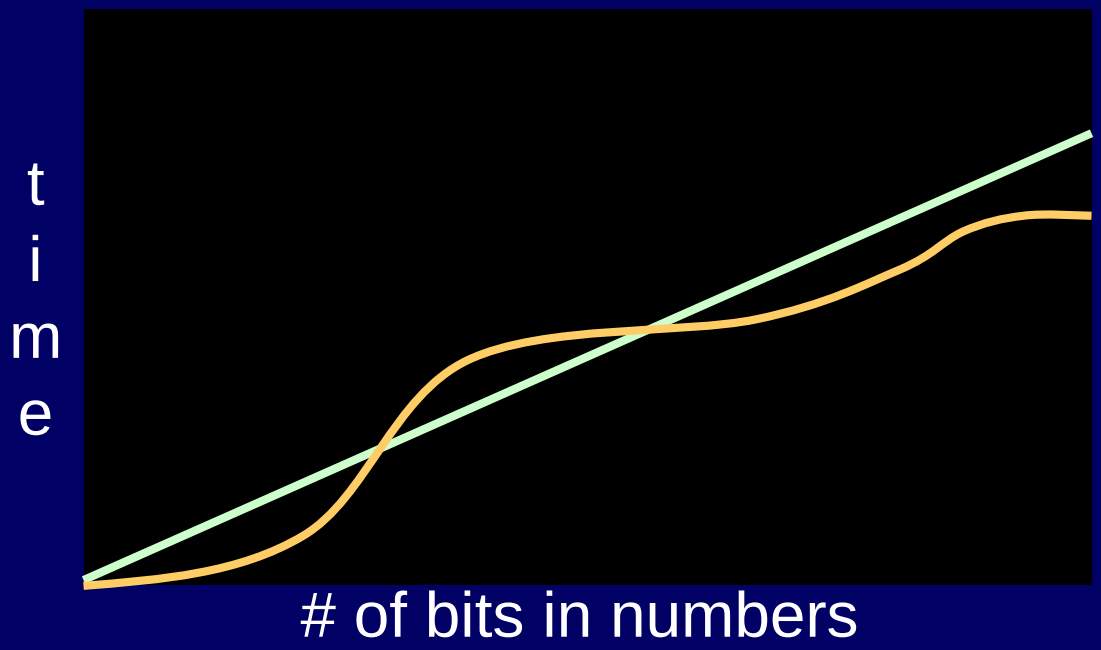
Useful notation to discuss growth rates

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

[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



Useful notation to discuss growth rates

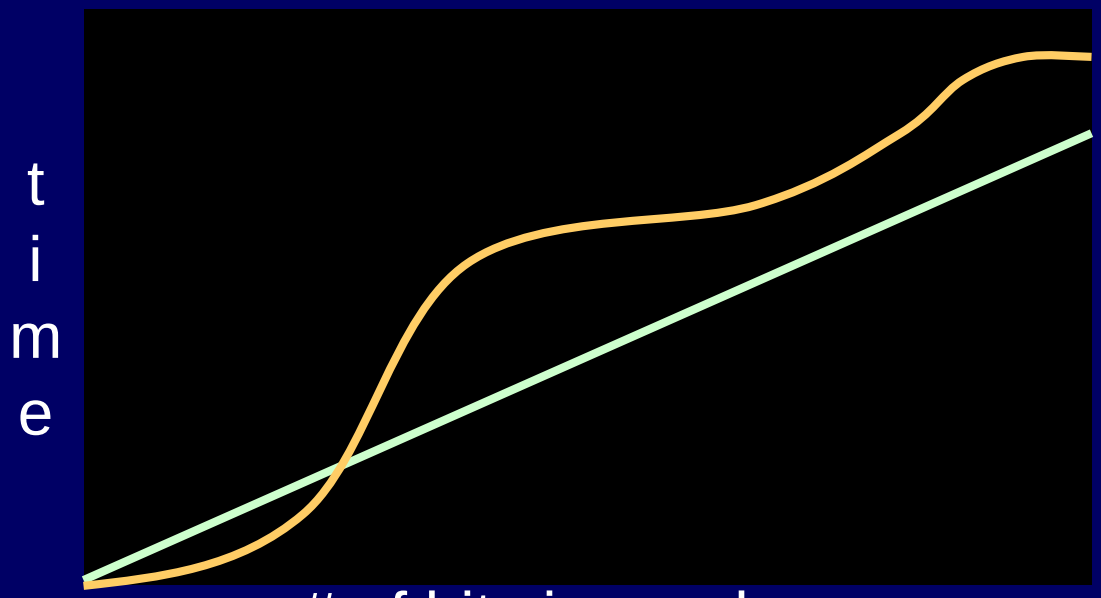
For any monotonic function f from the positive integers to the positive integers, we say "f = $\Omega(n)$" or "f is $\Omega(n)$"

if:

f eventually dominates some constant times n

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

f = Ω (n) means that there is a line that can be drawn that stays below f from some point on



Useful notation to discuss growth rates

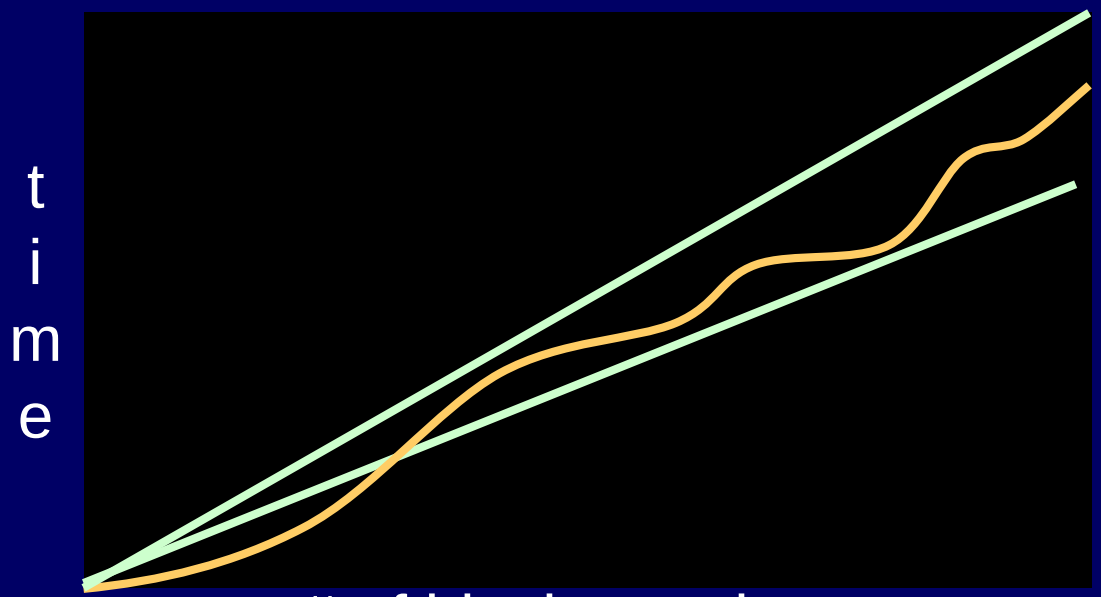
For any monotonic function f from the positive integers to the positive integers, we say

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

if:

$f = O(n)$ and $f = \Omega(n)$

$f = \Theta(n)$ means that f can be sandwiched between two lines



Useful notation to discuss growth rates

For any 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

[There exists a constant c such that for all sufficiently large n: f(n) <= cg(n)]

Useful notation to discuss growth rates

For any 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

[There exists a constant c such that for all sufficiently large n: f(n) >= cg(n)]

Useful notation to discuss growth rates

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

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

if:

$f = O(g)$ and $f = \Omega(g)$

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
- $n^2 \log n = \Theta(n^2)$
  - NO

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

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

Names For Some Growth Rates

Exponential Time means that for some constant k, $T(n) = O(k^n)$

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

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

Names For Some Growth Rates

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

Polylogarithmic Time means that 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

Doubly Exponential Time means that for some constant k, T(n) = is 2 to the 2^n

Triply Exponential.

And so forth.

2STACK

$2STACK(0) = 1$

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

$2STACK(1) = 2$
$2STACK(2) = 4$
$2STACK(3) = 16$
$2STACK(4) = 65536$
$2STACK(5) \geq 10^{80}$
= atoms in universe

$\log^*(n)$ = Inverse 2STACK(n)

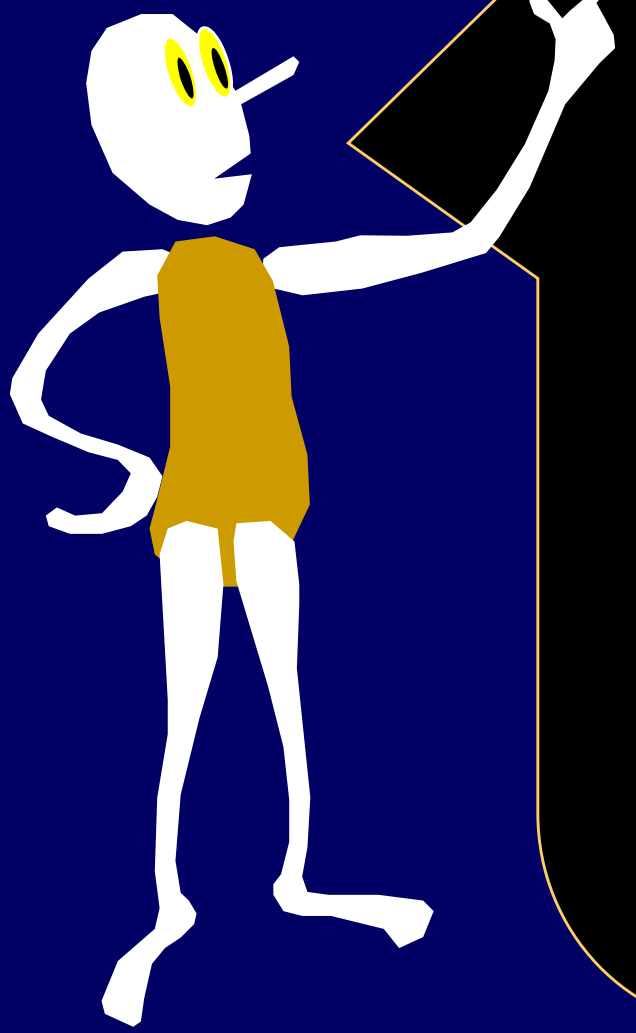
of times you have to apply the log function to n to get it ito 1

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

$Log^*(1) = 0$
$Log^*(2) = 1$
$Log^*(4) = 2$
$Log^*(16) = 3$
$Log^*(65536) = 4$
$Log^*(way\ big) = 5$



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 in this universe

Inverse Ackermann

$A(0, n) = n + 1 \quad \text{for } n \geq 0$

$A(m, 0) = A(m - 1, 1) \quad \text{for } m \geq 1$

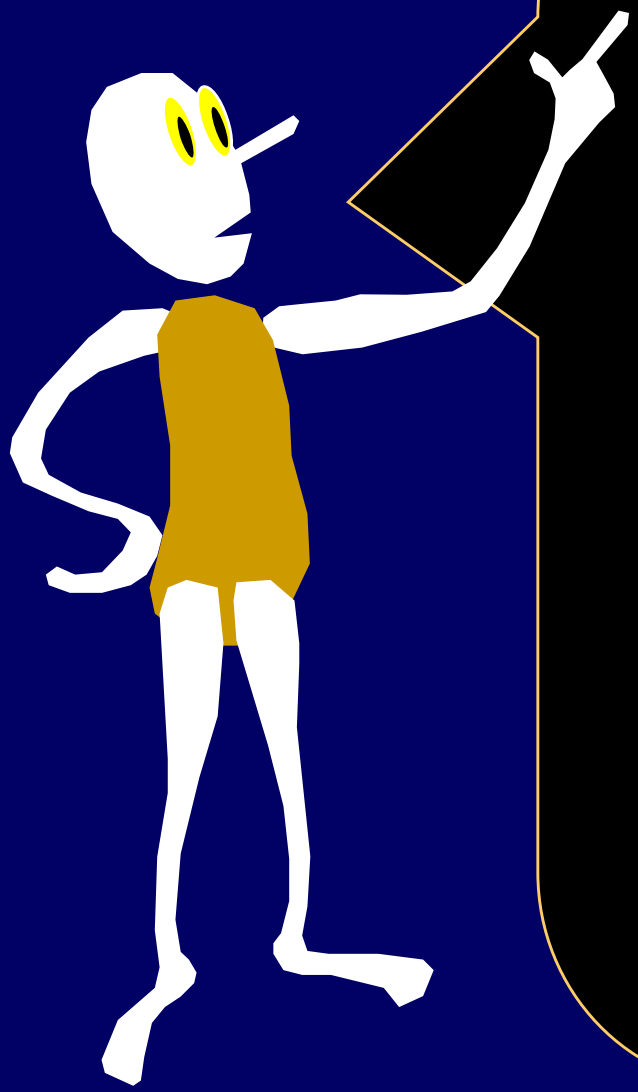
$A(m, n) = A(m - 1, A(m, n - 1)) \text{ for } m, n \geq 1$

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

Inverse Ackerman is the inverse of A'

Practically speaking:

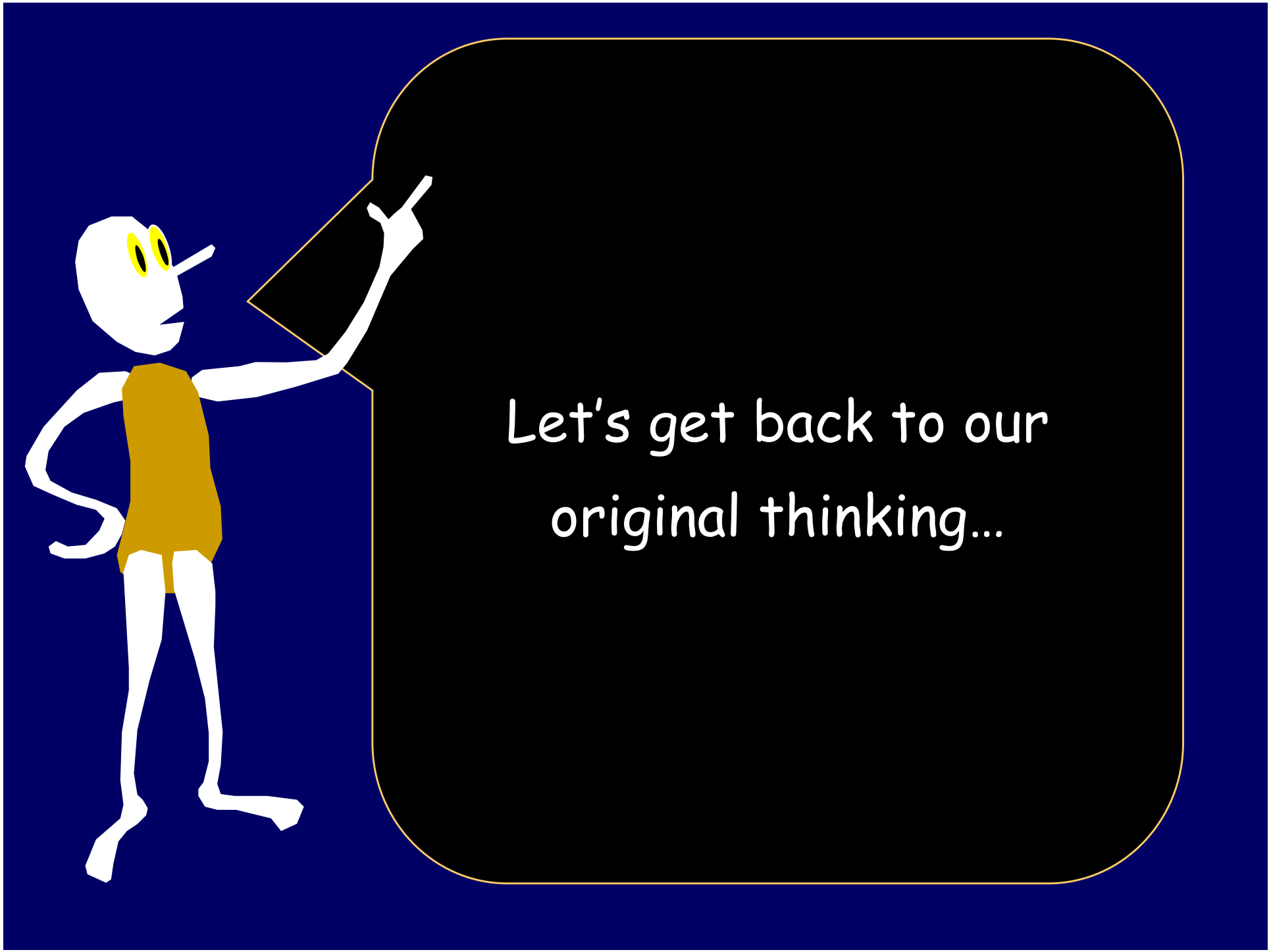
$$n^{*}\text{inverse-Ackermann}(n) \leq 4n$$



n*inverse-Ackermann(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.


One the last day of this course we will define the BUSYBEAVER function – it makes Akermann look like dust.

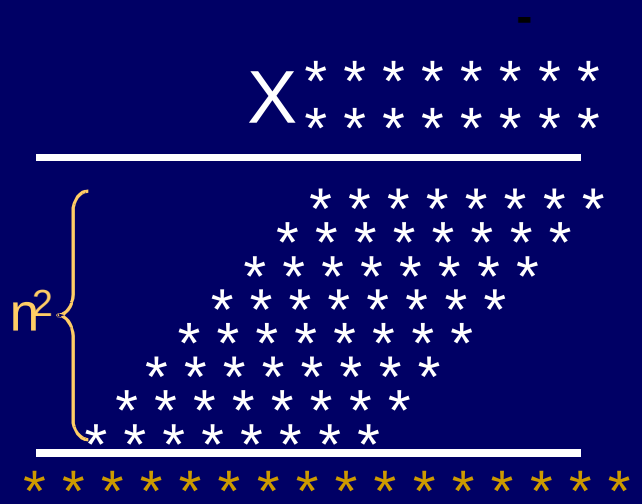

Let's get back to our original thinking...

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.

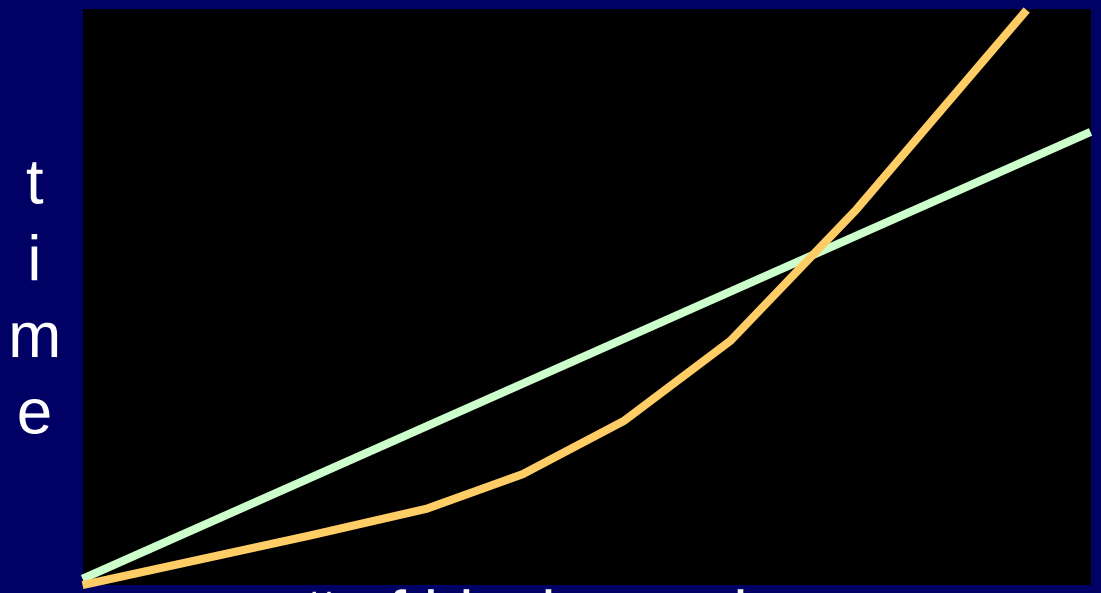
Time complexity of grade school multiplication



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

We saw that T(n) was quadratic.

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

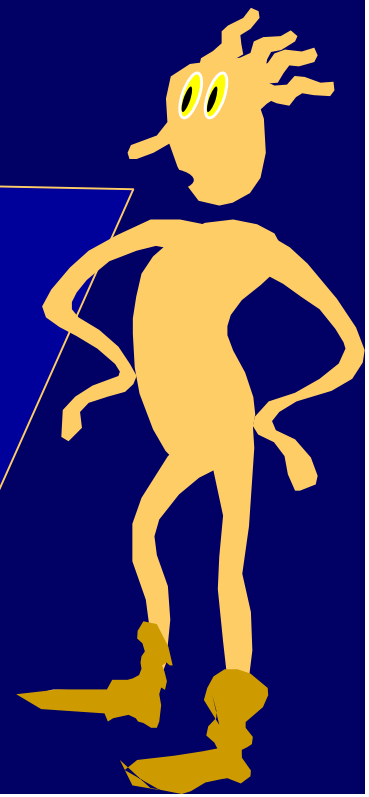
Neat! We have demonstrated that as things scale multiplication is a harder problem than addition.

Mathematical confirmation of our common sense.

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 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.



Grade school addition is linear time.

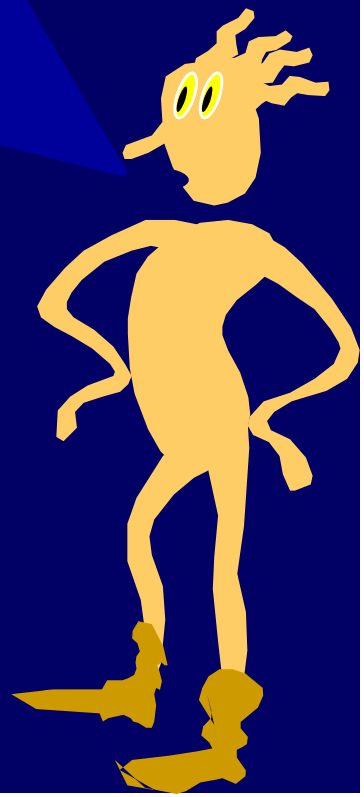
Is there a sub-linear time for addition?

Any algorithm for addition must read all of the input bits

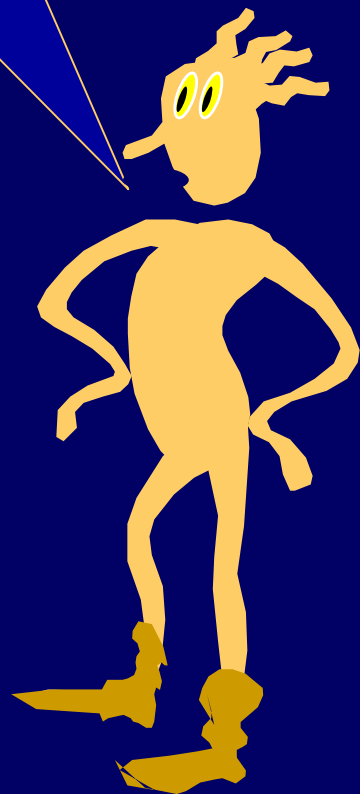
- Suppose there is a mystery algorithm A that does not examine each bit
- Give A a pair of numbers. There must be some unexamined bit position i in one of the numbers
- If the A is not correct on the numbers, we found a bug
- If A is correct, flip the bit at position i and give A the new pair of numbers. A give the same answer as before, which is now wrong.

So any algorithm for addition must use time at least linear in the size of the numbers.

Grade school addition can't be improved upon by more than a constant factor.

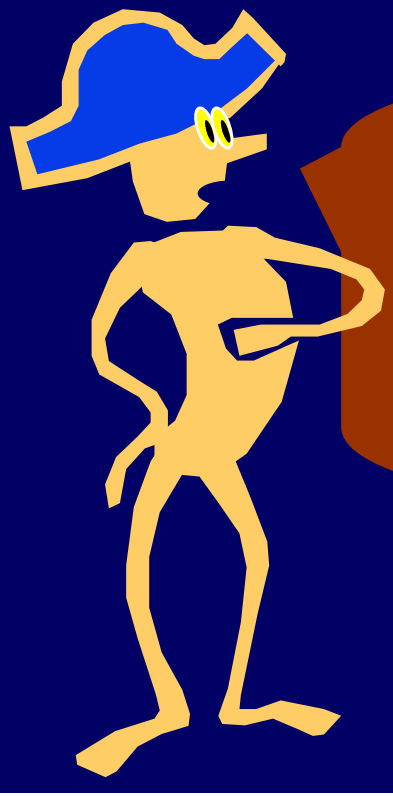


To argue that multiplication is an inherently harder problem than addition we would have to show that no possible multiplication algorithm runs in linear time.



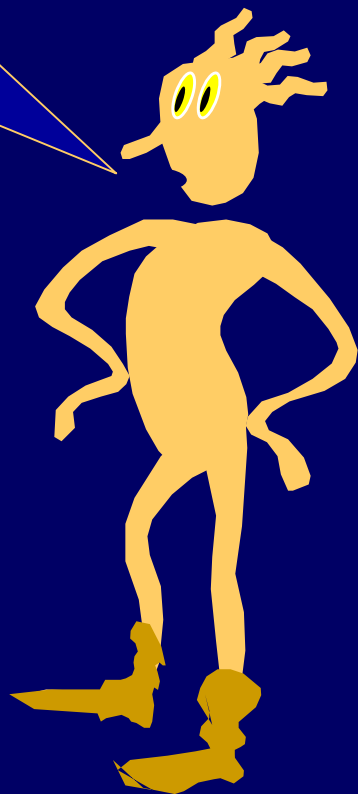
Grade School Addition: $\Theta(n)$ time
Furthermore, it is optimal

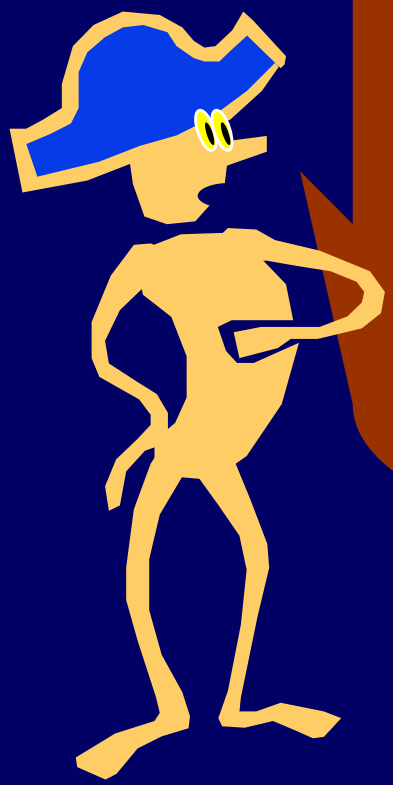
Grade School Multiplication: $\Theta(n^2)$ time



Is there a clever algorithm to multiply two numbers in linear time?

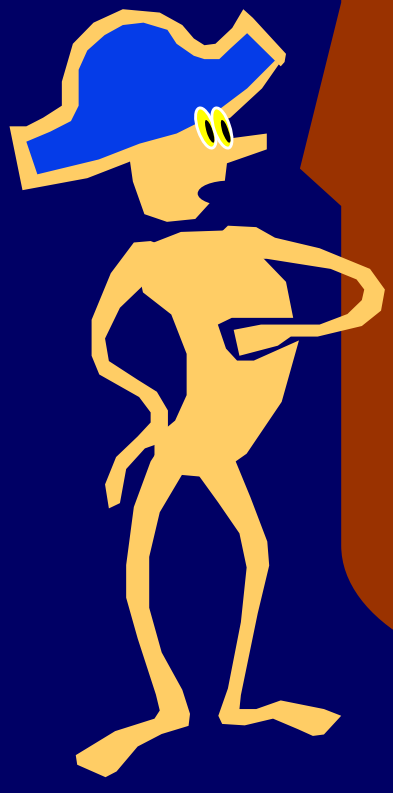
Despite years of research, no one knows! If you resolve this question, Carnegie Mellon will give you a PhD!





Can we even brake the quadratic time barrier – in other words can we do something very different than grade school multiplication?

Grade School Multiplication: $\Theta(n^2)$ time Kissing Intuition



Intuition: Let's say that each time an algorithm has to multiply a digit from one number with a digit from the other number, we call that a "kiss". It seems as if any correct algorithm must kiss at least n^2 times.