COL 106 – Data Structures and Algorithms

Asymptotic Analysis & Program Correctness

Announcements:

(Written)

- Quiz 1 on Friday (11 - 11:50 am)
- Venue: LH 325, LH 121
- Seating plan: on Piazza (by Wednesday night)
- Reminder: Lab Quiz 1 on Saturday. (9 - 11 am
  11 am - 1 pm)

Primality Testing

Input: $\boxed{n \in \mathbb{N}}$

Output: yes if n is prime

Factoring

Check-Prime(n)

1. if ($n \leq 1$) the return("not prime")
2. for ($i = 2$; $i < n$; i++)

   if (i divides n) then return("not prime")
3. return("prime")

$O(n)$ time algorithm

Check-Prime(n)

1. if ($n \leqslant 1$) then return("not prime")
2. set $i = 2$
3. while ($i \times i \leqslant n$)

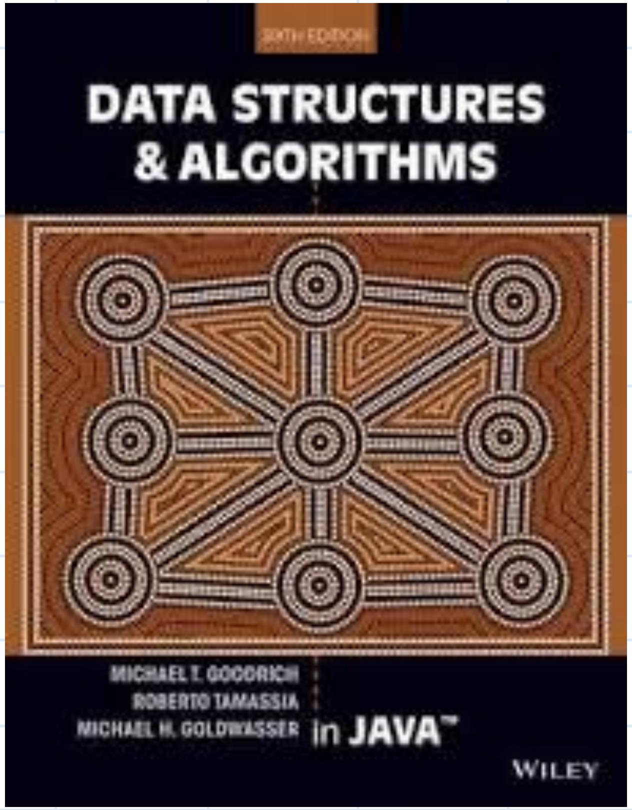
   if (i divides n) then return("not prime")

   $i = i + 1$
4. return("prime")

$O(\sqrt{n})$ algorithm

- What is the input size?
- What is the running time?

$(\lceil \log_2 n \rceil + 1)^2$ — bits



C-4.47 Communication security is extremely important in computer networks, and one way many network protocols achieve security is to encrypt messages. Typical *cryptographic* schemes for the secure transmission of messages over such networks are based on the fact that no efficient algorithms are known for factoring large integers. Hence, if we can represent a secret message by a large prime number p, we can transmit, over the network, the number $r = p \cdot q$, where $q > p$ is another large prime number that acts as the *encryption key*. An eavesdropper who obtains the transmitted number r on the network would have to factor r in order to figure out the secret message p.

Using factoring to figure out a message is hard without knowing the encryption key q. To understand why, consider the following naive factoring algorithm:

```
for (int p=2; p < r; p++)
  if (r % p == 0)
    return "The secret message is p!";
```

a. Suppose the eavesdropper's computer can divide two 100-bit integers in μs (1 millionth of a second). Estimate the worst-case time to decipher the secret message p if the transmitted message r has 100 bits.

b. What is the worst-case time complexity of the above algorithm? Since the input to the algorithm is just one large number r, assume that the input size n is the number of bytes needed to store r, that is, $n = \lfloor (\log_2 r)/8 \rfloor + 1$, and that each division takes time $O(n)$.

Greatest Common Divisors (Highest common factor)

$gcd(15, 12) = 3$

$gcd(11, 20) = 1$

$m = p \times q$

$n = p \times l$

$O(\sqrt{n})$ time

- How do you compute gcd?

input: $m, n \in \mathbb{N}$

output: $gcd(m, n)$

$O(\sqrt{a}) + O(\sqrt{b})$

$$\underbrace{\min(\log_2 x,\ \log_2 y)}$$

EuclideanGCD(a, b)

```
int  x, y, z;
x = a; y = b;
while (y > 0) {
    z = x mod y
    x = y
    y = z
}
return x;
```

$x = a$

$y = b$

input size? $\left(\lceil \log_2 a \rceil + 1 + \lceil \log_2 b \rceil + 1 \right)$

$x \geq y$

$a \quad b$

$\gcd(15, 12)$

$z = 15 \bmod 12 = 3$

$x = 12$

$y = 3$

$(x \bmod y,\ y)$

$\boxed{x \bmod y}$

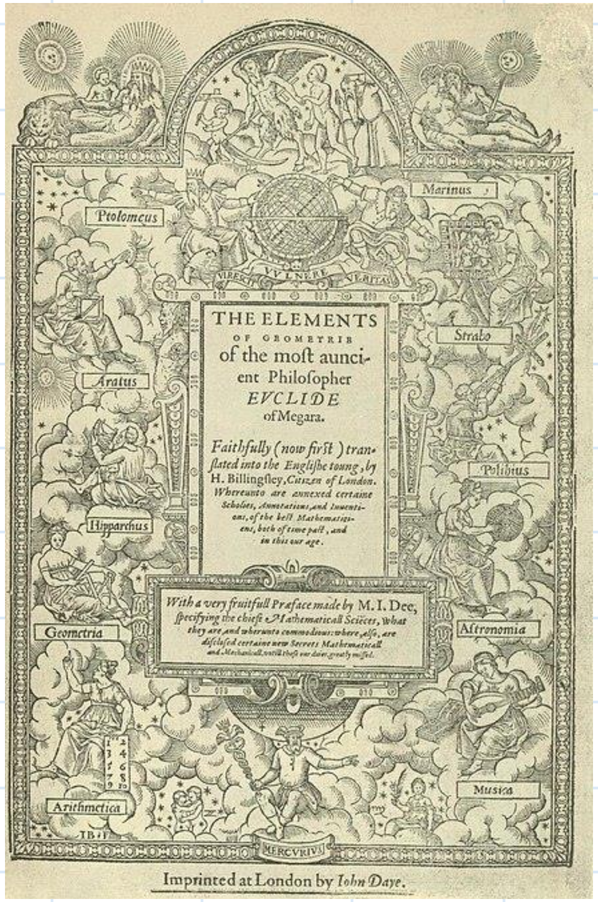
$y < x/2$

$\leq x/2$

$y \geq x/2$

$x \bmod y = (x - y)$

Running time: ?


THE ELEMENTS
OF GEOMETRIE
of the most aunci-
ent Philosopher
EVCLIDE
of Megara.

Faithfully (now first) trans-
lated into the Englishe toung, by
H. Billingsley, Citizen of London.

Imprinted at London by Iohn Daye.

Euclidean GCD(a,b)

```
int x, y, z;
x = a; y = b;
while (y > 0) {
    z = x mod y
    x = y
    y = z
}
return x;
```

gcd(x,y)

gcd(x,y) = gcd(a,b) = gcd(b, a mod b)

gcd(y, a mod b)

(y, x mod y)

x	a	b
y	b	a mod b

$\boxed{\log_2 a}$

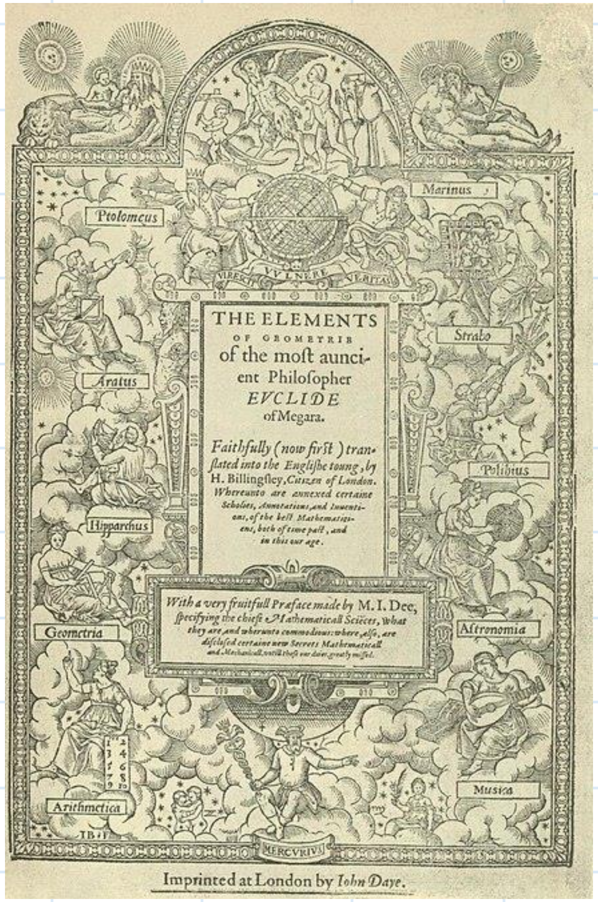
$O(\log_2 a)$ iterations

$a = lp$

$b = lq$

$a - b = l(p-q)$

Obs: If l divides a & b, then it also divides $a-b$ (assuming $a > b$)



THE ELEMENTS OF GEOMETRIE of the most auncient Philosopher EVCLIDE of Megara.

Faithfully (now first) translated into the Englishe toung, by H. Billingsley, Citizen of London. Whereunto are annexed certaine Scholies, Annotations, and Inventions, of the best Mathematiciens, both of time past, and in this our age.

With a very fruitfull Praeface made by M. I. Dee, specifying the chiefe Mathematicall Sciences, what they are, and whereunto commodious: where, also, are disclosed certaine new Secrets Mathematicall and Mechanicall, untill these our daies, greatly missed.

Imprinted at London by Iohn Daye.

Idea: Instead of computing $\gcd(a, b)$ might as well compute $\gcd(b, a-b)$

Idea: Instead of computing $\underbrace{\gcd(a, b)}$ might as well compute $\gcd(b, a-b)$

$\gcd(a-b, a-2b)$

$\vdots$

$\gcd(a,b) = \underbrace{\gcd(b, a \bmod b)}$ as long as $\neq 0$

$\boxed{\text{Theorem (Euclid):}}$ If $a > b > 0$ are positive integers, then if b divides a then $GCD(a,b) = b$.

Otherwise, $\gcd(a,b) = \gcd(a \bmod b, b)$

Idea: Instead of computing $gcd(a, b)$ might as well compute $gcd(b, a\ b)$

$gcd(a-b, a-2b)$

$\vdots$

$gcd(b, a \bmod b)$

as long as $\neq 0$

Correctness: Invariant condition of the algorithm:

$$gcd(x, y) = gcd(a, b)$$

Correctness via Invariants (Robert Floyd)

$P(1)$

$P(i) \Rightarrow P(i+1)$

① **Precondition:** Something that is true before the start of the loop.

$P(1)$

$P(2)$

② **Postcondition:** Something that is true after the end of the loop.

③ **Loop Invariant:** If true at step i, then is also true if the loop is executed once more.

Fend the largest element in an array

Algorithm arrayMax (A, n)

Input : array A with n elements

Output : largest element in the array

```
currentMax ← A[0];
for i ← 1 to n-1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

Fend the largest element in an array

i/p: Program (P, X)
o/p: Does P halt on x

Halting Problem

Algorithm arrayMax (A, n)
Input : array A with n elements
Output : largest element in the array

```
currentMax ← A[0];
for i ← 1 to n-1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

Precondition:
currentMax contains A[0]

Loop Invariant:
If currentMax contains maximum of A[0, ... i] & the loop is executed once more it contains max of A[0, ... i+1]

Postcondition: CurrentMax contains max of A

0, 1, 1, 2, 3, 5, 8, 13, - - - - .

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0\,. \end{cases}$$

Fibonacci Numbers

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0\,. \end{cases}$$

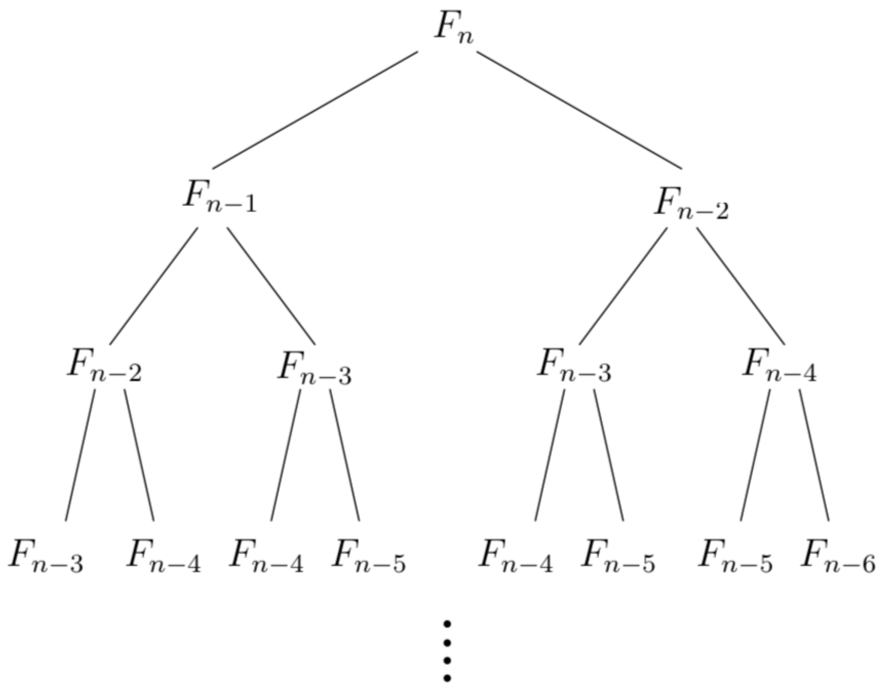
```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n-1) + fib1(n-2)
```

- Running time of fib1?

Fibonacci Numbers

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0\,. \end{cases}$$

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n-1) + fib1(n-2)
```



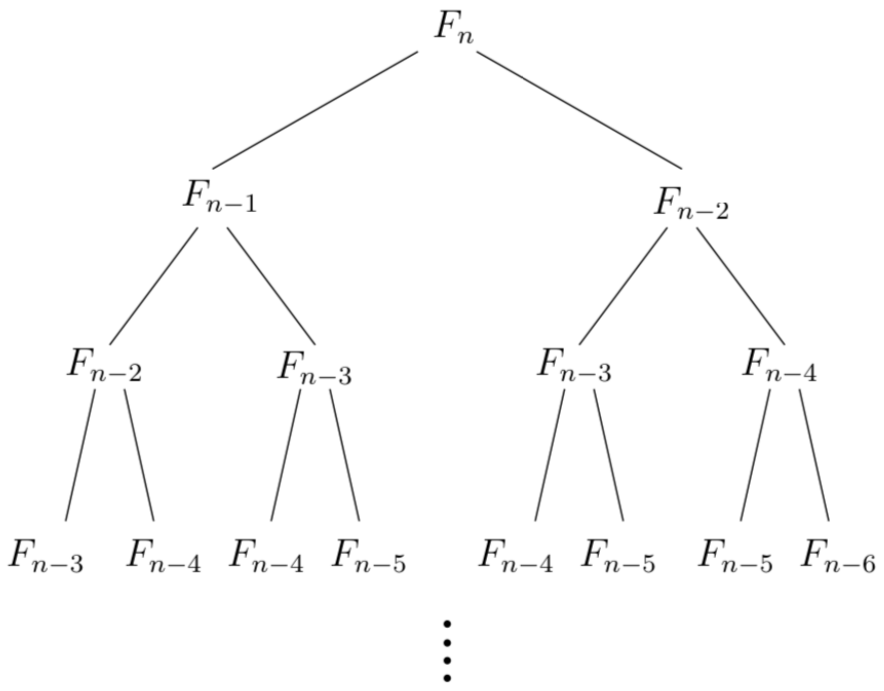
$$T(n) = T(n-1) + T(n-2)$$

- Running time of fib1?

How fast does $T(n)$ grow?

Fibonacci Numbers

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0\,. \end{cases}$$




```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n − 1) + fib1(n − 2)
```

$$T(n) = T(n-1) + T(n-2)$$

- Running time of fib1?

$$\left(\frac{3}{2}\right)^n \leq T(n) \leq 2^n$$

A better algorithm

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i-1] + f[i-2]
return f[n]
```

- Running time?
- Can you do better?

Question:

What is the largest Fibonacci number that can fit into a word in 64-bit machine?
128-bit

Algorithm Binary Search (A, n, T)

Inputs: A is an array of n sorted elements.
T is the element value we want to locate

Output: index of the array element with value T.
if it is present, else return NIL.

BINARY SEARCH

1	2	3	4	5	6	7	8	9	10	11
2	5	8	12	16	23	46	59	60	150	200
					23					

ITERATIVE-BINARY-SEARCH (A, v)

```
low = A[1]
high = A[A.length]
while low ≤ high
    mid = ⌊(low + high)/2⌋
    if v == A[mid]
        return mid
    elseif v > A[mid]
        low = mid + 1
    else
        high = mid - 1
return NIL
```

- What is the time Complexity of Binary Search? $\underline{O(\log n)}$
- Does it change with the data structure used to implement A?
- Loop Invariant?