Course Notes for

CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
    - Algorithms in C++ by Robert Sedgewick
    - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
    - Introduction to Algorithms, by Cormen, Leiserson and Rivest
    - Various Java and C++ textbooks
    - Various online resources (see notes for specifics)

- Previously we discussed two algorithms for multiplying N-bit integers:
  - Gradeschool, requiring Theta(N^2) time
  - Simple divide and conquer, also Theta(N^2)
- Between the two, we would prefer gradeschool due to less overhead
  - However, maybe we can make the divide and conquer algorithm asymptotically better
    - Let's reconsider this algorithm and see how we can improve it
    - Let's start by considering the recurrence relation for it

Improving Divide and Conquer Multiplication

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

- We can try to reduce the amount of work in the current call
  - This could work, but will not in this case
    - This work sums to Theta(N^2) but so does the left part of the equation, so reducing it will keep the overall time at N^2
- We can try to make our subproblems smaller
  - Ex: N/3 or N/4 → but this would complicate our formula and likely require more subproblems
- We can try to reduce the number of subprobs
  - If possible, without changing the rest of the recur.

- Karatsuba's Algorithm
  - If we can **reduce** the number of **recursive calls** needed for the divide and conquer algorithm, perhaps we can improve the run-time
    - How can we do this?
    - Let's look at the equation again

$$XY = 2^N X_H Y_H + 2^{N/2}(X_H Y_L + X_L Y_H) + X_L Y_L$$
$$(M_1) \qquad (M_2) \qquad (M_3) \qquad (M_4)$$

    - Note that we don't really NEED M_2 and M_3 individually
      – All we need is the SUM OF THE TWO, $M_2 + M_3$
    - If we can somehow derive this sum using only one rather than two multiplications, we can improve our overall run-time

- Now consider the following product:

$(X_H + X_L) * (Y_H + Y_L) = X_H Y_H + X_H Y_L + X_L Y_H + X_L Y_L$

> Note: This is the same as the original product but without any shifting of the high bits

- Using our M values from the previous slide, this equals

$M_1 + M_2 + M_3 + M_4$

- The value we want is $M_2 + M_3$, so define M_{23}

$M_{23} = (X_H + X_L) * (Y_H + Y_L)$

- And our desired value is $\boxed{M_{23} - M_1 - M_4} = \boxed{M_2 + M_3}$
- Ok, all I see here is wackiness! How does this help?
  - Let's go back to the original equation, and plug back in

$$XY = 2^N X_H Y_H + 2^{N/2}(\boxed{X_H Y_L + X_L Y_H}) + X_L Y_L$$

$$= 2^N M_1 + 2^{N/2}(\boxed{M_{23} - M_1 - M_4}) + M_4$$

- Only 3 mults needed: M_1, M_4 and M_{23}

More Integer Multiplication

- But will this cause other parts of the recurrence to increase?
  - Looking back, we see that M_{23} involves multiplying at most (N/2)+1 bit integers, so asymptotically it is the same size as our other recursive multiplications
  - We have to do some extra additions and two subtractions, but these are all Theta(N) operations
- Thus, we now have the following recurrence:

  T(N) = 3T(N/2) + Theta(N)

- This solves to Theta($N^{\lg 3}$) ≈ **Theta($N^{1.58}$)**
  - Now we have an asymptotic improvement over the Gradeschool algorithm
    - > Still a lot of overhead, but for large enough N it will run faster than Gradeschool
- See
  - http://en.wikipedia.org/wiki/Karatsuba_algorithm
  - http://www.javamex.com/tutorials/math/BigDecimal_BigInteger_performance_multiply.shtml

- Practical Use?
  - Hybrid algorithm that uses Gradeschool until large enough N (ex: ~3000 decimal digits or ~3000 x $lg_2 10$ bits) and then switches to Karatsuba
- Can we do even better?
  - If we multiply the integers indirectly using the Fast Fourier Transform (FFT), we can achieve a run-time of Theta(N[lgN][lglgN])
  - This requires even larger numbers before it shows superiority (10s of thousands of decimal digits)
  - Don't worry about the details of this algorithm
    - But if you are interested look at

http://en.wikipedia.org/wiki/Sch%C3%B6nhage-Strassen_algorithm

- How about integer powers: X^Y
  - Natural approach: simple for loop

```
ZZ ans = 1;  // assume ZZ is very large int
for (ZZ ctr = 1; ctr <= Y; ctr++)
     ans = ans * X;
```

  - This seems ok – one for loop and a single multiplication inside – is it linear?
  - Let's look more closely
    - Total run-time is
      1) **Number of iterations of loop** *
      2) **Time per multiplication**

This assumes that the results as we multiply are still N-bit numbers – we will see how to ensure this soon.

Exponentiation

- We already know 2) since we just did it
  - Assuming GradeSchool, Theta(N^2) for N-bit ints
- How about 1)
  - It seems linear, since it is a simple loop
  - In fact, it is **LINEAR IN THE VALUE of Y**
  - However, our calculations are based on N, the **NUMBER OF BITS in Y**
  - What's the difference?
    > We know an N-bit integer can have a value of up to $\approx 2^N$
    > So **linear in the value of Y** is **exponential in the bits of Y**
  - Thus, the iterations of the for loop are actually Theta(2^N) and thus our total runtime is Theta($N^2 2^N$)
- This is RIDICULOUSLY BAD
    > Consider N = 512 – we get $(512)^2(2^{512})$
    > Just how big is this number?

- Let's calculate in base 10, since we have a better intuition about size
- Since every 10 powers of 2 is approximately 3 powers of ten, we can multiply the exponent by 3/10 to get the base 10 number
- So $(512)^2(2^{512}) = (2^9)^2(2^{512}) = 2^{530} \approx 10^{159}$
- Let's assume we have a 10 GHz machine (10^{10} cyc/sec)
- This would mean we need 10^{149} seconds
- $(10^{149}\text{sec})(1\text{hr}/3600\text{sec})(1\text{day}/24\text{hr})(1\text{yr}/365\text{days}) = (10^{149}/(31536000))$ years $\approx 10^{149}/10^8 \approx$ 10^{141} years
- This is ridiculous!!

- But we need exponentiation for RSA, so how can we do it more efficiently?

- How about a divide and conquer algorithm
  - Divide and conquer is usually worth a try
- Consider

  $X^Y = (X^{Y/2})^2$ when Y is even

  how about when Y is odd?

  $X^Y = X * (X^{Y/2})^2$ when Y is odd
- Naturally we need a base case

  $X^Y = 1$ when $Y = 0$
- We can easily code this into a recursive function
- What is the run-time?

- Let's see…our problem is to calculate the exponential X^Y for X and Y
  - So we have a recursive call with an argument of ½ the original size, plus a multiplication (again assume we will use GradeSchool)
    - **We'll put the multiplication time back in later**
    - For now let's determine the number of function calls
  - How many times can we divide Y by 2 until we get to a base case?
  - Since Y is a N-bit integer, it could be up to 2^N
  - Thus, we will start at 2^N

Exponentiation

Step 0: $2^N \quad = Y$

Step 1: $2^{N-1} \quad = Y/2^1$

Step 2: $2^{N-2} \quad = Y/2^2$

…

Step N: $2^{N-N} = Y/2^N = 1$

- How many total steps?
  - $N+1 = \lg_2(Y) + 1$
- Thus, the number of recursive calls is logarithmic in Y and **linear in N**
  - Compare this to the for loop version
- Since we have one or two mults per call, we end up with a total runtime of Theta(N^2*N) = **Theta(N^3)**

- This is an AMAZING improvement
  - Consider again N = 512
  - N^3 = 134217728 – less than a billion
  - On a 10GHz machine this would take less than a second
    - Remember that or naïve algorithm required 10^{141} years
  - Think about that difference!
- But is this result actually correct?
  - Let’s think about our result value

Power Mod

- Note that the power function can create enormous numbers
  - If X is N bits, X^2 is 2N bits, X^3 is 3N bits and so on
    - This increases the time required for the next multiplication and causes a lot of overhead for memory allocation
    - In the end our run-time will be dominated by multiplication itself, once the numbers get $\text{HUGE}^{\text{HUGE}}$
    - We also could not fit these numbers in memory
  - In practice (ex: for encryption) we perform the power operation modulo some other value
    - The final result must therefore be less than the modulo value, keeping the run-time for multiplication and the memory overhead in check
  - See Power.java

Exponentiation

- Can we improve even more?
  - Well removing the recursion can always help
  - If we start at X, then square repeatedly, we get the same effect as the recursive calls
  - Square at each step, and also multiply by X if there is a 1 in the binary representation of Y (from left to right)
  - Ex: $X^{45} = X^{101101} =$

    $1, X \quad X^2 \quad X^4, X^5 \quad X^{10}, X^{11} \quad X^{22} \quad X^{44}, X^{45}$

    1 0 1 1 0 1

  - Same idea as the recursive algorithm but building from the "bottom up"
  - See Power.java