

# 15-251

## Great Theoretical Ideas in Computer Science

## This is The Big Oh!

Lecture 21 (November 3, 2009)



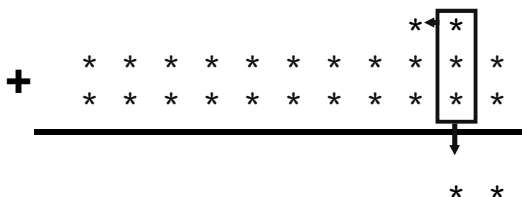
### How to add 2 n-bit numbers



### How to add 2 n-bit numbers



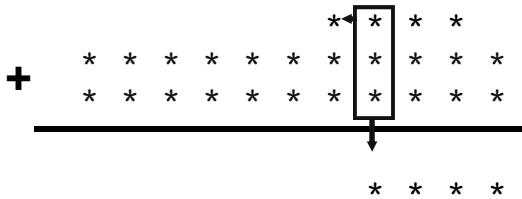
### How to add 2 n-bit numbers



### How to add 2 n-bit numbers



## How to add 2 n-bit numbers

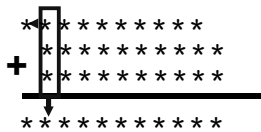


## How to add 2 n-bit numbers



“Grade school addition”

## Time complexity of grade school addition



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

What do we mean by “time”?

## Our Goal


We want to define “time” 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

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.

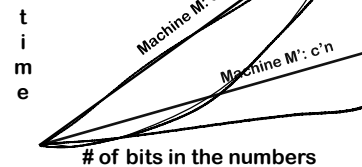
Total time to add two n-bit numbers using grade school addition:

$cn$  [i.e.,  $c$  time for each of  $n$  columns]

On another computer  $M'$ , the time to perform  $\square$  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 the time taken grows linearly as input size increases.

Thus we arrive at an implementation-independent insight:

Grade School Addition is a linear time algorithm

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

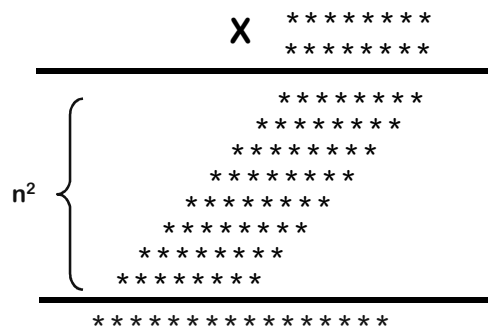
$TIME_n$  = the worst-case amount of time used by the algorithm on inputs of size  $n$

*# of bits*

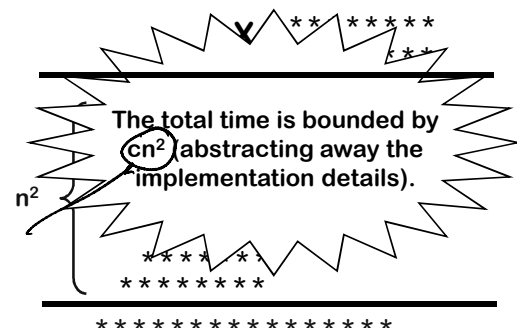
We often ask:

What is the growth rate of  $Time_n$ ?

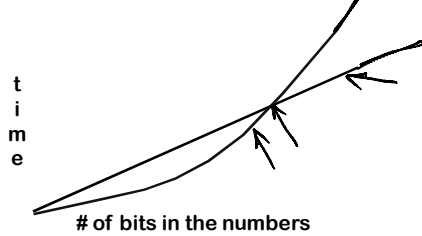
## How to multiply 2 $n$ -bit numbers.



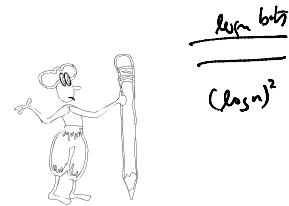
## How to multiply 2 $n$ -bit numbers.



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



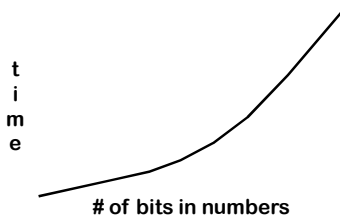
No matter how dramatic the difference in the constants, the quadratic curve will eventually dominate the linear curve



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

*log n bits to represent*

Grade School Multiplication:  
Quadratic time



$c(\log n)^2$  time to square the number  $n$   
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 increment (by 1)  $b$  times

$T(n) = ?$

If  $b = 000\dots0000$ , then NSA takes almost no time

If  $b = 1111\dots11111$ , then NSA takes  $cn2^n$  time

*n because incrementing might take cn time.*

*$b = 2^n - 1$*

## Worst Case Time

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

$$T(n) = \max_{\text{all permissible inputs } X \text{ of size } n} \text{Runtime}(A, X)$$

$\text{Runtime}(A, X) =$   
Running time of algorithm A on input X.

## 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,  $T(n) = cn2^n$

*one of b.  
adding 2 what it's*

Thus, Nursery School addition and Kindergarten 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.

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

*no*

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

Trial division up to  $\sqrt{N}$

for  $k = 2$  to  $\sqrt{N}$  do

if  $k \mid N$  then

return “ $N$  has a non-trivial factor  $k$ ”

return “ $N$  is prime”

*runtime  $\sqrt{N}$   
=  $\sqrt{2^{\log N}}$   
=  $2^{\frac{\log N}{2}}$*

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

Is this efficient?

No! The input length  $n = \log N$ .

Hence we’re using  $c 2^{n/2} n^2$  time.

Can we do better?

We know of methods for FACTORING that are sub-exponential (about  $2^{n^{1/3}}$  time) but nothing efficient.

*for dense inputs*

## Notation to Discuss Growth Rates

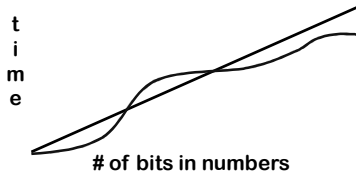
For any monotonic function  $f$  from the positive integers to the positive integers, we say

“ $f = O(n)$ ” 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: $\Omega$

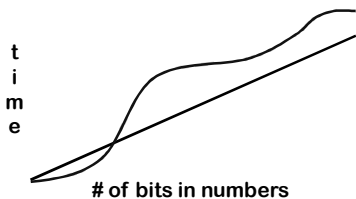
For any monotonic function  $f$  from the positive integers to the positive integers, we say

“ $f = \Omega(n)$ ” 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 below  $f$  from some point on



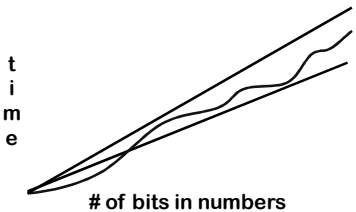
### Yet More Useful Notation: $\Theta$

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 from some point on.



### Notation to Discuss Growth Rates

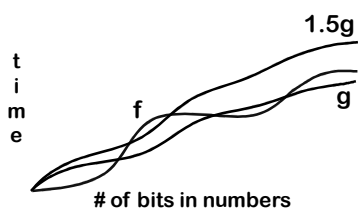
For any two monotonic functions  $f$  and  $g$  from the positive integers to the positive integers, we say

“ $f = O(g)$ ” 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.



## Other Useful Notation: $\Omega$

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: $\Theta$

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

•  $n = O(n^2)$  ? Yes!

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

$100n = O(n^2)$   
sufficiently  
 $\downarrow$   
 $n \geq 100$   
 $100n \leq 1 \cdot n^2$

•  $n = O(n^2)$  ? Yes!

•  $n = O(\sqrt{n})$  ? No

Suppose it were true that  $n \leq c \sqrt{n}$  for some constant  $c$  and large enough  $n$ .  
Cancelling, we would get  $\sqrt{n} \leq c$ .  
Which is false for  $n > c^2$

•  $n = O(n^2)$  ? Yes!

•  $n = O(\sqrt{n})$  ? No

•  $3n^2 + 4n + 4 = O(n^2)$  ? Yes!  $3n^2 + 4n + 4 \leq 4n^2$  for  $n \geq 5$

•  $3n^2 + 4n + 4 = \Omega(n^2)$  ? Yes!  $3n^2 + 4n + 4 \geq 3n^2$  for  $n \geq 0$

•  $n^2/10 = \Omega(n \log n)$  ? Yes!  $n^2/10 \geq (n \log n)/10$  for  $n \geq 1$

•  $n^2 \log n = \Theta(n^2)$  ? No

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

- $n^2 \log n = \Theta(n^2)$  ? No

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

But,  $n^2 \log n \neq O(n^2)$

If it were, then  $n^2 \log n \leq c n^2$   
for some  $c$  and large enough  $n$

But this is false for  $n > 2^c$

- $f = O(g)$  and  $g = O(h)$

then  $f = O(h)$  ? Yes!

$f(n) \leq c g(n)$  for all  $n \geq n_0$ .  
and  $g(n) \leq c' h(n)$  for all  $n \geq n_0'$ .

So  $f(n) \leq (cc') h(n)$  for all  $n \geq \max(n_0, n_0')$

- $f = O(g)$

then  $g = \Omega(f)$  Yes!

## Names For Some Growth Rates

Linear Time:  $T(n) = O(n)$  ←

Quadratic Time:  $T(n) = O(n^2)$  ←

Cubic Time:  $T(n) = O(n^3)$  ←

Quartic Time:  $T(n) = O(n^4)$

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

$$\begin{aligned} n2^n &= O(2^n) ?? \text{ No.} \\ &= 2^{n \uparrow} \\ \exists c=2 \\ n2^n &\leq 2^{2n} = (2^2)^n \\ &= 4^n \end{aligned}$$

## 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.

A very common example of logarithmic time is looking up a word in a sorted dictionary (binary search)

## Some Big Ones

Doubly Exponential Time means that for some constant  $k$

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

Triply Exponential

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



## Faster and Faster: 2STACK

$$2STACK(0) = 1$$

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

$$2STACK(1) = 2$$

$$2STACK(2) = 4$$

$$2STACK(3) = 16$$

$$2STACK(4) = 65536$$

$$2STACK(5) \geq 10^{80}$$

$$= \text{atoms in universe}$$

$$2^{2^{2^{2^{\dots^2}}}}$$

"tower of n 2's"

## And the inverse of 2STACK: $\log^*$

$$2STACK(0) = 1$$

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

$$2STACK(1) = 2 \quad \log^*(2) = 1$$

$$2STACK(2) = 4 \quad \log^*(4) = 2$$

$$2STACK(3) = 16 \quad \log^*(16) = 3$$

$$2STACK(4) = 65536 \quad \log^*(65536) = 4$$

$$2STACK(5) \geq 10^{80} \quad \log^*(\text{atoms}) = 5$$

$$= \text{atoms in universe}$$

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

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

However, for every constant  $c$ ,  $n \log^* n > cn$  when  $n$  gets large enough

## An Ackermann Function

$$A(1, n) = 2n \quad \text{for } n \geq 1$$

$$A(m, 1) = 2 \quad \text{for } m \geq 1$$

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

	n=1	2	3	4	5	6	
m=1	2	4	6	8	10	12	2n
2	2	$A(1,2)=4$	$A(1,3)=6$	$A(1,4)=8$	$A(1,5)=10$	$A(1,6)=12$	$2^2$
3	2	$A(2,2)=4$	$A(2,3)=6$	$A(2,4)=8$	$A(2,5)=10$	$A(2,6)=12$	$2^{2^2}$
4	2	$A(3,2)=4$	$A(3,3)=6$	$A(3,4)=8$	$A(3,5)=10$	$A(3,6)=12$	$2^{2^{2^2}}$
5	2	$A(4,2)=4$	$A(4,3)=6$	$A(4,4)=8$	$A(4,5)=10$	$A(4,6)=12$	$2^{2^{2^{2^2}}}$
6	2	$A(5,2)=4$	$A(5,3)=6$	$A(5,4)=8$	$A(5,5)=10$	$A(5,6)=12$	$2^{2^{2^{2^{2^2}}}}$

$\wedge$

## An Ackermann Function

$$A(1, n) = 2n \quad \text{for } n \geq 1$$

$$A(m, 1) = 2 \quad \text{for } m \geq 1$$

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

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

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

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

## An Ackermann Function

$$A(1, n) = 2n \quad \text{for } n \geq 1$$

$$A(m, 1) = 2 \quad \text{for } m \geq 1$$

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

$$A(5,3) > \# \text{ of particles in universe}$$

$A(6,3)$  can't be written out as decimal in this universe

## An Ackermann Function

$$A(1, n) = 2n \quad \text{for } n \geq 1$$

$$A(m, 1) = 2 \quad \text{for } m \geq 1$$

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

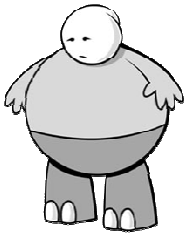
Define:  $A'(k) = A(k, k)$

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

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

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.



Here's What  
You Need to  
Know...

- How is “time” measured
- Definitions of:
  - $O, \Omega, \Theta$
  - linear, quadratic time, etc
  - $\log^*(n)$
  - Ackerman Function