CS/COE 1501

cs.pitt.edu/~bill/1501/

Integer Multiplication
Say we have 5 baskets with 8 apples in each

- 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, let's simply add $8 + 8 + 8 + 8 + 8$
    - $= 40$!
- This is essentially multiplication!
  - $8 \times 5 = 8 + 8 + 8 + 8 + 8$
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:
  ○ $1284 \times 1583 = 1284 \times 3 + 1284 \times 80 + 1284 \times 500 + 1284 \times 1000$

\[
\begin{array}{c}
  1284 \\
  \times 1583 \\
  \hline
  3852 \\
  + 102720 \\
  + 642000 \\
  + 1284000 \\
  \hline
  = 2032572
\end{array}
\]
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:
    - \( n^2 \)
- Space?
Yeah, but the processor has a MUL instruction

- Assuming x86
- Given two 32-bit 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 at least 2048 bits
  - Back to grade school…
Gradeschool algorithm on binary numbers

\[
\begin{array}{c}
10100000100 \\
x 11000101111 \\
\hline
10100000100 \\
10100001000 \\
101000010000 \\
1010000100000 \\
0000000000000000 \\
10100001000000 \\
0000000000000000 \\
1010000100000000 \\
0000000000000000 \\
1010000010000000 \\
0000000000000000 \\
10100000100000000 \\
0000000000000000 \\
\hline
111110000001110111100
\end{array}
\]
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 \(y\)
    - \(x \times y = (2^{n/2}x_H + x_L) \times (2^{n/2}y_H + y_L)\)
    - \(x \times y = 2^n x_H y_H + 2^{n/2}(x_H y_L + x_L y_H) + x_L y_L\)
So what does this mean?

4 multiplications of $n/2$ bit integers

$2^n x_H y_H + 2^{n/2} (x_H y_L + x_L y_H) + x_L y_L$

3 additions of $n$-bit integers

A couple shifts of up to $n$ positions

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

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

...
Recursion really complicates our analysis…

We’ll use a \textit{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:
  - 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:
  - \( a \) is a constant \( \geq 1 \)
  - \( b \) is a constant \( > 1 \)
  - and \( f(n) \) is an asymptotically positive function
Applying the master theorem

\[ T(n) = aT\left(\frac{n}{b}\right) + f(n) \]

- If \( f(n) \) is \( O(n^{\log_b a} - \varepsilon) \):
  - \( T(n) \) is \( \Theta(n^{\log_b a}) \)

- If \( f(n) \) is \( \Theta(n^{\log_b a}) \)
  - \( T(n) \) is \( \Theta(n^{\log_b a} \lg n) \)

- If \( f(n) \) is \( \Omega(n^{\log_b a} + \varepsilon) \) and \( a \cdot f(n/b) \leq c \cdot f(n) \) for some \( c < 1 \):
  - \( T(n) \) is \( \Theta(f(n)) \)
Mergesort master theorem analysis

Recurrence relation for mergesort?

- \( a = 2 \)
- \( b = 2 \)
- \( f(n) \) is \( \Theta(n) \)
- So...
  - \( n^{\log_b(a)} = \ldots \)
    - \( n \log 2 = n \)
  - Being \( \Theta(n) \) means \( f(n) \) is \( \Theta(n^{\log_b(a)}) \)
  - \( T(n) = \Theta(n^{\log_b(a)} \lg n) = \Theta(n \log 2 \lg n) = \Theta(n \lg n) \)

\[ T(n) = 2T(n/2) + \Theta(n) \]
For our divide and conquer multiplication approach

\[ T(n) = 4T(n/2) + \Theta(n) \]

- \( a = 4 \)
- \( b = 2 \)
- \( f(n) \) is \( \Theta(n) \)
- So...
  - \( n^{\log_b a} = \ldots \)
  - \( n^{\lg 4} = n^2 \)
  - Being \( \Theta(n) \) means \( f(n) \) is polynomially smaller than \( n^2 \)
  - \( T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\lg 4}) = \Theta(n^2) \)

- If \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \):
  - \( T(n) \) is \( \Theta(n^{\log_b a}) \)
- If \( f(n) \) is \( \Theta(n^{\log_b a}) \):
  - \( T(n) \) is \( \Theta(n^{\log_b a} \lg n) \)
- If \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \)
  and \( a \ast f(n/b) \leq c \ast f(n) \) for some \( c < 1 \):
  - \( T(n) \) is \( \Theta(f(n)) \)
• 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:

\[ T(n) = 4T(n/2) + \Theta(n) \]

Can we reduce the amount of work done by the current call?
Can we reduce the subproblem size?
Can we reduce the number of subproblems?
Karatsuba’s algorithm

- By reducing the number of recursive calls (subproblems), we can improve the runtime.
- \[ x \times y = 2^n x_H y_H + 2^{n/2} (x_H y_L + x_L y_H) + x_L y_L \]

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 = xhyh; M2 = xhyi; M3 = xiyh; M4 = xiyi;
- The sum of all of them can be expressed as a single mult:
  - M1 + M2 + M3 + M4
  - = xhyh + xhyi + xiyh + xiyi
  - = (xh + xi) * (yh + yi)
- Let's call this single multiplication M5:
  - M5 = (xh + xi) * (yh + yi) = 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 $\Theta(n)$
- So, the recurrence relation for Karatsuba’s algorithm is:
  - $T(n) = 3T(n/2) + \Theta(n)$
    - Which solves to be $\Theta(n^{\log_2 3})$
      - 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?
The Schönhage–Strassen algorithm

- Uses Fast Fourier transforms to achieve better asymptotic runtime
  - $O(n \cdot \log(n) \cdot \log(\log(n)))$
  - Fastest asymptotic runtime known from 1971-2007
    - Required $n$ to be astronomical to achieve practical improvements to runtime
      - Numbers beyond $2^{2^{15}}$ to $2^{2^{17}}$

Fürer was able to achieve even better asymptotic runtime in 2007

- $n \cdot \log(n) \cdot 2^{O(\log^* n)}$
- No practical difference for realistic values of $n$