15-251

Great Theoretical Ideas in Computer Science

This is The Big Oh!

Lecture 21 (March 31, 2009)

How to add 2 n-bit numbers

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

How to add 2 n-bit numbers

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

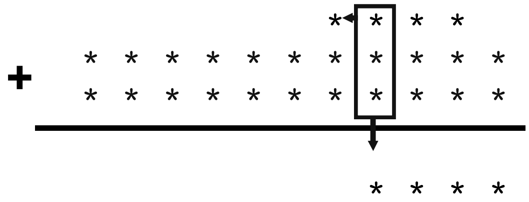
How to add 2 n-bit numbers

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

How to add 2 n-bit numbers

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

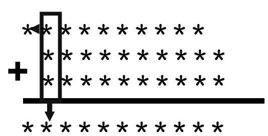
How to add 2 n-bit numbers



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_M to be the time it takes to perform on that computer.

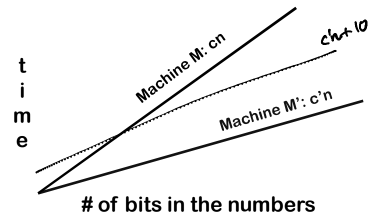
Total time to add two n-bit numbers using grade school addition:

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

On another computer M’, the time to perform [icon] 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

We often ask: What is the growth rate of $Time_n$?

How to multiply 2 n-bit numbers.

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

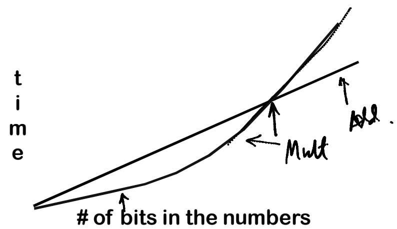
n^2

How to multiply 2 n-bit numbers.

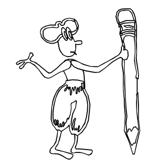
The total time is bounded by cn^2 (abstracting away the implementation details).

n^2

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

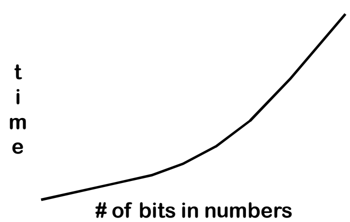


No matter how dramatic the difference in the constants, the quadratic curve will eventually dominate the linear curve



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
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…0000, then NSA takes almost no time

If b = 1111…11111, then NSA takes $cn2^n$ time

Worst Case Time

Worst Case Time T(n) for algorithm A:

$T(n) = \mathrm{Max}_{[\text{all permissible inputs X of size n}]}\ \mathrm{Runtime}(A,X)$

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, $T(n) = cn2^n$

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.

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 √N do
    if k | 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

Is this efficient? $\sqrt{2^{\text{length of } N}}$

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

Can we do better?

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

$2^{\sqrt[3]{n}}$

Notation to Discuss Growth Rates

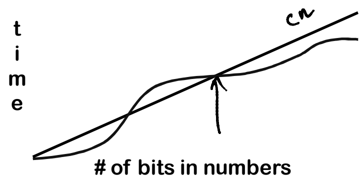
For any monotonic function f from the positive integers to the positive integers, we say

"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: $f(n) \le cn$]

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

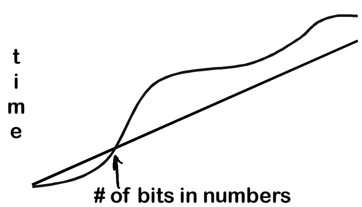
For any monotonic function f from the positive integers to the positive integers, we say

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



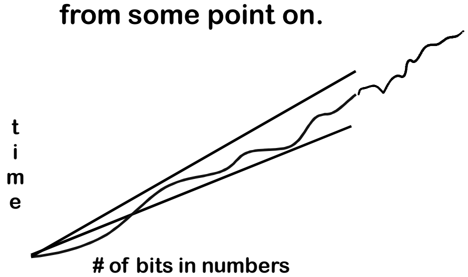
Yet More Useful Notation: Θ

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

“f = Θ(n)” or “f is Θ(n)”

if: f = O(n)   and   f = Ω(n)

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



Notation to Discuss Growth Rates

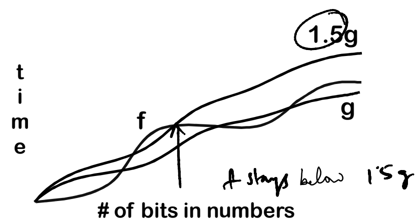
For any two monotonic functions f and g from the positive integers to the positive integers, we say

“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: Ω

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

“f = Ω(g)” or “f is Ω(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(n)

“f = Θ(g)” or “f is Θ(g)”

If: f = O(g) and f = Ω(g)

$f(n) = n$ $g(n) = n^2$

- $n = O(n^2)$? Yes!

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

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

$3n^2 + 4n + 4 \leq 100n^2 \ \forall n \geq 1$

- $3n^2 + 4n + 4 = O(n^2)$? 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

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

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

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

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

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

  (Assuming that $\log n = \log_2 n$)

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

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

Q: is $n \cdot 2^n = O(2^n)$? No
$= O(2^{1 \cdot n})$? Yes
is $n \cdot 2^n = 2^{O(n)}$? Yes

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

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

Triply Exponential

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

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) $\geq 10^{80}$
= atoms in universe

2STACK(n) = $\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{\text{“tower of n 2’s”}}$

And the inverse of 2STACK: log*

2STACK(0) = 1
2STACK(n) = $2^{2STACK(n-1)}$

2STACK(1) = 2	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*(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 first row
A(m, 0) = A(m - 1, 1) for m $\geq$ 1 first col
A(m, n) = A(m - 1, A(m, n - 1)) for m, n $\geq$ 1

	n=0	1	2	3	4	5	
m=0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	5	7	9	11	13	15
3	5	13	29				
4							
5							

2n+3

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

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

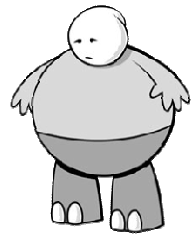
Define: A’(k) = A(k,k)

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

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

The inverse Ackermann function – in fact, Θ(n α(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, Ω, Θ
  - linear, quadratic time, etc
  - log*(n)
  - Ackerman Function