MA/CSSE 473 Day 02

Some Numeric Algorithms and their Analysis

Leftovers

- Algorithm definition:
  - Sequence of instructions
  - For solving a problem
  - Unambiguous (including order)
  - Can depend on input
  - Terminates in a finite amount of time
- Session # → day of week algorithm

Student questions on …

- Syllabus?
- Course procedures, policies, or resources?
- Course materials?
- Homework assignments?
- Anything else?

notation: **lg n** means **$\log_2 n$**

Also, **log n** without a specified base will usually mean **$\log_2 n$**

Levitin Algorithm picture

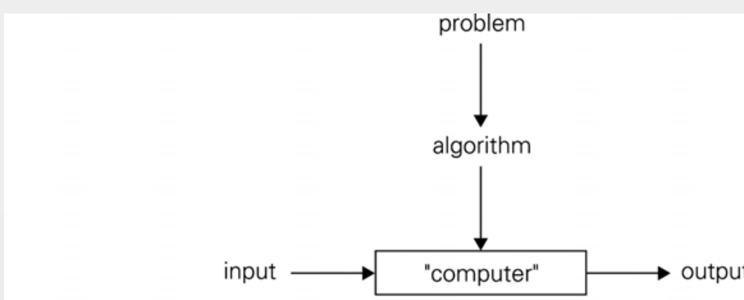

FIGURE 1.1 Notion of algorithm

Algorithm design Process

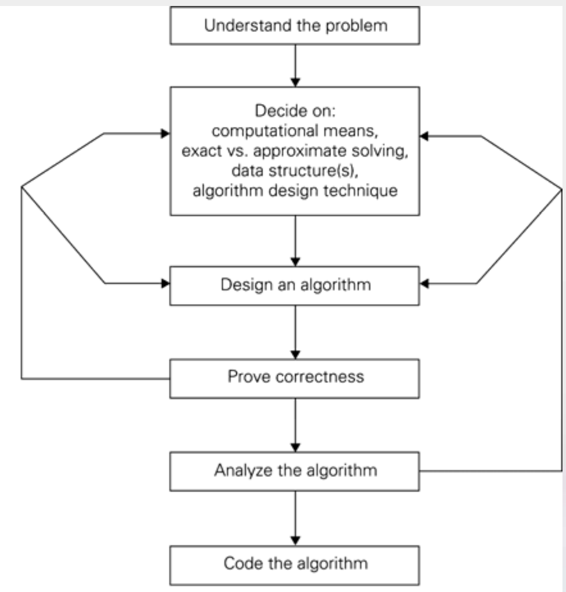

FIGURE 1.2 Algorithm design and analysis process

Interlude

- 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

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 $a > b^k$

For details, see Levitin pages 490-491 [483-485] or Weiss section 7.5.3.

Grimaldi's Theorem 10.1 is a special case of the Master Theorem.

Note that page numbers in brackets refer to Levitin 2nd edition

We will use this theorem often. You should review its proof soon (Weiss's proof is a bit easier than Levitin's).

Arithmetic algorithms

- For the next few days:
  - Reading: mostly review from CSSE 230 and DISCO
  - In-class: Some review, but mainly arithmetic algorithms
    - Examples: Fibonacci numbers, addition, multiplication, exponentiation, modular arithmetic, Euclid's algorithm, extended Euclid.
  - Lots of problems to do
  - some over review material
  - Some over arithmetic algorithms.

Fibonacci Numbers

- F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)
- Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- Straightforward recursive algorithm:

```
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)
```

- Correctness is obvious. Why?

Analysis of the Recursive Algorithm

```
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)
```

- What do we count?
  - For simplicity, we count basic computer operations
- Let T(n) be the number of operations required to compute F(n).
- T(0) = 1, T(1) = 2, T(n) = T(n-1) + T(n-2) + 3
- What can we conclude about the relationship between T(n) and F(n)?
- How bad is that?
- How long to compute F(200) on an exaflop machine (10^18 operations per second)?
  - http://slashdot.org/article.pl?sid=08/02/22/040239&from=rss

A Polynomial-time algorithm?

- ```
  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]
  ```
- Correctness is obvious because it again directly implements the Fibonacci definition.
- Analysis?
- Now (if we have enough space) we can quickly compute F(14000)

A more efficient 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 $\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = X \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = X^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}, \ldots, \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = X^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$
- How many additions and multiplications of numbers are needed to compute the product of two 2x2 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 **(details on 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):  #efficiently calculate m^n
    result = identity_matrix
    # Fill in the details



    return result


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

# Test code
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]
```

Why so complicated?

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

$$f(N) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

***See Weiss, exercise 7.8**

For review, this proof is part of HW1.

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 can there be in the sum of three one-digit decimal numbers?
- Is the same result true in binary?
- Add two 6-bit positive integers (53+35):

Carry:	1			1	1	1		
		1	1	0	1	0	1	(35)
		1	0	0	0	1	1	(53)
	1	0	1	1	0	0	0	(88)

- So adding two k-bit integers is $\Theta(\quad)$ time.

Multiplication of Two k-bit Integers

- Example: multiply 13 by 11

```
              1  1  0  1
           x  1  0  1  1
------------------------
              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)
------------------------
  1  0  0  0  1  1  1  1   (binary 143)
```

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