

# COMPLEXITY ANALYSIS OF ALGORITHMS

---

Problem Solving with Computers-II

C++

```
#include <iostream>
using namespace std;

int main() {
    cout<<"Hola Facebook\n";
    return 0;
}
```



Join iclicker at <https://join.iclicker.com/ZHLY>

# Problem: Fibonacci Numbers

## Definition:

The Fibonacci numbers are the sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55,...

Defined by

$$F_0 = F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

Problem: Given  $n$ , compute  $F_n$ .

# Which implementation is significantly faster ?

A.

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```

B.

```
F(int n){  
    Initialize A[0 . . . n]  
    A[0] = A[1] = 1  
  
    for i = 2 : n  
        A[i] = A[i-1] + A[i-2]  
  
    return A[n]  
}
```

C. *Both are almost equally fast*

# Which implementation is significantly faster ?

A.

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```

B.

```
F(int n){  
    Initialize A[0 . . . n]  
    A[0] = A[1] = 1  
  
    for i = 2 : n  
        A[i] = A[i-1] + A[i-2]  
  
    return A[n]  
}
```

C. *Both are almost equally fast*

The “right” question is: How does the running time grow?

E.g. How long does it take to compute  $F(200)$  recursively?

....let's say on....a supercomputer that can compute 40 trillion operations per sec

How long does it take to compute  $\text{Fib}(200)$  recursively?

....let's say on.... a supercomputer that runs 40 trillion operations per second

It will take approximately  $2^{92}$  seconds to compute  $F_{200}$ .

Time in seconds

Interpretation

$2^{10}$

17 minutes

$2^{20}$

12 days

$2^{30}$

32 years

$2^{40}$

35000 years  
(cave paintings)

$2^{50}$

35 million years ago

$2^{70}$

Big Bang

**What is the main takeaway so far?**

How long does it take to compute  $\text{Fib}(200)$  recursively?

....let's say on.... a supercomputer that runs 40 trillion operations per second

It will take approximately  $2^{92}$  seconds to compute  $F_{200}$ .

### Time in seconds

$2^{10}$

$2^{20}$

$2^{30}$

$2^{40}$

$2^{50}$

$2^{70}$

### Interpretation

17 minutes

12 days

32 years

35000 years  
(cave paintings)

35 million years ago

Big Bang

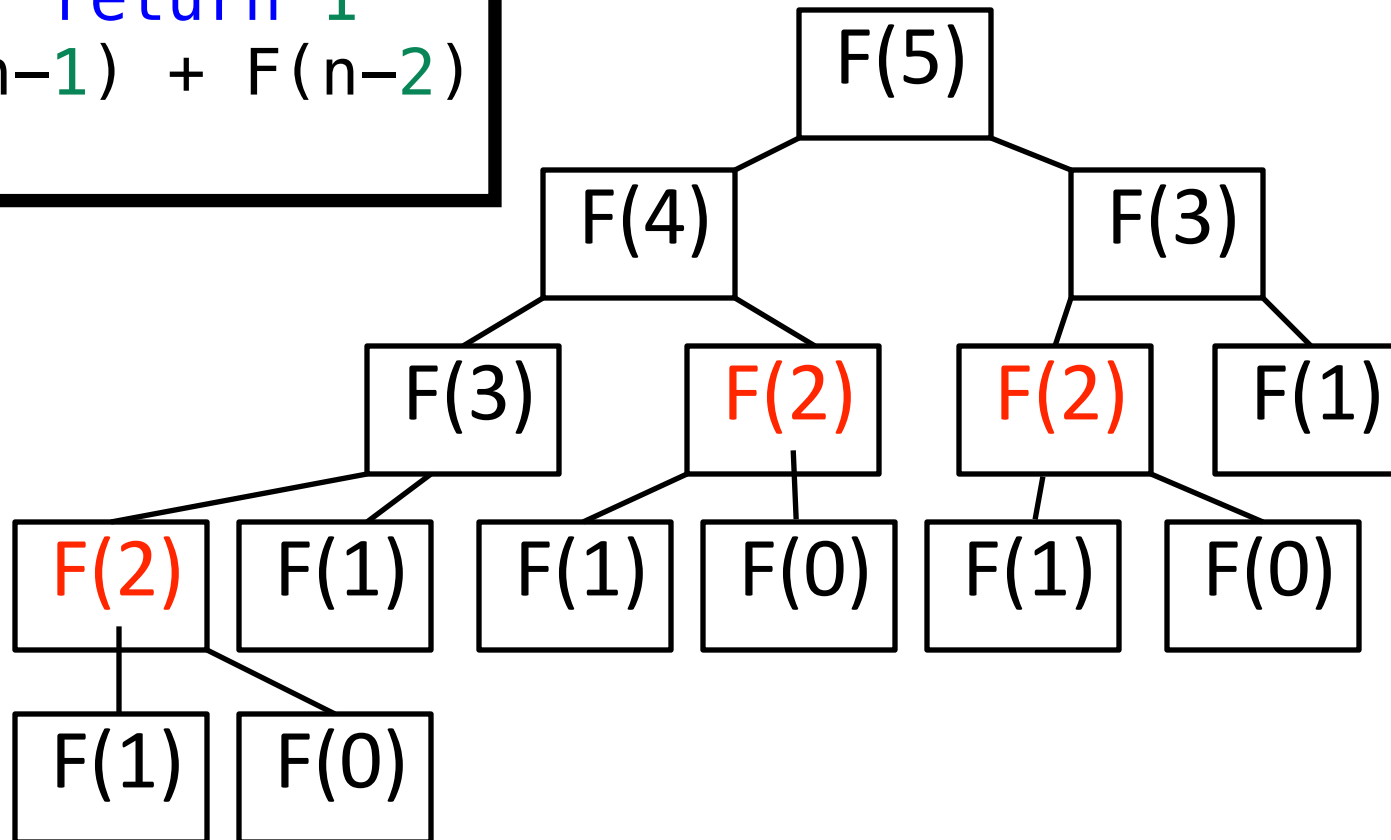
### Questions of interest:

- Why is Algo A so slow?
- How do we quantify efficiency?
- Is Algo A better than Algo B?
- When will my code finish running?

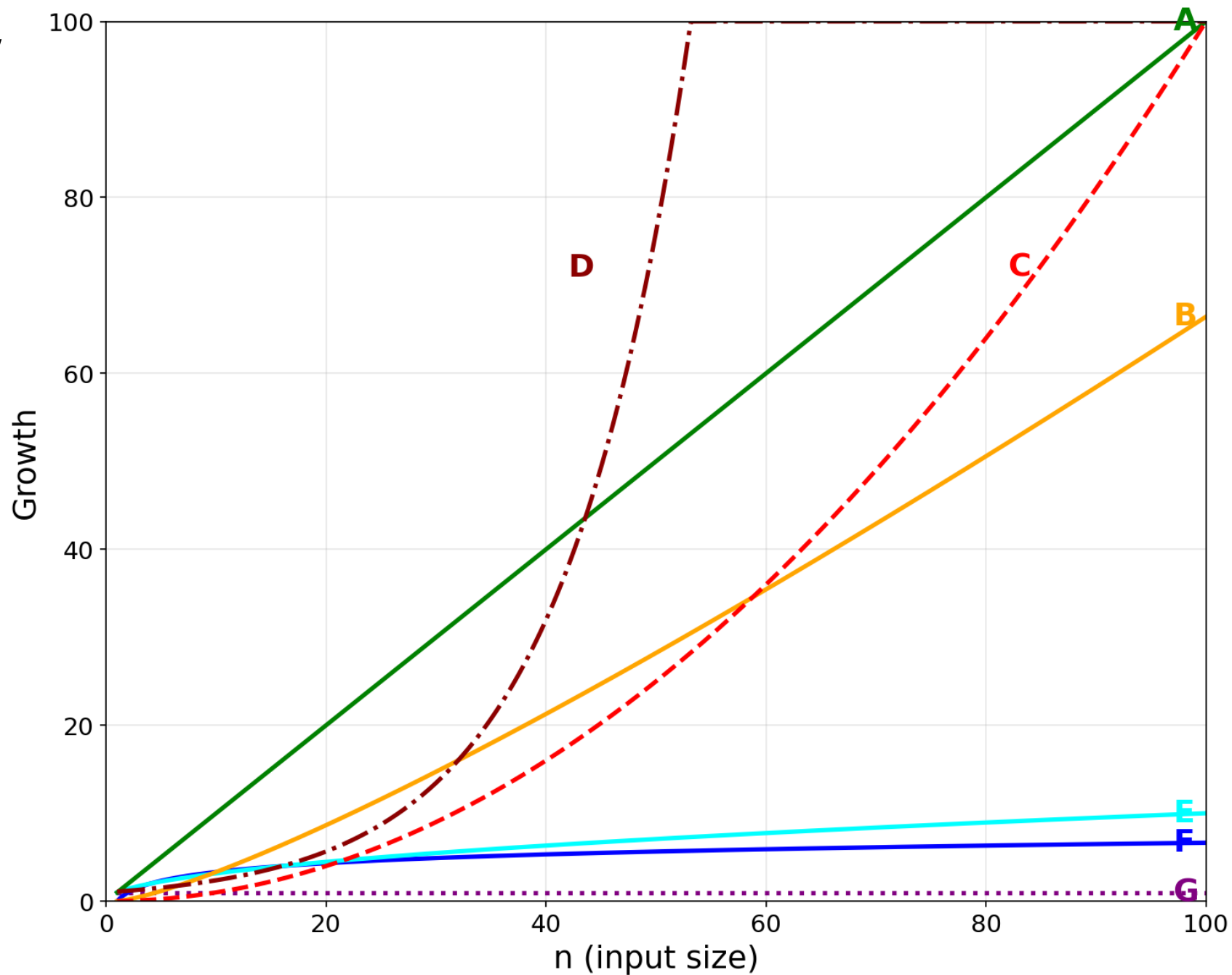
# Why So Slow?

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```

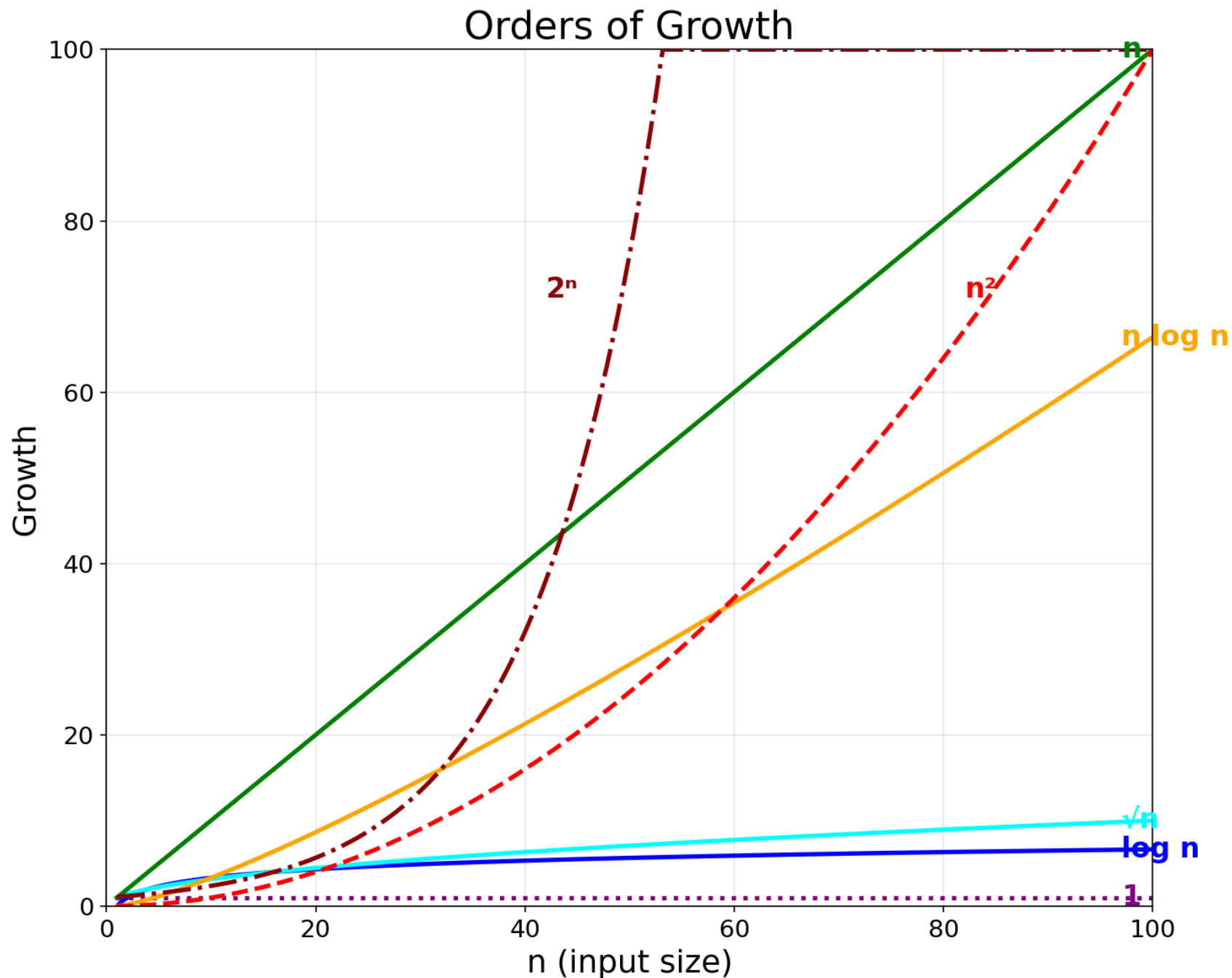
Too many recursive calls.



Which curve represents how the recursive fibonacci function grows?



- An **order of growth** is a set of functions whose growth behavior is considered equivalent.
- Functions that grown similarly belong to the same order of growth



## ***ORDERS OF GROWTH ACTIVITY***

***1. Rank these functions from SMALLEST to LARGEST growth order***

***100***

***n***

***50n***

***2n<sup>2</sup>***

***n log n***

***2<sup>n</sup>***

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

***2. Which functions belong to the SAME order of growth?***

\_\_\_\_\_

***3. The recursive Fibonacci has \_\_\_\_\_ order of growth.***

***4. The iterative Fibonacci has \_\_\_\_\_ order of growth.***

# Big-O: Notation to name the order of growth

<i>Order of Growth</i>	<i>Big-O Notation</i>
<i>Constant</i>	
<i>Logarithmic</i>	
<i>Linear</i>	
<i>Linearithmic</i>	
<i>Quadratic</i>	
<i>Exponential</i>	

- $50n$  and  $n$  are both  $O(n)$  — same order of growth.
- Big-O captures the growth rate, ignoring constants.

## Express in Big-O notation

1. 100000000
2.  $3n$
3.  $6n-2$
4.  $15n + 44$
5.  $50n\log(n)$
6.  $n^2$
7.  $n^2-6n+9$
8.  $3n^2+4*\log(n)+1000$
9.  $3^n + n^3 + \log(3*n)$

### Common sense rules

1. Multiplicative constants can be omitted:  
 $14n^2$  becomes  $n^2$ .
2.  $n^a$  dominates  $n^b$  if  $a > b$ : for instance,  $n^2$  dominates  $n$ .
3. Any exponential dominates any polynomial:  
 $3^n$  dominates  $n^5$  (it even dominates  $2^n$ ).

**For polynomials, use only leading term, ignore coefficients: linear, quadratic**

## Big O running time analysis: clicker

```
/* n is the length of the array*/  
int sum(int arr[], int n)  
{  
    int result = 0;  
    for(int i = 0; i < n; i+=2)  
        result+=arr[i];  
    return result;  
}
```

- A.  $O(n^2)$
- B.  $O(n)$
- C.  $O(n/2)$
- D.  $O(\log n)$
- E. None of the above

Join iclicker at <https://join.iclicker.com/ZHLY>

# Iterative Fibonacci Algorithm

$T(n)$  : running time of  $F(n)$

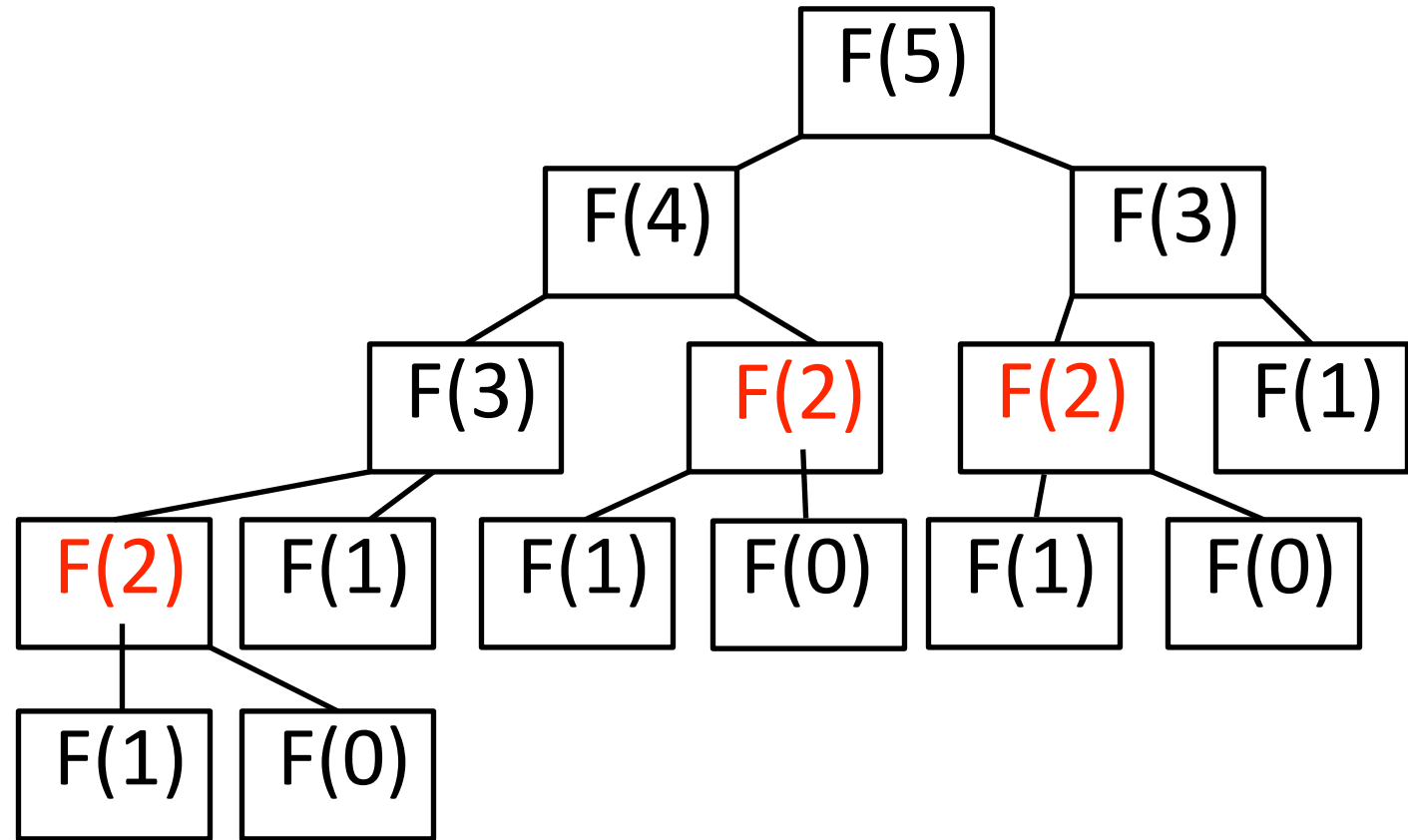
**: number of primitive operations to execute  $F(n)$**

```
F(int n){  
    Initialize A[0 . . . n]  
    A[0] = A[1] = 1  
  
    for i = 2 : n  
        A[i] = A[i-1] + A[i-2]  
  
    return A[n]  
}
```

Derive  $T(n) = O(2^n)$

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```

# Derive $T(n) = O(2^n)$



# Space Complexity

$S(n)$  = auxiliary memory needed to compute  $F(n)$

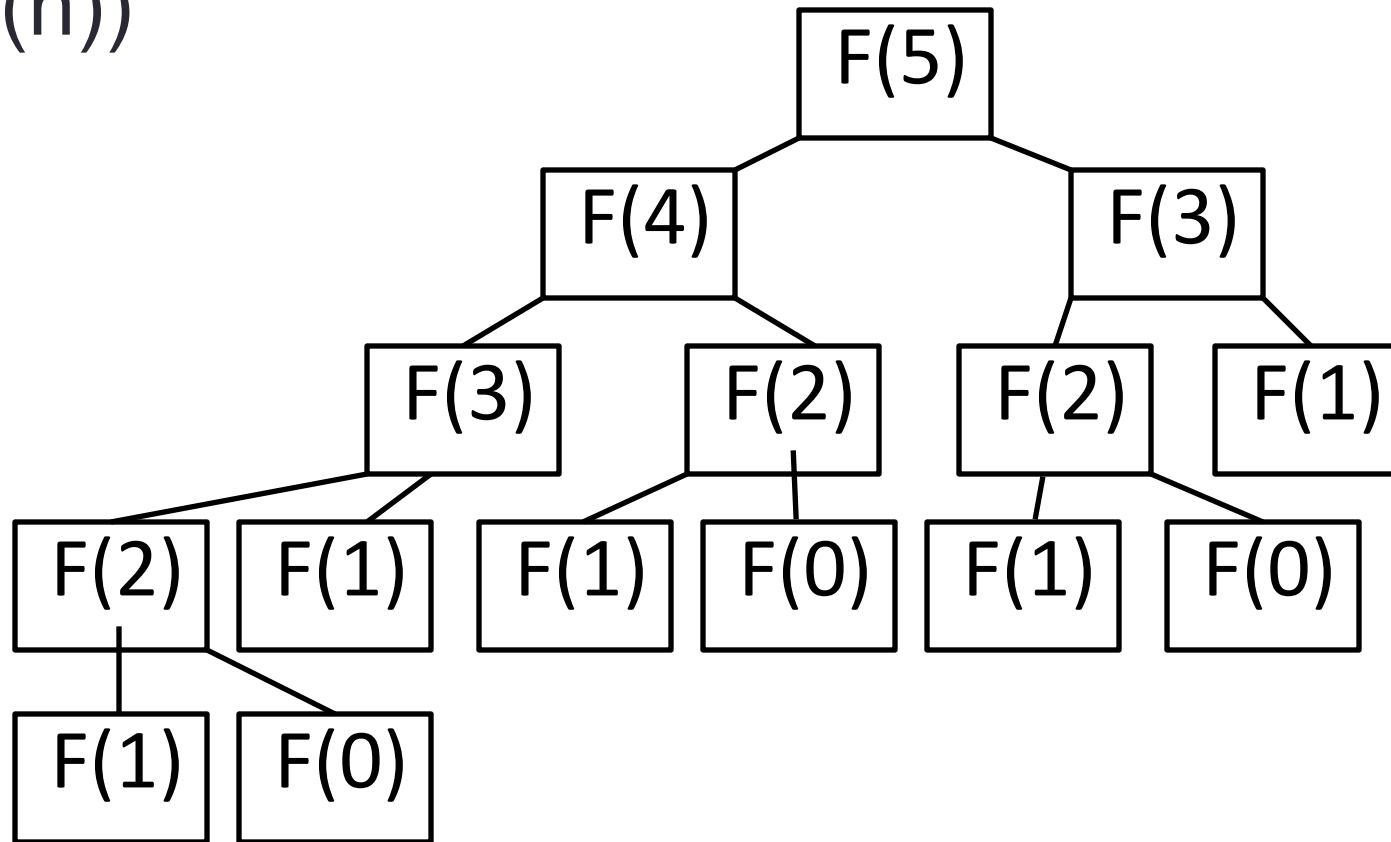
In general space complexity includes space to store inputs + auxiliary space. But for this class assume auxiliary space only

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```

What is  $S(n)$ ? Express your answer in Big-O notation

What is  $S(n)$ ? Express your answer in Big-O notation

- A.  $O(1)$
- B.  $O(\log(n))$
- C.  $O(n)$
- D.  $O(n^2)$
- E.  $O(2^n)$



Tree of recursive calls needed to compute  $F(5)$

$S(n)$  relates to maximum depth of the recursion

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```

F(5)

$S(n)$  relates to maximum depth of the recursion

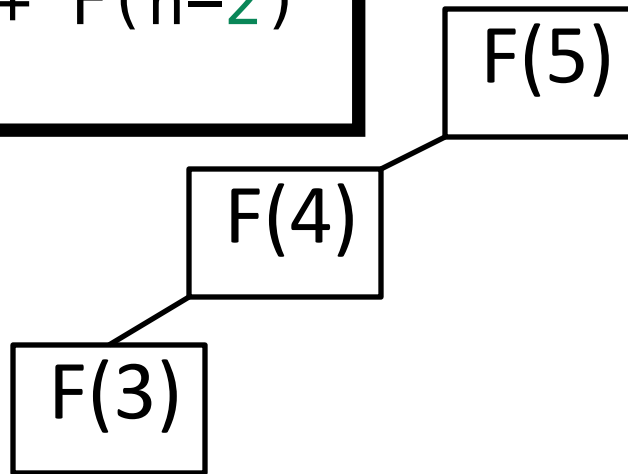
```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```

F(5)

F(4)

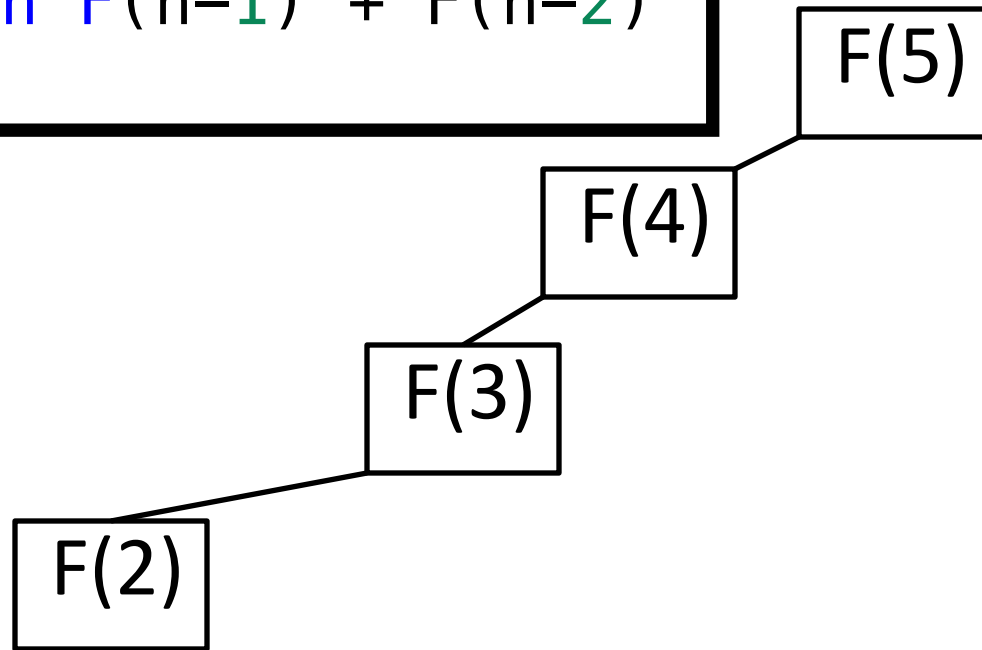
$S(n)$  relates to maximum depth of the recursion

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



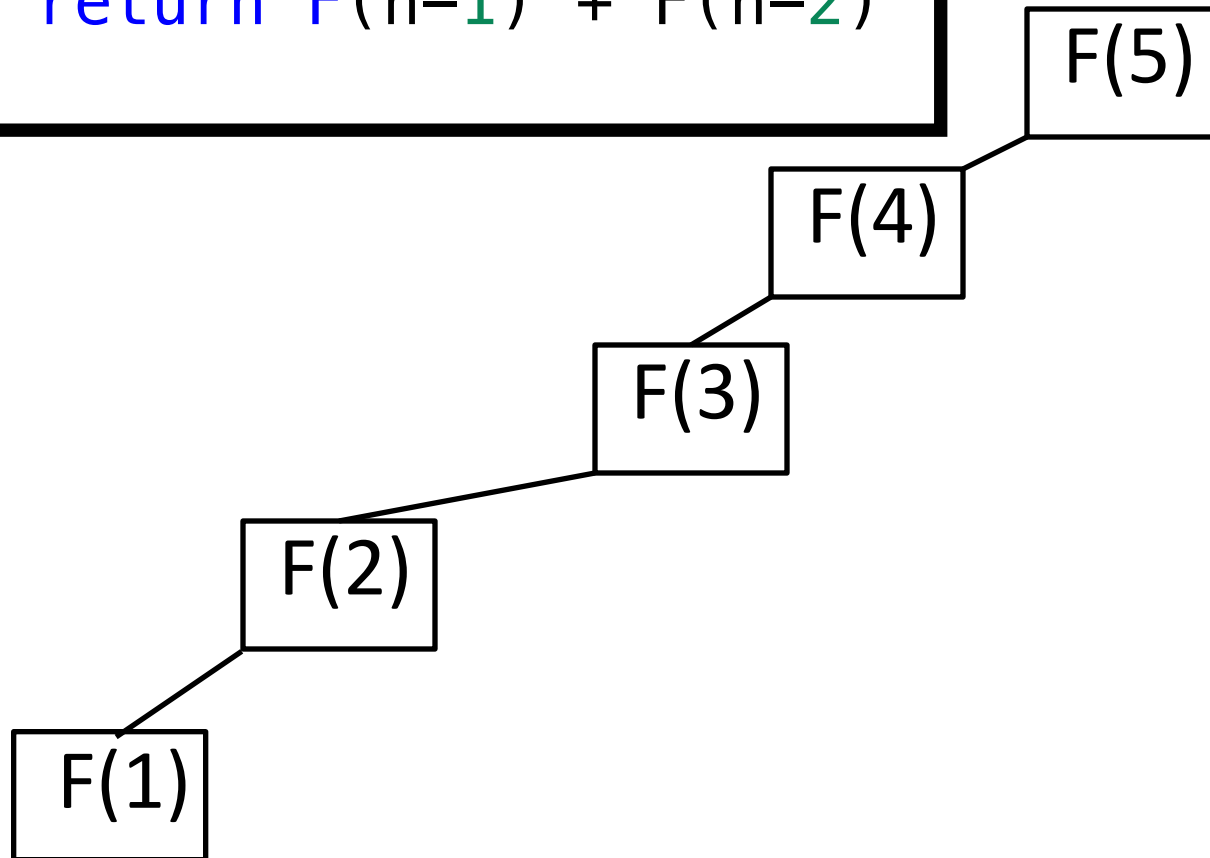
$S(n)$  relates to maximum depth of the recursion

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



$S(n)$  relates to maximum depth of the recursion

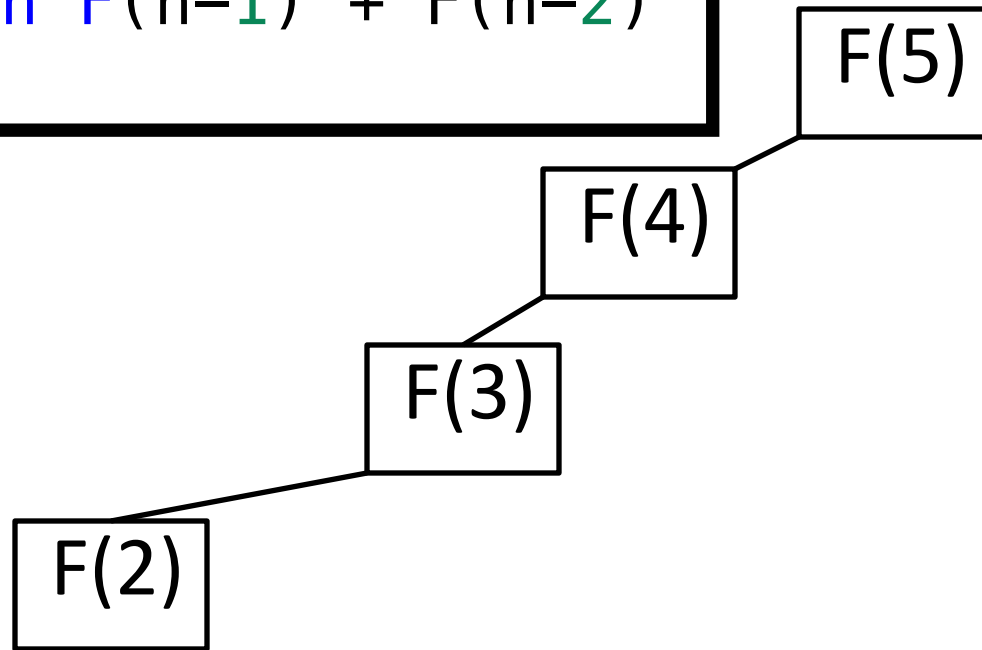
```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



Maximum depth of the recursion = 5

$S(n)$  relates to maximum depth of the recursion

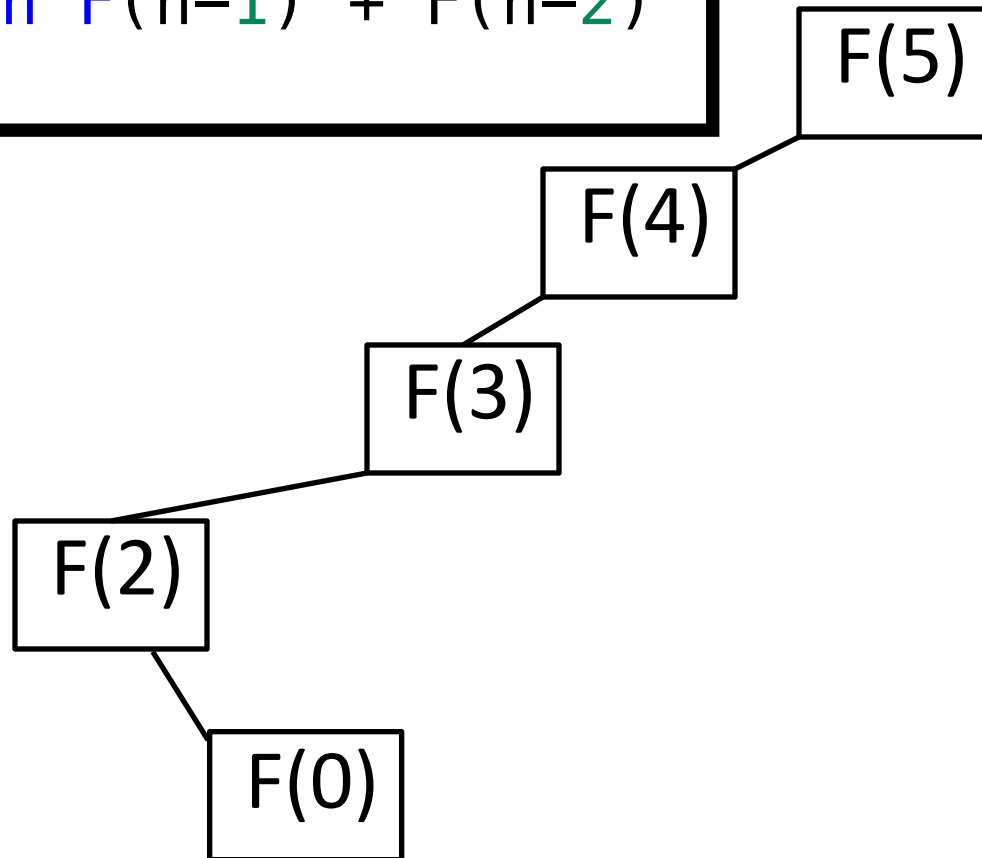
```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



Maximum depth of the recursion = 5

$S(n)$  relates to maximum depth of the recursion

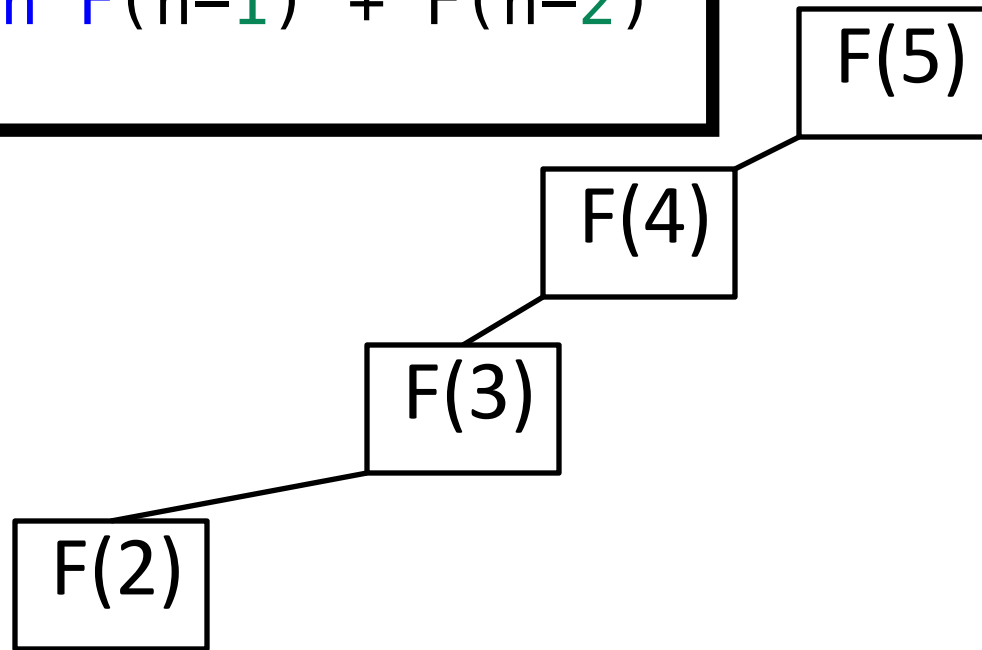
```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



Maximum depth of the recursion = 5

$S(n)$  relates to maximum depth of the recursion

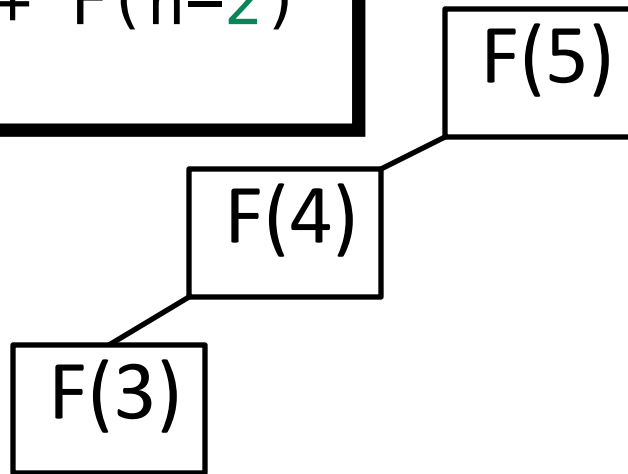
```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



Maximum depth of the recursion = 5

$S(n)$  relates to maximum depth of the recursion

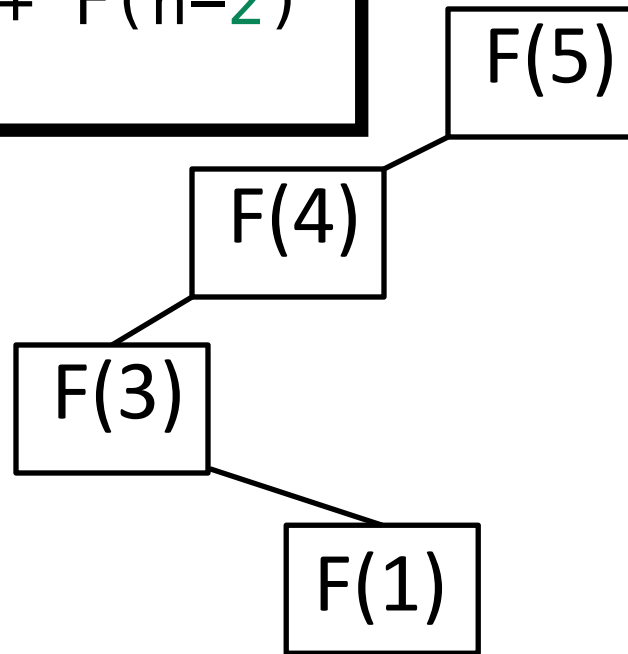
```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



Maximum depth of the recursion = 5

$S(n)$  relates to maximum depth of the recursion

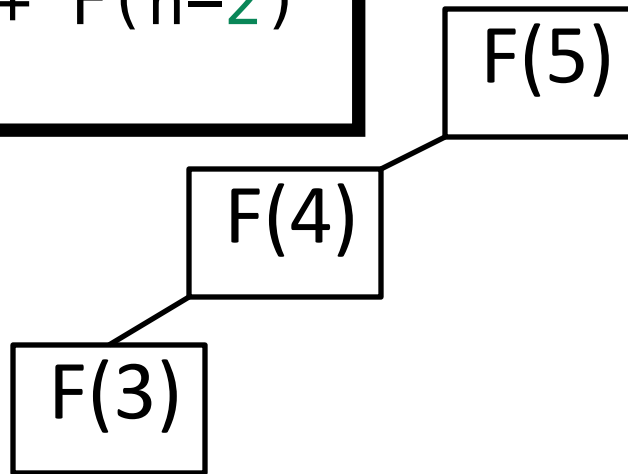
```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



Maximum depth of the recursion = 5

$S(n)$  relates to maximum depth of the recursion

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



Maximum depth of the recursion = 5

$S(n)$  relates to maximum depth of the recursion

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```

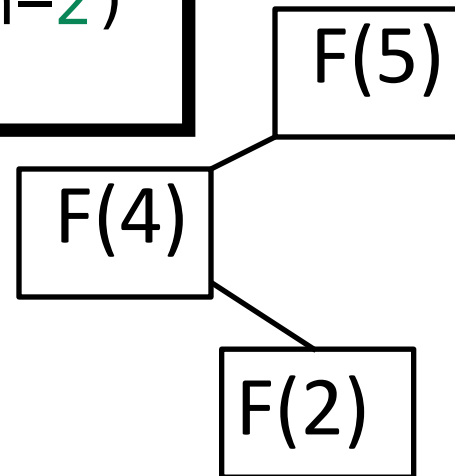
F(5)

F(4)

Maximum depth of the recursion = 5

$S(n)$  relates to maximum depth of the recursion

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



Maximum depth of the recursion = 5

$S(n)$  relates to maximum depth of the recursion

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```

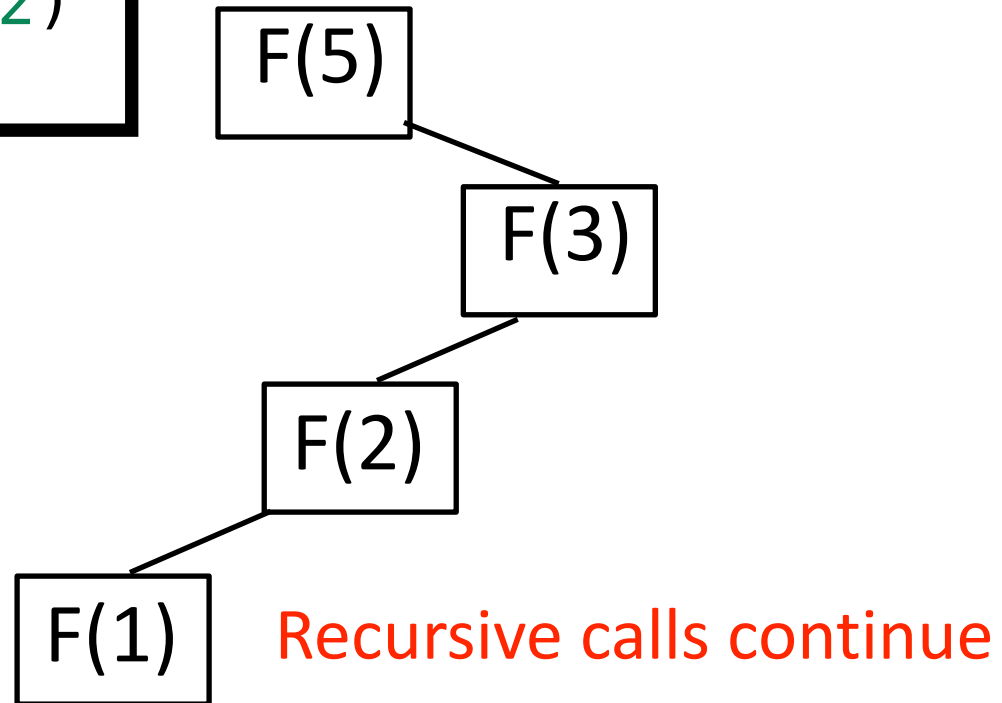
F(5)

F(4)

Maximum depth of the recursion = 5

$S(n)$  relates to maximum depth of the recursion

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```



Maximum depth of the recursion for  $F(n) = n$

Therefore,  $S(n) = O(n)$

# Which algorithm is more space efficient?

A.

```
F(int n){  
    if(n <= 1) return 1  
    return F(n-1) + F(n-2)  
}
```

B.

```
F(int n){  
    Initialize A[0 . . . n]  
    A[0] = A[1] = 1  
  
    for i = 2 : n  
        A[i] = A[i-1] + A[i-2]  
  
    return A[n]  
}
```

C. Both are the same:  $O(n)$

# Next time

- Quiz 1: Includes Lecture 1 to 3.
- 30 minutes during lecture
- Bring dark pencil or pen
- Binary Search Trees

Credits and references:

Slides based on presentations by Professors Sanjoy Das Gupta and Daniel Kane at UCSD  
<https://cseweb.ucsd.edu/~dasgupta/book/toc.pdf>