CS/COE 1501

www.cs.pitt.edu/~nlf4/cs1501/

Integer Multiplication

Integer multiplication

- Say we have 5 baskets with 8 apples in each
 - Our How do we determine how many apples we have?
 - Count them all?
 - That would take awhile...
 - Since we know we have 8 in each basket, and 5 baskets, lets simply add 8 + 8 + 8 + 8 + 8
 - \bullet = 40!
 - This is essentially multiplication!
 - \bullet 8 * 5 = 8 + 8 + 8 + 8 + 8

What about bigger numbers?

- Like 1284 * 1583, I mean!
 - That would take way longer than counting the 40 apples!
- Let's think of it like this:
 - 0 1284 * 1583 = 1284*3 + 1284*80 + 1284*500 + 1284*1000

	1284
X	1583
	3852
+	102720
+	642000
+	1284000
=	2032572

OK, I'm guessing we all knew that...

- ... and learned it quite some time ago ...
- So why bring it up now? What is there to cover about multiplication
- What is the runtime of this multiplication algorithm?
 - For 2 n-digit numbers:
 - \blacksquare n^2

Yeah, but the processor has a MUL instruction

- Assuming x86
- Given two 32bit integers, MUL will produce a 64 bit integer in a few cycles
- What about when we need to multiply large ints?
 - VERY large ints?
 - RSA keys should be 2048 bits
 - Back to grade school…

Gradeschool algorithm on binary numbers

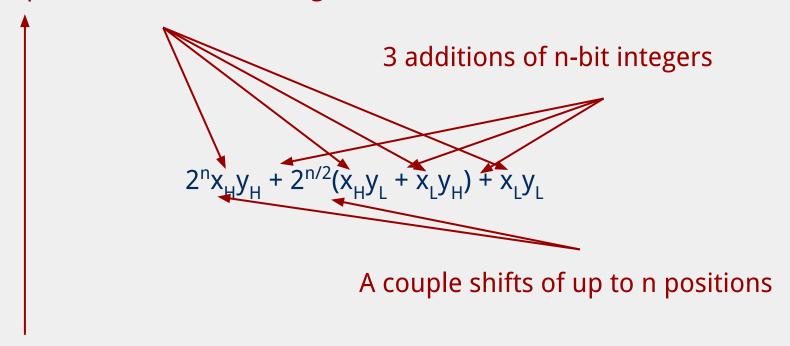
x 11000101111

How can we improve our runtime?

- Let's try to divide and conquer:
 - Break our n-bit integers in half:
 - **x** = 1001011011001000, n = 16
 - Let the high-order bits be $x_H = 10010110$
 - Let the low-order bits be $x_L = 11001000$
 - $x = 2^{n/2}x_H + x_L$
 - Do the same for y
 - $\mathbf{x} * \mathbf{y} = (2^{n/2}\mathbf{x}_{H} + \mathbf{x}_{L}) * (2^{n/2}\mathbf{y}_{H} + \mathbf{y}_{L})$
 - $\mathbf{x} * y = 2^n x_H y_H + 2^{n/2} (x_H y_L + x_L y_H) + x_L y_L$

So what does this mean?

4 multiplications of n/2 bit integers



Actually 16 multiplications of n/4 bit integers (plus additions/shifts)

Actually 64 multiplications of n/8 bit integers (plus additions/shifts)

• • •

So what's the runtime???

- Recursion really complicates our analysis...
- We'll use a recurrence relation to analyze the recursive runtime
 - Goal is to determine:
 - How much work is done in the current recursive call?
 - How much work is passed on to future recursive calls?
 - All in terms of input size

Recurrence relation for divide and conquer multiplication

- Assuming we cut integers exactly in half at each call
 - I.e., input bit lengths are a power of 2
- Work in the current call:
 - \circ Shifts and additions are $\Theta(n)$
- Work left to future calls:
 - 4 more multiplications on half of the input size

• $T(n) = 4T(n/2) + \Theta(n)$

Soooo... what's the runtime?

- Need to solve the recurrence relation
 - Remove the recursive component and express it purely in terms of n
 - A "cookbook" approach to solving recurrence relations:
 - The master theorem

The master theorem

Usable on recurrence relations of the following form:

$$T(n) = aT(n/b) + f(n)$$

- Where:
 - o a is a constant >= 1
 - b is a constant > 1
 - o and f(n) is an asymptotically positive function

Applying the master theorem

$$T(n) = aT(n/b) + f(n)$$

- If f(n) is $O(n^{\log_{-b(a)} \epsilon})$:
 - T(n) is $\Theta(n^{\log_{-b(a)}})$
- If f(n) is $\Theta(n^{\log_{-}b(a)})$
 - \circ T(n) is Θ(n^{log_b(a)} lg n)
- If f(n) is $\Omega(n^{\log_{-b(a)} + \epsilon})$ and (a * f(n/b) <= c * f(n)) for some c < 1:
 - \circ T(n) is $\Theta(f(n))$

Mergesort master theorem analysis

Recurrence relation for mergesort? $T(n) = 2T(n/2) + \Theta(n)$

$$T(n) = \frac{2}{2}T(n/2) + \Theta(n)$$

- a = 2 If f(n) is $O(n^{\log_{-b(a)} - \epsilon})$: • b = 2 \circ T(n) is $\Theta(n^{\log_{b(a)}})$ If f(n) is $\Theta(n^{\log_{b(a)}})$ f(n) is $\Theta(n)$ • T(n) is $\Theta(n^{\log_{b(a)}} \lg n)$ If f(n) is $\Omega(n^{\log_{-b(a)} + \varepsilon})$ • So... and $(a * f(n/b) \le c * f(n))$ for some c < 1: $\circ \quad n^{\log_{b(a)}} = \dots$ \circ T(n) is $\Theta(f(n))$
 - Being $\Theta(n)$ means f(n) is $\Theta(n^{\log_{-}b(a)})$
 - $T(n) = \Theta(n^{\log_{-b(a)}} \lg n) = \Theta(n^{\lg 2} \lg n) = \Theta(n \lg n)$

For our divide and conquer multiplication approach

$$T(n) = \frac{4}{4}T(n/2) + \Theta(n)$$

- a = 4
- b = 2
- f(n) is Θ(n)
- So...
 - $\circ \quad n^{\log_{b(a)}} = \dots$

- If f(n) is $O(n^{\log_{-}b(a)-\epsilon})$:

 T(n) is $\Theta(n^{\log_{-}b(a)})$ If f(n) is $\Theta(n^{\log_{-}b(a)})$ T(n) is $\Theta(n^{\log_{-}b(a)} | g | n)$ If f(n) is $\Omega(n^{\log_{-}b(a)+\epsilon})$ and (a * f(n/b) <= c * f(n)) for some c < 1:

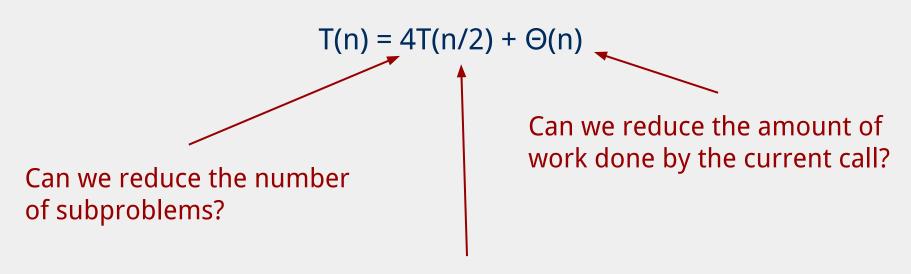
 T(n) is $\Theta(f(n))$
- \circ Being $\Theta(n)$ means f(n) is polynomially smaller than n^2
- $OT(n) = \Theta(n^{\log_{-b}(a)}) = \Theta($

@#\$%^&*

- Leaves us back where we started with the grade school algorithm...
 - Actually, the overhead of doing all of the dividing and conquering will make it slower than grade school

SO WHY EVEN BOTHER?

- Let's look for a smarter way to divide and conquer
- Look at the recurrence relation again to see where we can improve our runtime:



Can we reduce the subproblem size?

Karatsuba's algorithm

 By reducing the number of recursive calls (subproblems), we can improve the runtime

•
$$x * y = 2^{n}x_{H}y_{H} + 2^{n/2}(x_{H}y_{L} + x_{L}y_{H}) + x_{L}y_{L}$$

M1 M2 M3 M4

- We don't actually need to do both M2 and M3
 - We just need the sum of M2 and M3
 - If we can find this sum using only 1 multiplication, we decrease the number of recursive calls and hence improve our runtime

Karatsuba craziness

- M1 = $x_h y_h$; M2 = $x_h y_l$; M3 = $x_l y_h$; M4 = $x_l y_l$;
- The sum of all of them can be expressed as a single mult:
 - \circ M1 + M2 + M3 + M4
 - $\circ = X_h Y_h + X_h Y_l + X_l Y_h + X_l Y_l$
 - $\circ = (x_h + x_l) * (y_h + y_l)$
- Lets call this single multiplication M5:
 - \circ M5 = $(x_h + x_l) * (y_h + y_l) = M1 + M2 + M3 + M4$
- Hence, M5 M1 M4 = M2 + M3
- So: $x * y = 2^nM1 + 2^{n/2}(M5 M1 M4) + M4$
 - Only 3 multiplications required!
 - At the cost of 2 more additions, and 2 subtractions

Karatsuba runtime

- To get M5, we have to multiply (at most) n/2 + 1 bit ints
 - Asymptotically the same as our other recursive calls
- Requires extra additions and subtractions...
 - But these are all Θ(n)
- So, the recurrence relation for Karatsuba's algorithm is:
 - \circ T(n) = 3T(n/2) + Θ (n)
 - Which solves to be $\Theta(n^{\lg 3})$
 - Asymptotic improvement over grade school algorithm!
 - For large n, this will translate into practical improvement

Large integer multiplication in practice

- Can use a hybrid algorithm of grade school for large operands, Karatsuba's algorithm for VERY large operands
 - Why are we still bothering with grade school at all?

Is this the best we can do?

- The Schönhage–Strassen algorithm
 - Uses Fast Fourier transforms to achieve better asymptotic runtime
 - O(n log n log log n)
 - Fastest asymptotic runtime known from 1971-2007
 - Required n to be astronomical to achieve practical improvements to runtime
 - \circ Numbers beyond $2^{2^{15}}$ to $2^{2^{17}}$
- Fürer was able to achieve even better asymptotic runtime in 2007
 - o $n \log n 2^{O(\log^{n} n)}$
 - No practical difference for realistic values of n