LECTURE 1

FIBONACCI SEQUENCE

1, 1, 2, 3, 5, 8, 13, 21, 34 . . . .

is defined by

$$F_n = F_{n-1} + F_{n-2}$$

(every number in sequence is sum of previous two)

Recursive Implementation

Here is a recursive algorithm to compute the n^{th} Fibonacci number

```
F (n : integer)
    if n = 1    return 1
    if n = 2    return 1
    return  F(n-1) + F(n-2)  }
```

We will now calculate the time complexity of algorithm (a.k.a. How long does it take on input n)

Define

$T[n]$ = # of operations in the execution of $F(n)$

We do not want to compute $T[n]$ exactly BECAUSE

1) It's too HARD

2) It depends on details of machine / programming language, et.c

We need notation for "crude approximation"

BIG OH-NOTATION

Definition: Given two non-negative functions $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N} \to \mathbb{N}$

we say $f = O(g)$ iff

$\iff \exists$ a constant c, such that $f(n) \leq c \cdot g(n)$ $\forall n \in \mathbb{N}$

Intuitively, f is growing NO FASTER than g

Examples:

1) $4n^4 + 9n^3 + 27n^2 + 15 = O(n^4)$

RULE 1: In BigOh, we can drop all multiplicative constants

RULE 2: Among polynomials n, n^2, n^3: ignore all but the highest degree term.

2) $n^{100} = O(2^n)$

RULE 3: All polynomials (of fixed degree) are O(exponential)

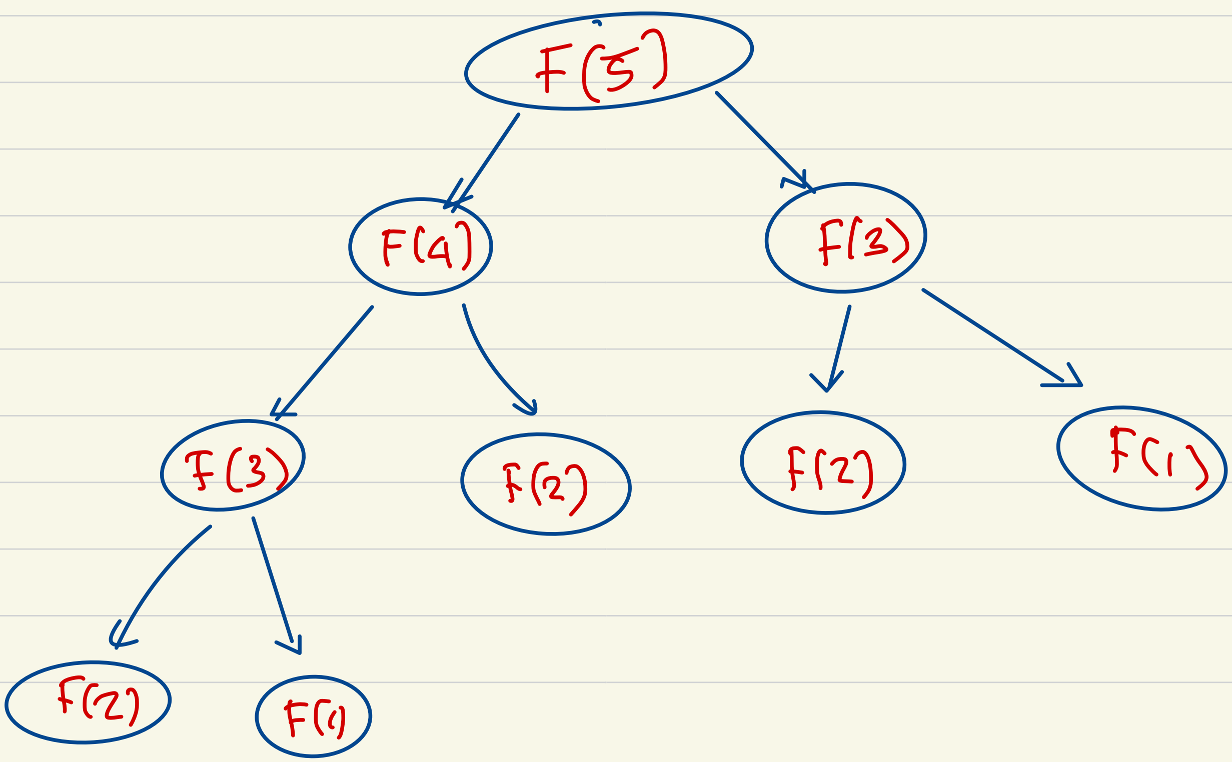
3) $(\log n)^{20} = O(n)$

RULE 4: All logarithms & fixed degree polys in logarithms are O(any polynomial)

RUNTIME ANALYSIS FOR: RECURSIVE FIBONACCI

Define $T[n]$ = # of operations in recursive Fibonacci on input n

To gain intuition, let us look at the function calls in $F[5]$



Execution of $F(n)$

$\leftarrow$ Compute $F(n-1)$ $\longrightarrow$ $T[n-1]$ operations

Compute $F(n-2)$ $\longrightarrow$ $T[n-2]$ operations

Add the results

& return

Total # of operations

$$T(n) = T[n-1] + T[n-2] + \left(\begin{array}{c}\text{Addition} \\ \& \\ \text{returning}\end{array}\right)$$

$$\Rightarrow \quad T[n] \geq T[n-1] + T[n-2]$$

The runtime $T[n]$ is really large, in fact grows exponentially in n

To see this we will show

Claim 1: $T[n] \geq \Omega\left(\left(\frac{3}{2}\right)^n\right)$

Proof: By induction, we will show $T[n] > \left(\frac{1}{4}\right)\left(\frac{3}{2}\right)^n$

for $n = 1, 2$, $T[1], T[2]$ are ≥ 1

$$T(1) \geq \frac{1}{4}\left(\frac{3}{2}\right) \qquad T(2) \geq \left(\frac{1}{4}\right)\left(\frac{3}{2}\right)^2$$

$$T[n] \geq T[n-1] + T[n-2]$$

$$\geq \frac{1}{4}\left(3/2\right)^{n-1} + 1/4\left(3/2\right)^{n-2}$$

[By induction hypothesis]

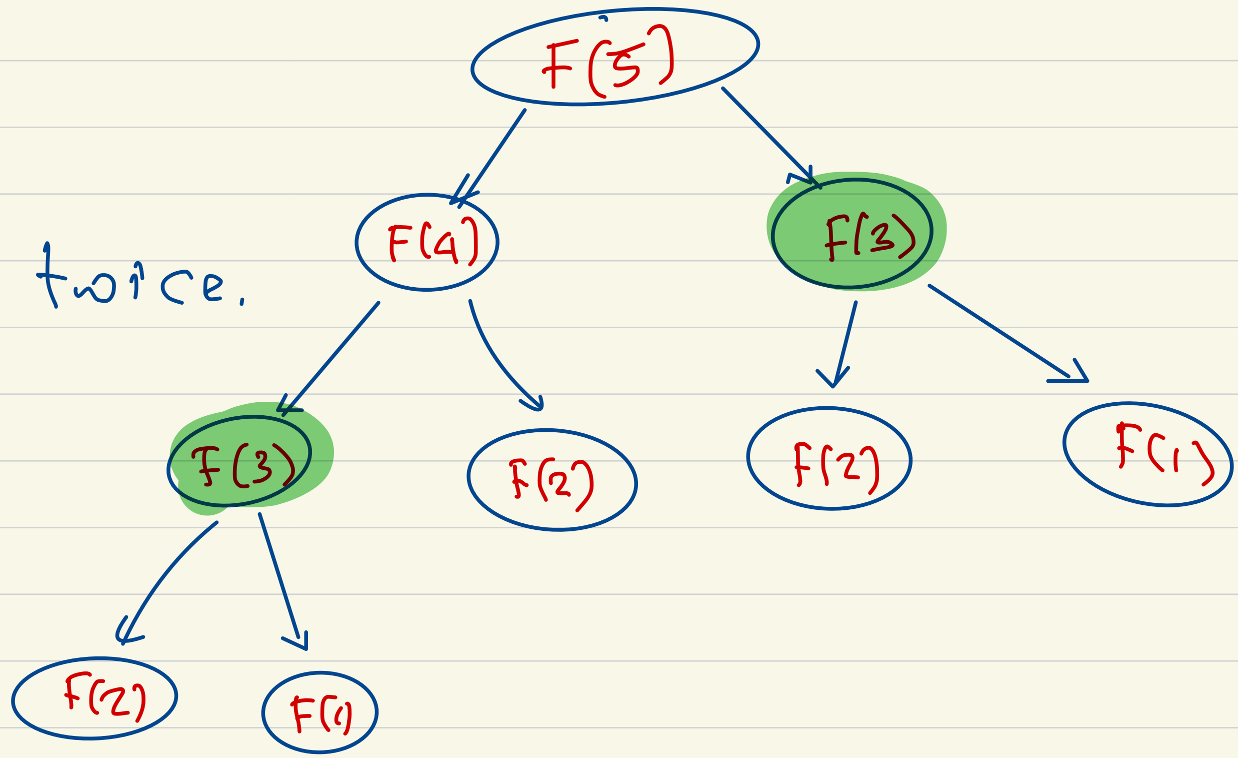
$$> \frac{1}{4}\left[3/2\right]^{n-2}\left[\left(3/2\right) + 1\right]$$

$$> 1/4\left(3/2\right)^{n-2} \cdot \left(3/2\right)^2 = 1/4\left(3/2\right)^n$$

Recursive Fibonacci is very inefficient: because it computes the same quantities, again & again & again.

For example

F(3) is computed twice.



To improve the algorithm, we store & reuse the values

ITERATIVE FIBONACCI

```
FIB( n : integer )
        f[1...n] : an array of n integers
        f[1] = 1
        f[2] = 1
        for i=3 to n
            f[i] = f[i-1] + f[i-2]
```

ADDITION

RUNTIME ANALYSIS FOR: ITERATIVE FIBONACCI

$$T(n) = O\left(n * \left(\begin{matrix}\text{time for one} \\ \text{addition}\end{matrix}\right)\right)$$

Suppose we use a fixed size integer (say 64 bit)

to store $f(n)$ then each addition is

one machine instruction $\Rightarrow$ 1 addition = 1 operation

and $T(n) = O(n)$

However can we really use a 64 bit integer to store $f(n)$?.?

How many bits long is n^{th} Fibonacci number F_n?

Any number X has $\lceil \log_2 X \rceil$ bits long

Fact: $2^n > F_n > \left(\frac{3}{2}\right)^n$

[REMARK: Can be proved by induction, just like Claim 1]

$$\Rightarrow \quad n \geq \log_2 F_n \geq n \log_2(1.5) = (0.58)\,n$$

$\Rightarrow$ F_n is between $(0.58)n$ to n bits long

For example, $F_{1000} \gg 580$ bits long.

So we can't use 64 bit integers to store F_n.

We need BIG INTEGER data type,

i.e. store $f(n)$ as a sequence of bits/digits in an array

BIG INTEGER

- Integer stored as a sequence of digits in an array.