

MA/CSSE 473

Day 03

A Closer Look at
Arithmetic



Prelude

- What we become depends on what we read after all of the professors have finished with us. The greatest university of all is a collection of books.
- Thomas Carlyle



MA/CSSE 473 Day 03

- Reminder: HW 1 is due tomorrow
- My ??????? ate your quizzes
- **Student Questions**
- Continue Algorithm Overview/Review
 - Fibonacci Recap
 - Fibonacci computation by Matrix Multiplication
 - A Closer Look at Addition and Multiplication
 - Four different multiplication algorithms
 - Standard
 - "European"
 - Divide and Conquer
 - Divide and Conquer plus Gauss
 - Impact on Fibonacci (probably tomorrow)



Recap: Fibonacci Numbers

- Straightforward recursive algorithm:

- Correctness is obvious

- $T(N) \geq F(N)$, which is exponential ($\approx 2^{0.69N}$)

- Algorithm with storage to avoid recomputing:

- Again, correctness is obvious

- And the run-time is linear

- until we dig deeper later

```
def fib1(n):  
    if n==0:  
        return 0  
    if n==1:  
        return 1  
    return fib1(n-1) + fib1(n-2)  
  
print fib1(6), fib1(7), fib1(8)
```

```
def fib2(n):  
    nums = [0]*(n+1)  
    nums[0] = 0  
    nums[1] = 1  
    for i in range(2, n+1):  
        nums[i] = nums[i-1] + nums[i-2]  
    return nums[n]
```



A Creative $O(\log N)$ Algorithm?

- Let X be the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$
- Then $\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = X \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$
- also $\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = X \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = X^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}, \text{ and } \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = X^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$
- How many additions and multiplications of numbers are necessary to multiply two 2×2 matrices?
- If $n = 2^k$, how many matrix multiplications does it take to compute X^n ?
 - What if n is not a power of 2?
 - Implement it with a partner (next slide)
 - Then we will analyze it
- But there is a catch!



```
identity_matrix = [[1,0],[0,1]]
x = [[0,1],[1,1]]
```

```
def matrix_multiply(a, b):
    return [[a[0][0]*b[0][0] + a[0][1]*b[1][0],
            a[0][0]*b[0][1] + a[0][1]*b[1][1]],
            [a[1][0]*b[0][0] + a[1][1]*b[1][0],
            a[1][0]*b[0][1] + a[1][1]*b[1][1]]]
```

```
def matrix_power(m, n):
    result = identity_matrix
    # Fill in the details
```

```
    return result
```

```
def fib (n) :
    return matrix_power(x, n)[0][1]
```

```
print [fib(i) for i in range(11)]
```



```
identity_matrix = [[1,0],[0,1]]
```

```
x = [[0,1],[1,1]]
```

```
def matrix_multiply(a, b):
```

```
    return [[a[0][0]*b[0][0] + a[0][1]*b[1][0],  
            a[0][0]*b[0][1] + a[0][1]*b[1][1]],  
           [a[1][0]*b[0][0] + a[1][1]*b[1][0],  
            a[1][0]*b[0][1] + a[1][1]*b[1][1]]]
```

```
def matrix_power(m, n):
```

```
    result = identity_matrix
```

```
    power = m
```

```
    while n > 0:
```

```
        if n % 2 == 1:
```

```
            result = matrix_multiply(result, power)
```

```
            power = matrix_multiply(power, power)
```

```
            n = n / 2
```

```
    return result
```

```
def fib (n) :
```

```
    return matrix_power(x, n)[0][1]
```

Back to the end of the 2nd previous slide!



Why so complicated?

- Why not just use the formula that you proved in CSSE 230 to calculate $F(N)$?

$$F(N) = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^N - \left(\frac{1 - \sqrt{5}}{2} \right)^N \right)$$



The catch!

- Are addition and multiplication constant-time operations?
- We take a closer look at the "basic operations"
- **Addition first:**
- At most, how many digits in the sum of three decimal one-digit numbers?
- Is the same result true in binary?
- Add two n-bit positive integers (53+35):

$$\begin{array}{r} \text{Carry:} \quad 1 \qquad \qquad \qquad 1 \quad 1 \quad 1 \\ \qquad \qquad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad (35) \\ \qquad \qquad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad (53) \\ \hline 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad (88) \end{array}$$

- So adding two n-bit integers is $O(n)$.



Multiplication

- Example: multiply 13 by 11

				1	1	0	1		
				x	1	0	1	1	
<hr/>									
					1	1	0	1	(1101 times 1)
					1	1	0	1	(1101 times 1, shifted once)
			0	0	0	0			(1101 times 1, shifted twice)
		1	1	0	1				(1101 times 1, shifted thrice)
<hr/>									
1	0	0	0	1	1	1	1		(binary 143)

- There are n rows of $2n$ bits to add, so we do an $O(n)$ operation n times, thus the whole multiplication is $O()$?
- Can we do better?



Multiplication by an Ancient Method

- This approach was known to Al Khwarizimi
- According to Dasgupta, *et al*, still used today in some European countries
- Repeat until 1st number is 1, keeping all results:
 - Divide 1st number by 2 (rounding down)
 - double 2nd number

- Example

11	13
5	26
2	52
1	104
	<hr/>
	143

Then strike out any rows whose first number is even, and add up the remaining numbers in the second column.

- Correct? Analysis



Recursive code for this algorithm

```
def multiply(m, n):  
    "multiply two integers m and n, where n >= 0"  
    if n == 0:  
        return 0  
    z = multiply (m, n / 2)  
    if n % 2 == 0:  
        return 2 * z  
    return m + 2 * z  
  
print multiply(11, 13)
```



Can we do better than $O(n^2)$?

- Is there an algorithm for multiplying two n -bit numbers in time that is less than $O(n^2)$?
- **Basis:** A discovery of Carl Gauss (1777-1855)
 - Multiplying complex numbers:
 - $(a + bi)(c + di) = ac - bd + (bc + ad)i$
 - Seems to need four real-number multiplications and two additions
 - But $bc + ad = (a+b)(c+d) - ac - bd$
 - And we have already computed ac and bd for the real part!
 - Thus we can do the original product with 3 multiplications and 5 additions
 - Additions are so much faster than multiplications that we can essentially ignore them.
 - A little savings, but not a big deal until applied recursively!



Review: The Master Theorem

- The Master Theorem for Divide and Conquer recurrence relations:
- Consider the recurrence $T(n) = aT(n/b) + f(n)$, $T(1)=c$, where $f(n) = \Theta(n^k)$ and $k \geq 0$,
- The solution is
 - $\Theta(n^k)$ if $a < b^k$
 - $\Theta(n^k \log n)$ if $a = b^k$
 - $\Theta(n^{\log_b a})$ if $a > b^k$



New Multiplication Approach

- **Divide and Conquer**

- To multiply two n -bit integers x and y :

- Split each into its left and right halves so that

$$x = 2^{n/2}x_L + x_R, \quad \text{and} \quad y = 2^{n/2}y_L + y_R$$

- The straightforward calculation of xy would be

$$(2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = \\ 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

- Code on next slide

- We can do the four multiplications of $n/2$ -bit integers using four recursive calls, and the rest of the work (a constant number of bit shifts and additions) in time $O(n)$

- Thus $T(n) =$. Solution?



Code for divide-and-conquer

```
def multiply(x, y, n):  
    """multiply two integers x and y, where n >= 0  
       is a power of 2, and as large as the maximum number of bits in x or y"""  
  
    if n == 1:  
        return x * y  
  
    n_over_two = n/2  
  
    two_to_the_n_over_two = 1 << n_over_two  
  
    xL, xR = x / two_to_the_n_over_two, x % two_to_the_n_over_two  
    yL, yR = y / two_to_the_n_over_two, y % two_to_the_n_over_two  
    # note that these two operations could be done by bit shifts and masking.  
  
    p1 = multiply (xL, yL, n_over_two)  
    p2 = multiply (xL, yR, n_over_two)  
    p3 = multiply (xR, yL, n_over_two)  
    p4 = multiply (xR, yR, n_over_two)  
  
    return (p1 << n) + ((p2 + p3) << n_over_two) + p4
```

