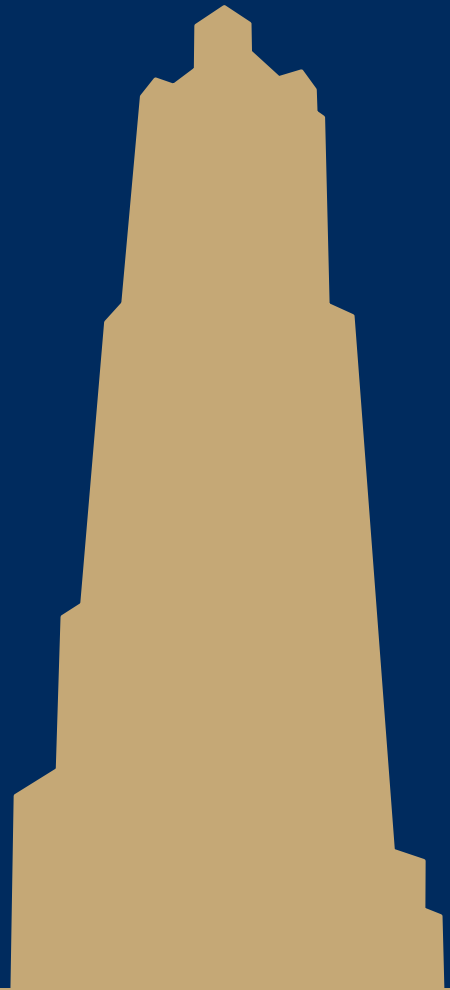


CS/COE 1501

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

Integer Multiplication



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!
 - $8 * 5 = 8 + 8 + 8 + 8 + 8$

What about bigger numbers?

- e.g. $1284 * 356$
 - That would take much longer than counting the 40 apples!
- Let's think of it like this:
 - $1284 * 356 = 1284 * 6 + 1284 * 50 + 1284 * 300$

$$\begin{array}{r} 1284 \\ \times 356 \\ \hline 7704 \\ + 64200 \\ + 385200 \\ \hline = 457104 \end{array}$$

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?
- What about space complexity?

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 2048 bits
 - Back to grade school...

Gradeschool algorithm on binary numbers

$$\begin{array}{r} 10100000100 \\ \times \quad 101100100 \\ \hline 00000000000 \\ 00000000000 \\ 1010000010000 \\ 00000000000000 \\ 000000000000000 \\ 1010000010000000 \\ 10100000100000000 \\ 000000000000000000 \\ 1010000010000000000 \\ \hline 1101111100110010000 \end{array}$$

Let's improve the space requirements

- Assume x and y are n digits long
- Want to compute z ($2n-1$ digits long)

GradeschoolImproved (x, y):

```
s = 0
for j = 0 to (2n-1) do           // go through digits of result
  for i = 0 to (n-1) do         // digits of x
    if (0 <= (j-i) <= (n-1)) then
      s = s + x[i] * y[j-i]
    z[j] = s mod 10             // remainder to get cur digit
    s = s / 10                  // integer division for carry
return z
```

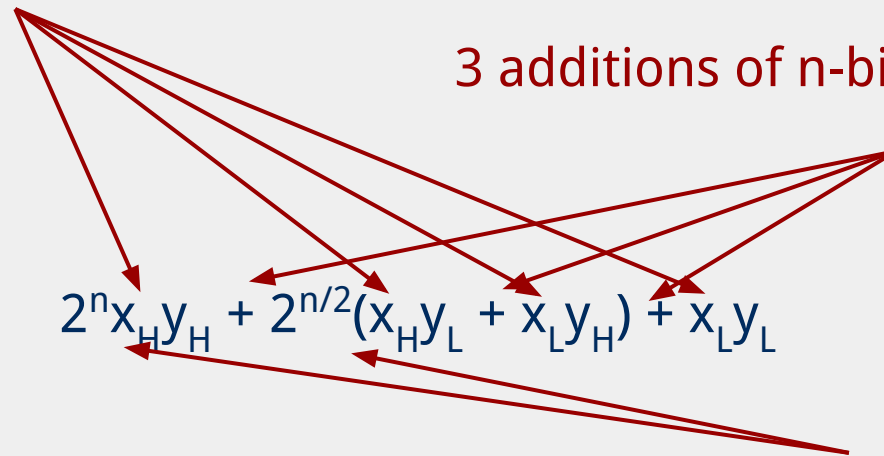
- By calculating the result by column, we can achieve huge memory savings

How can we improve on time?

- 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
 - $x * y = (2^{n/2}x_H + x_L) * (2^{n/2}y_H + y_L)$
 - $x * y = 2^n x_H y_H + 2^{n/2}(x_H y_L + x_L y_H) + x_L y_L$

So what does this mean?

4 multiplications of $n/2$ bit integers



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)

...

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

Introduction to Recurrence Relations

- Before tackling the divide & conquer multiplication algorithm, let's start with a more familiar algorithm:
 - Merge sort

```
void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo) return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid + 1, hi);  
    merge(a, aux, lo, mid, hi);  
}
```

- What's the runtime for Merge sort?

Recurrence Relation for Merge Sort

```
void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {  
    if (hi <= lo) return;  
    int mid = lo + (hi - lo) / 2;  
    sort(a, aux, lo, mid);  
    sort(a, aux, mid + 1, hi);  
    merge(a, aux, lo, mid, hi);  
}
```

- Let's determine:
 - How much work is done in the current recursive call?
 - How much work is passed on to future recursive calls?
 - How many recursive calls do we make in the current call?
 - What assumptions did we make?

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 ≥ 1
 - b is a constant > 1
 - 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 $\Theta(n^{\log_b(a)})$
 - $T(n)$ is $\Theta(n^{\log_b(a)} \lg n)$
- If $f(n)$ is $\Omega(n^{\log_b(a) + \epsilon})$ and $(a * f(n/b) \leq c * f(n))$ for some $c < 1$:
 - $T(n)$ is $\Theta(f(n))$

For our divide and conquer approach

$$T(n) = 2T(n/2) + 2n$$

- $a = 2$
- $b = 2$
- $f(n)$ is $2n$
- So...

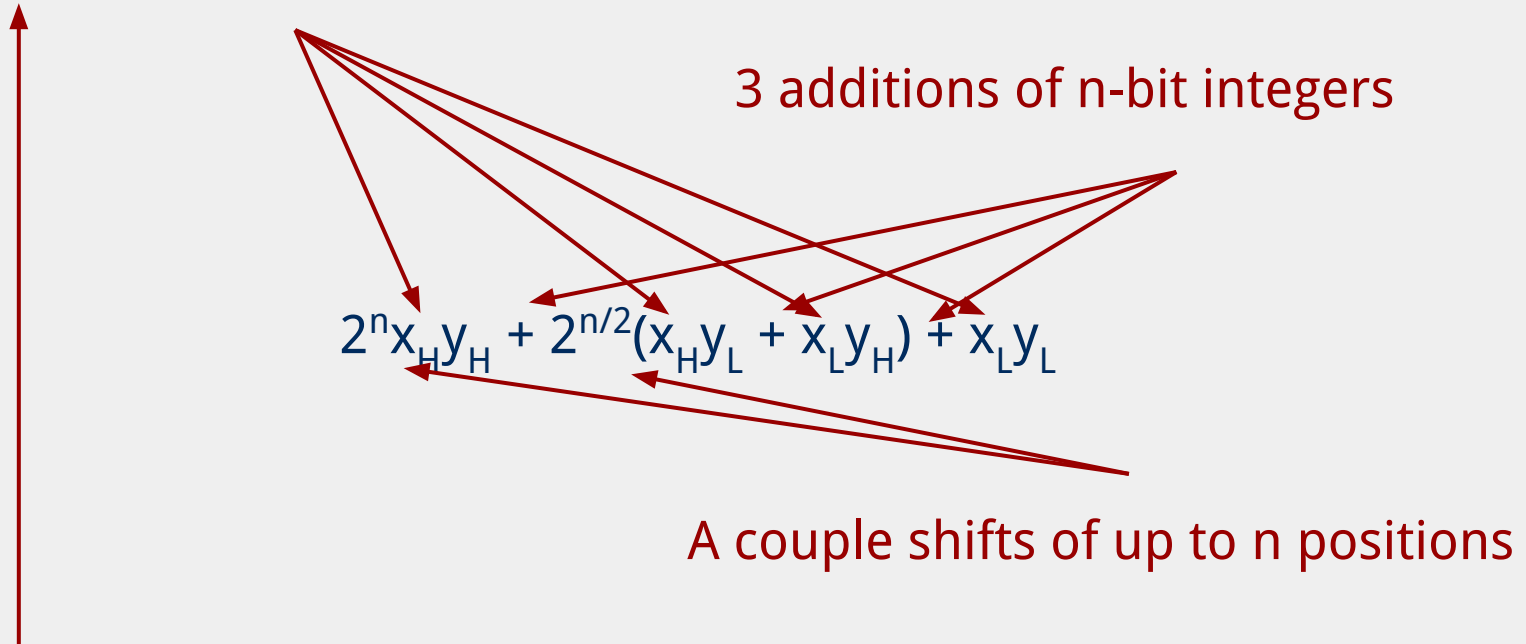
- $n^{\log_b(a)} = n^{\lg 2} = n^1$

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

- Being $\Theta(n)$ means $f(n)$ is equal to n^1
- $T(n) = \Theta(n^{\log_b(a)} \lg n) = \Theta(n^{\lg 2} \lg n) = \Theta(n \lg n)$

Returning to Divide & Conquer Multiplication

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)

...

Recurrence relation for divide and conquer multiplication

- Returning to divide & conquer multiplication
- Let's determine:
 - How much work is done in the current recursive call?
 - How much work is passed on to future recursive calls?
 - How many recursive calls do we make in the current call?
 - What assumptions did we make?

For our divide and conquer approach

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

- $a =$
- $b =$
- $f(n)$ is

- $T(n) = ?$

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

Conclusion

- 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 number of subproblems?

Can we reduce the amount of work done by the current call?

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:
 - $M1 + M2 + M3 + M4$
 - $= x_h y_h + x_h y_l + x_l y_h + x_l y_l$
 - $= (x_h + x_l) * (y_h + y_l)$
- Lets call this single multiplication M5:
 - $M5 = (x_h + x_l) * (y_h + y_l) = M1 + M2 + M3 + M4$
- Hence, $M5 - M1 - M4 = M2 + M3$
- So: $x * y = 2^n M1 + 2^{n/2} (M5 - M1 - M4) + M4$
 - Only 3 multiplications required!
 - At the cost of 2 more additions, and 2 subtractions

Karatsuba runtime

- To get M_5 , 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 $\Theta(n^{\lg 3})$
 - 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
 - Numbers beyond $2^{2^{15}}$ to $2^{2^{17}}$
- Fürer was able to achieve even better asymptotic runtime in 2007
 - $n \log n 2^{O(\log^* n)}$
 - No practical difference for realistic values of n