COMPLEXITY ANALYSIS OF ALGORITHMS

Problem Solving with Computers-II

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

Problem: Fibonacci Numbers

<u>Definition:</u>

The Fibonacci numbers are the sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55,…

Defined by

$F_0 = F_1 = 1$ (Base case)

$F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$ (Recursive case)

<u>Problem:</u> 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]
      A[2] = A[1] + A[0]
    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 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 Fib(200) recursively?

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

Theory - Big-O analysis

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

Insigh

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?

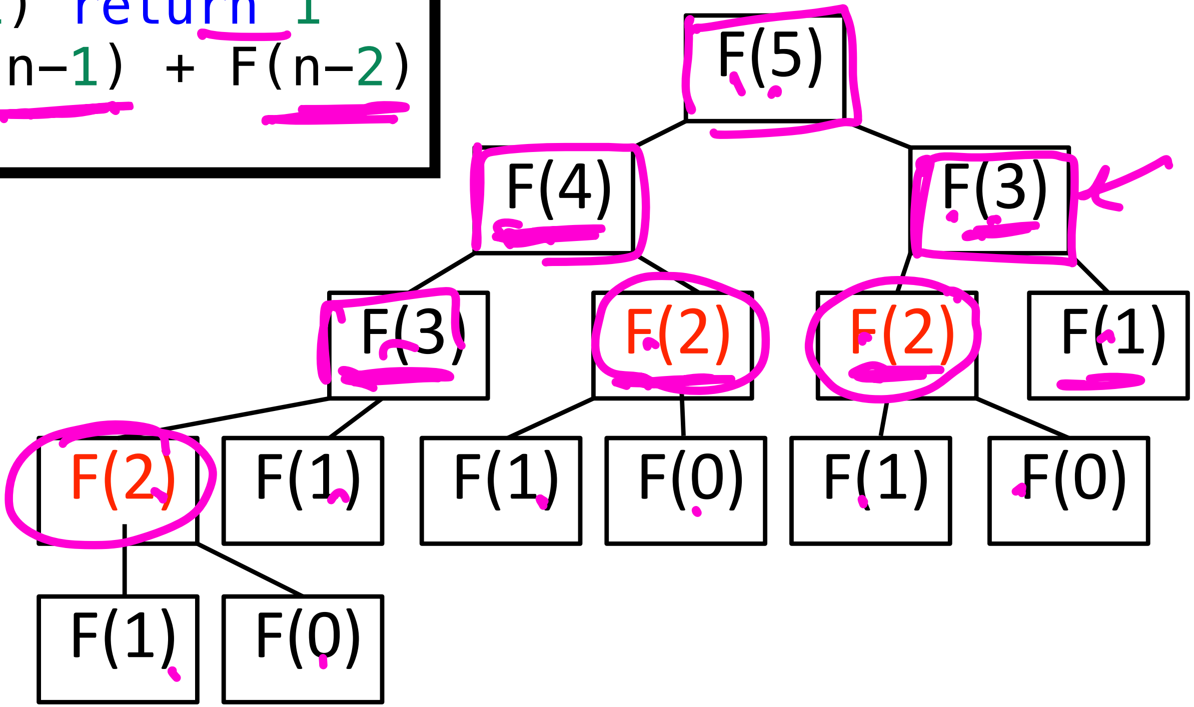
Practical. .

modeling - empirical analysis

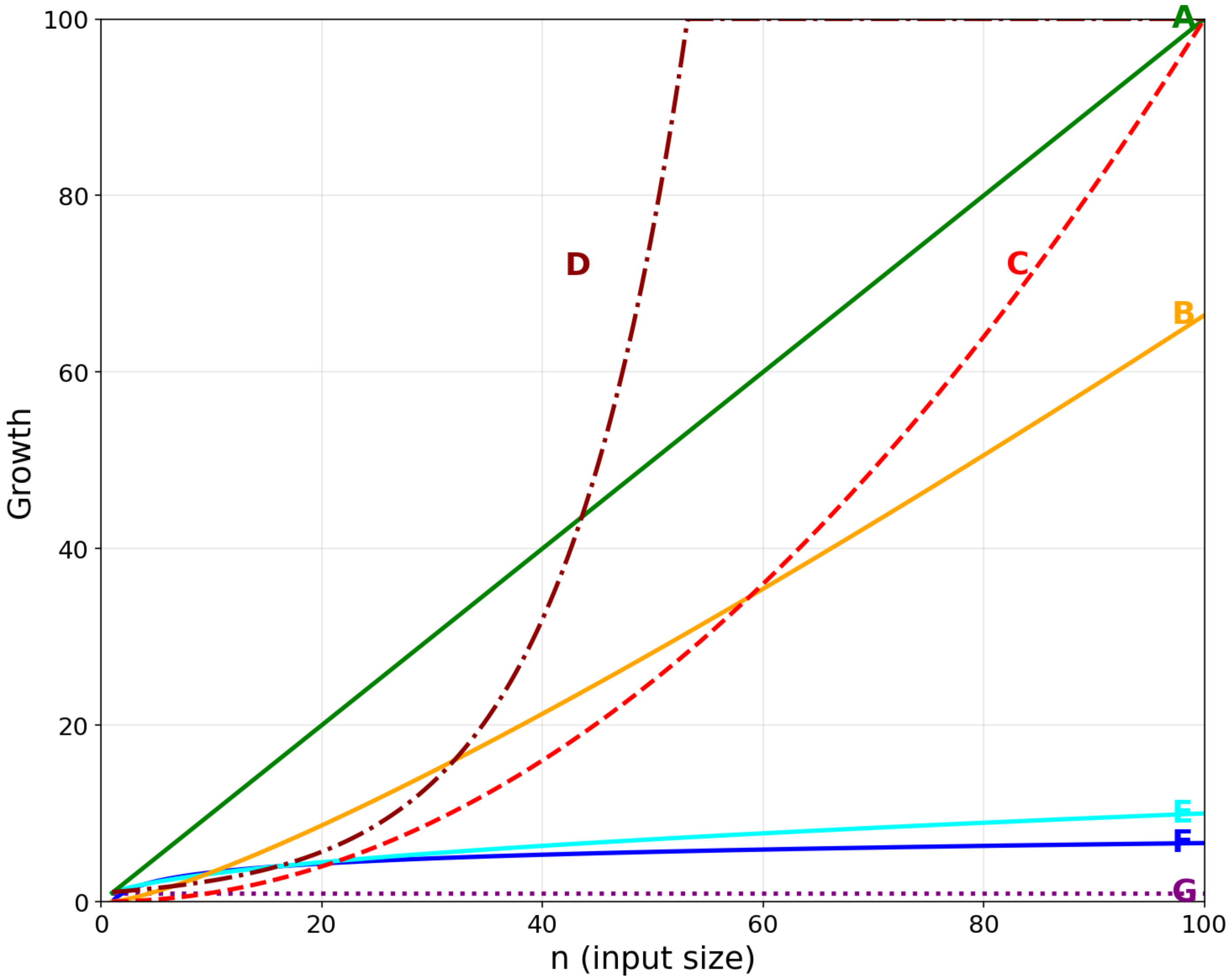
Why So Slow?

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

Too many recursive calls.



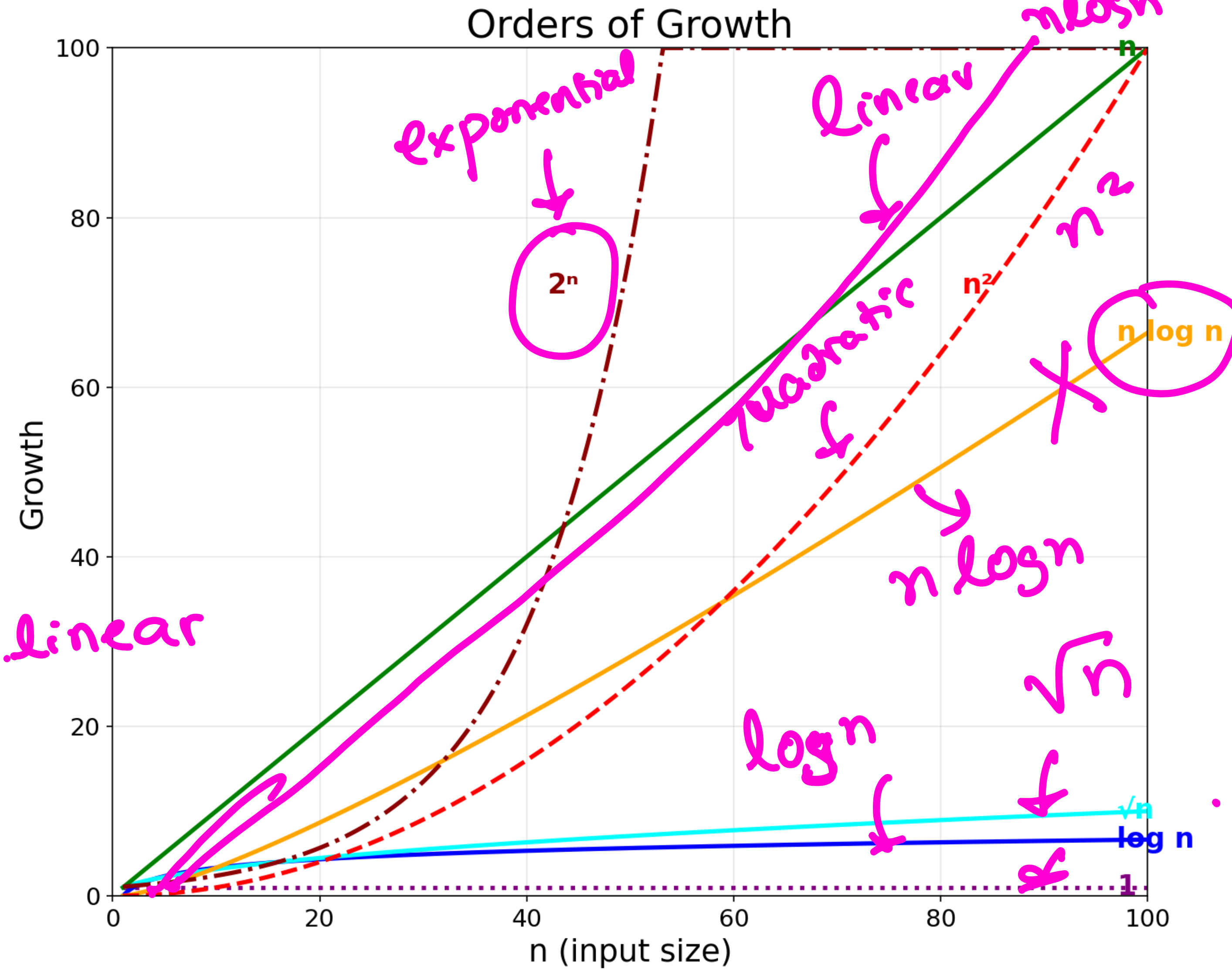
Which curve represents how the recursive fibonacci function grows?



- 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

$f(n) = n$
$g(n) = 3n+100$
$5n$
$200n$
$\rightarrow$ linear



ORDERS OF GROWTH ACTIVITY

1. Rank these functions from SMALLEST to LARGEST growth order

100 n $50n$ $2n^2$ $n \log n$ 2^n

$100 < n = 50n < n\log n < 2n^2 < 2^n$

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

n & 50 n ← linear

3. The recursive Fibonacci has $O(2^n)$ order of growth.

4. The iterative Fibonacci has $O(n)$ order of growth.

Big-O: Notation to name the order of growth

Order of Growth	Big-O Notation
Constant	$O(1)$
Logarithmic	$O(\log_2 n)$
Linear	$O(n)$
Linearithmic	$O(n \log n)$
Quadratic	$O(n^2)$
Exponential	$O(2^n)$

$f(n) = O(n^2)$

- 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. 10000000 = O(1)
2. 3n = O(n)
3. 6n-2 = O(n)
4. 15n + 44 = O(n)
5. 50nlog(n) = O(nlogn)
6. n^2 = O(n^2)
7. n^2-6n+9 = O(n^2)
8. $3n^2$+4*log(n)+1000 = O(n^2)
9. 3^n + n^3 +log(3*n) = O(3^n)

exponential polynomial

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;
}
```

c_1

Loop runs $\frac{n}{2}$ times

c_2

c_3

c_1, c_2, c_3 are const. values

$T(n) = c_1 + c_2 \cdot \frac{n}{2} + c_3$,

$= O(n)$

A. $O(n^2)$

B. $O(n)$

C. $O(n/2)$

D. $O(\log n)$

E. None of the above

linear

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]        O(1)
    A[0] = A[1] = 1                O(1)

    for i = 2 : n                  (n-1) .   O(1)
     A[i] = A[i-1] + A[i-2]

    return A[n]                    O(1)
}
```

Total no. of times loop runs

work done in every iteration

$T(n) = O(1) + O(1) + \underline{(n-1)\,O(1)} + O(1)$

$= O(n)$

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

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

$T(n) = c\cdot(\text{number of function calls})$, for some c is a constant

Big O allows upperbounding

upper bound

$\le c\cdot(1+2+\cdots\cdots 2^n)$

$= c\cdot(2^{n+1}-1)$

$= c\cdot 2\cdot 2^n - c$

$= O(2^n)$

$1+2 = 2^2-1$

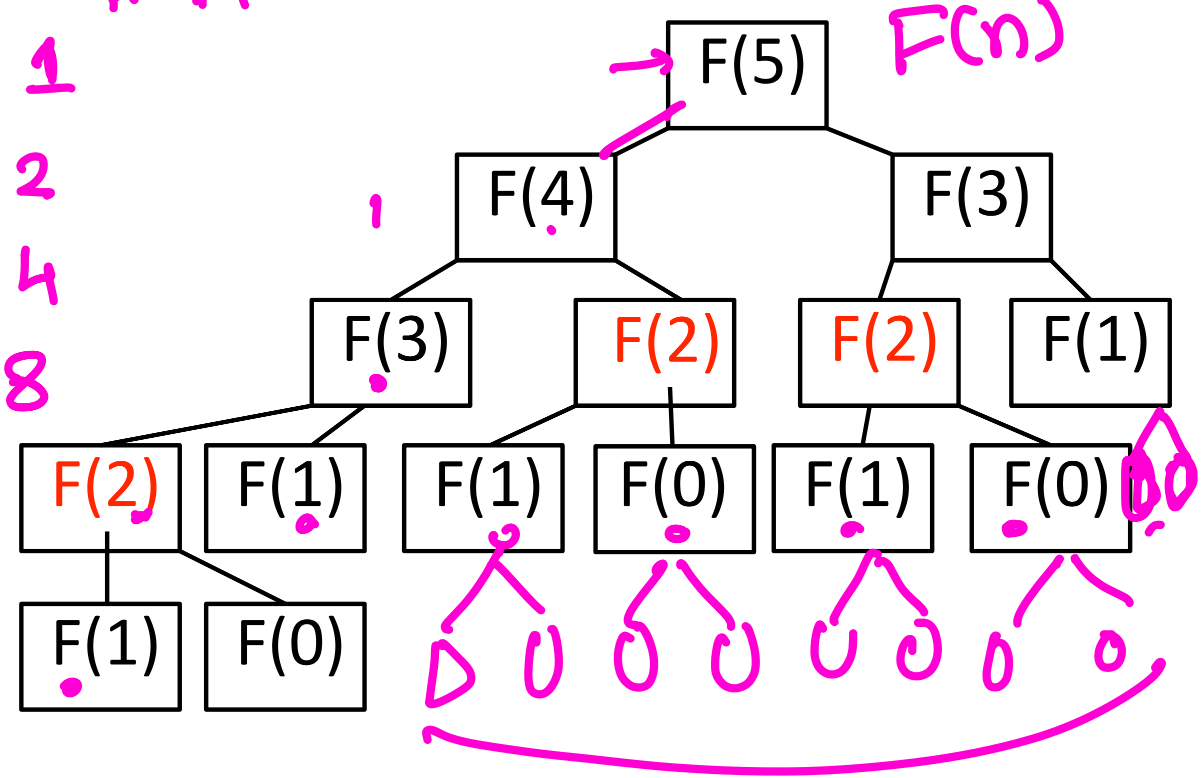
$\boxed{1+2+4} = 2^3-1$

$1+2+4+8 = 2^4-1$

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

Total number of function calls $\leq 1 + 2 + 4 + \cdots 2^n$

level	# of function calls
0	1
1	2
2	4
3	8
⋮	
k	2^k



We added these function calls to get a full tree, makes analysis simpler.

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 auxilliary space only

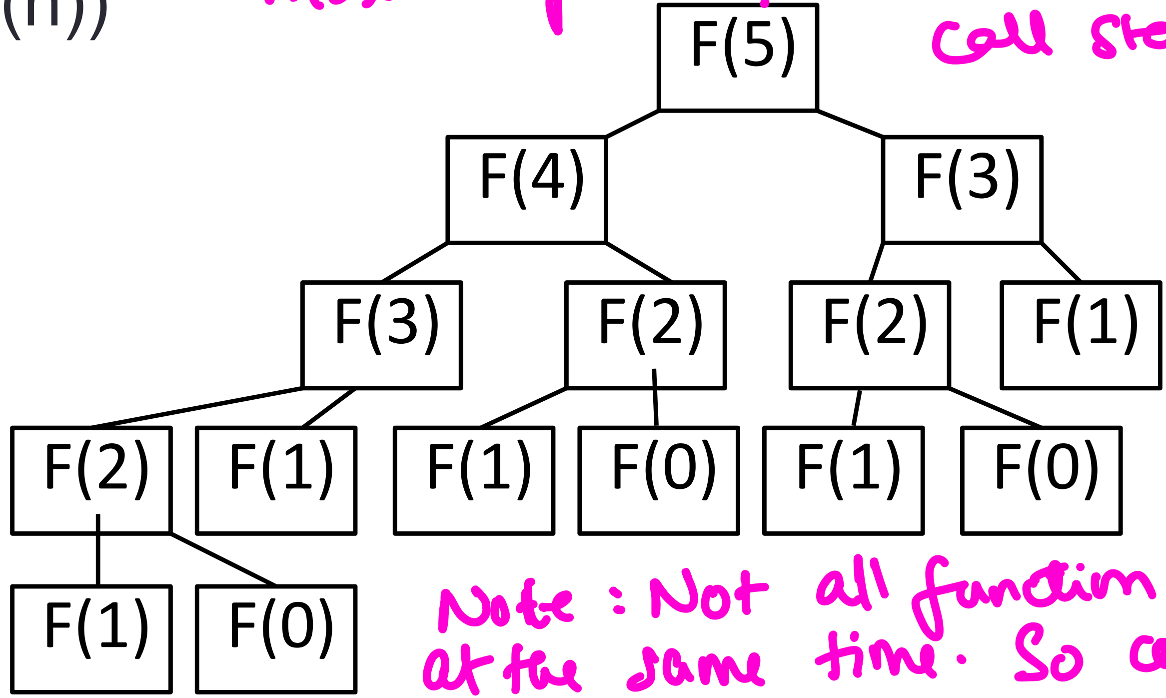
```
F(int n){
  if(n <= 1) return 1
  return F(n-1) + F(n-2)
}
```

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. $O(2^n)$

The space usage is dominated by the max. depth of recursion on the function call stack



Note: Not all function calls are made at the same time. So complexity is not $O(2^n)$

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)
}
```

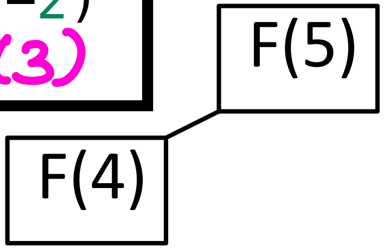
F(5)

S(n) relates to maximum depth of the recursion

5

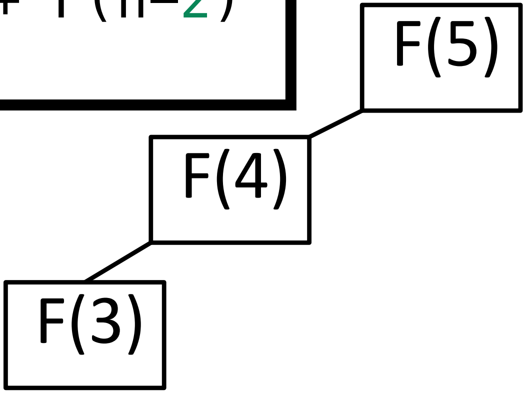
```
F(int n){
  if(n <= 1) return 1
  return F(n-1) + F(n-2)
}
```

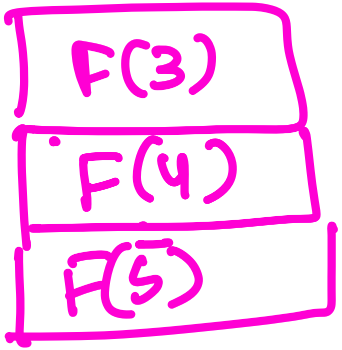
F(4) + F(3)



S(n) relates to maximum depth of the recursion

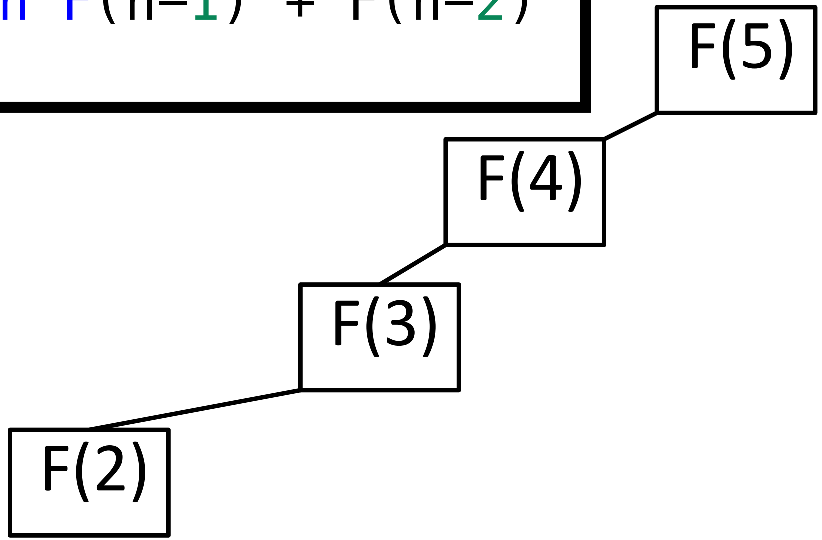
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```





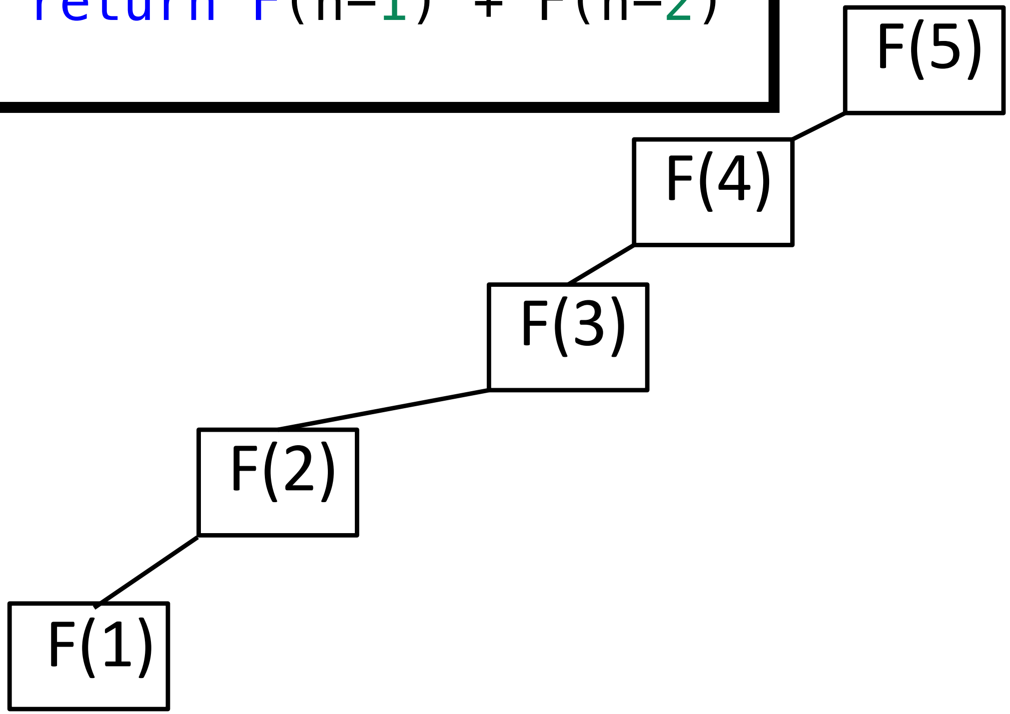
S(n) relates to maximum depth of the recursion

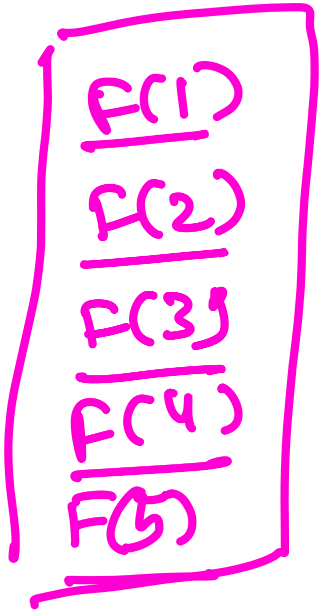
```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



S(n) relates to maximum depth of the recursion

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

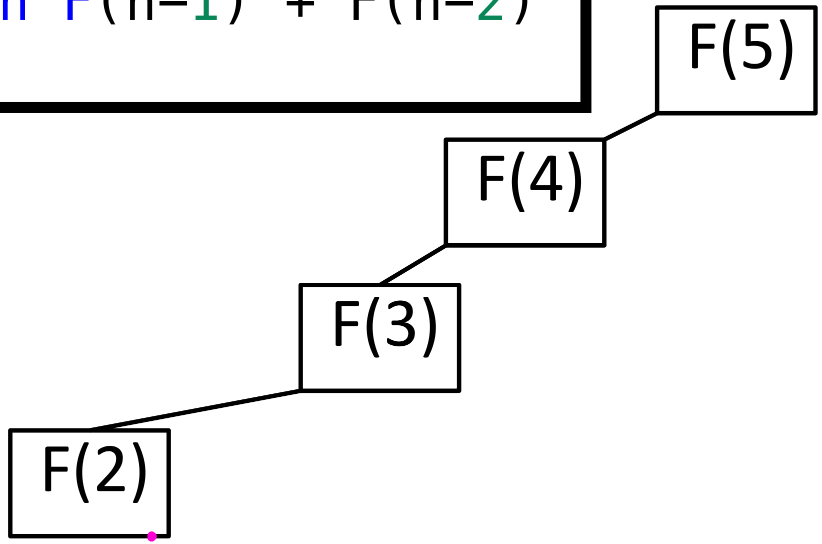




Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

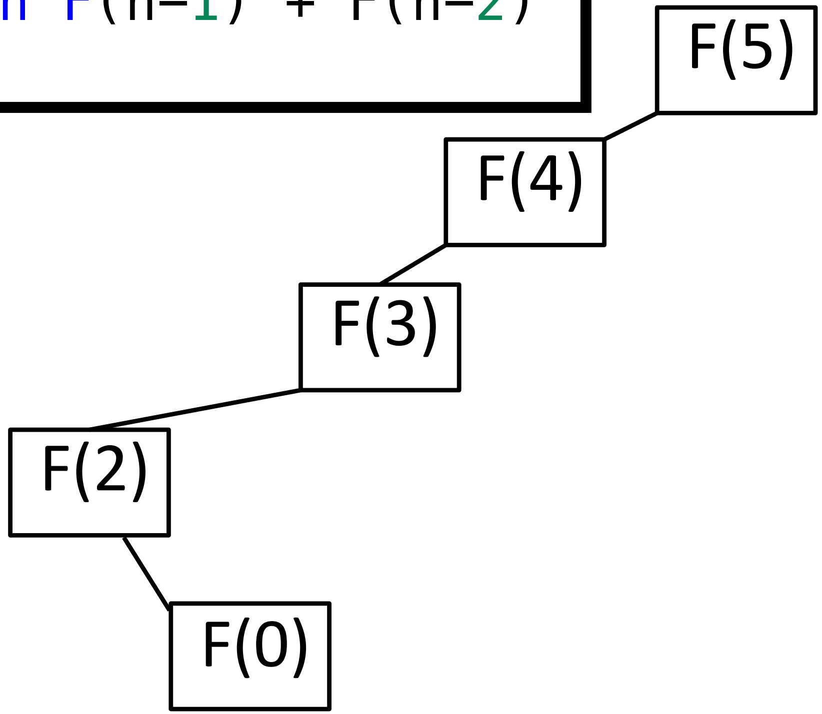
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

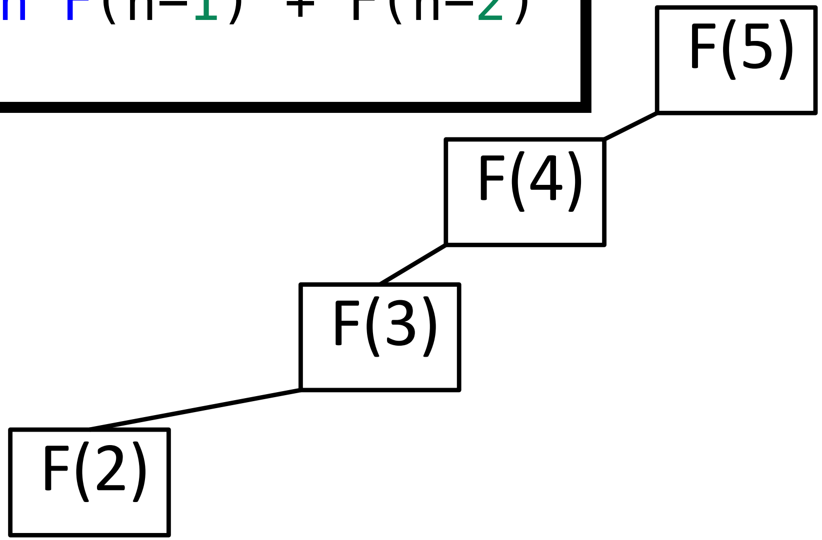
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

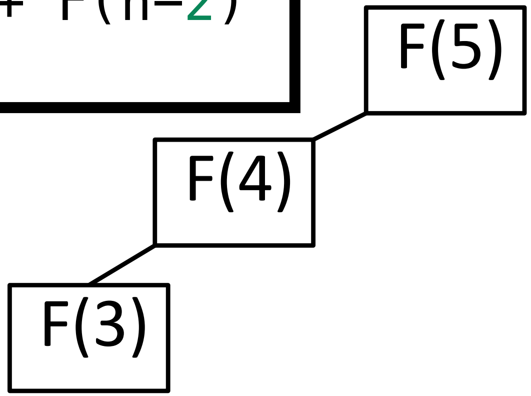
```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

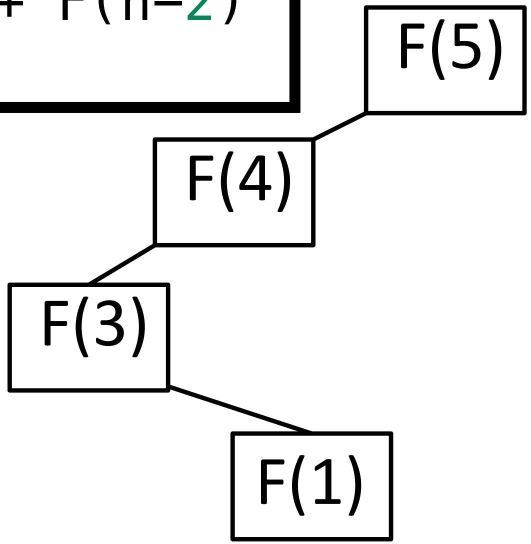
```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

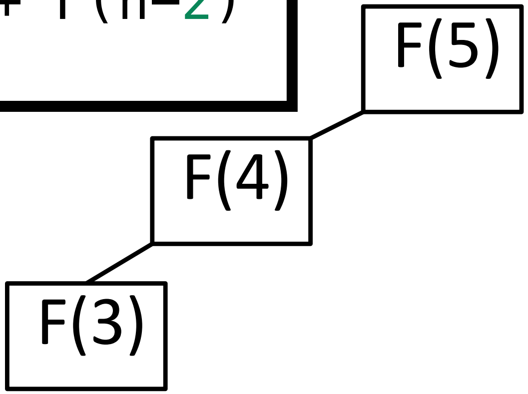
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

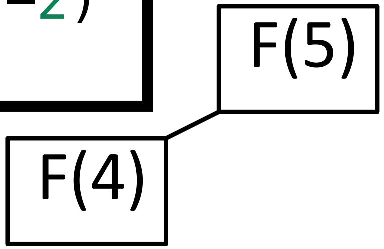
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

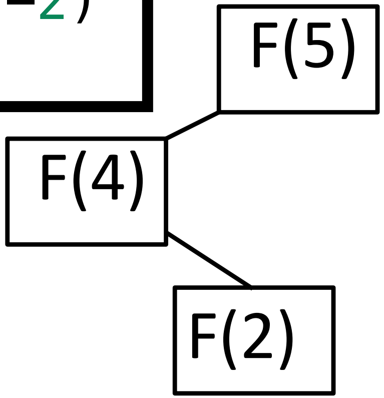
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

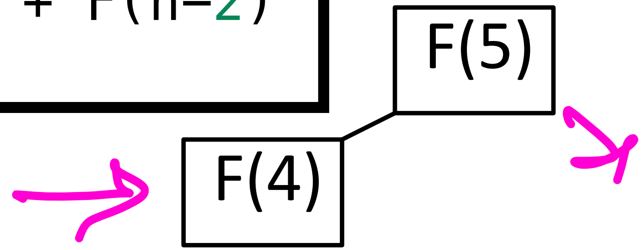
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

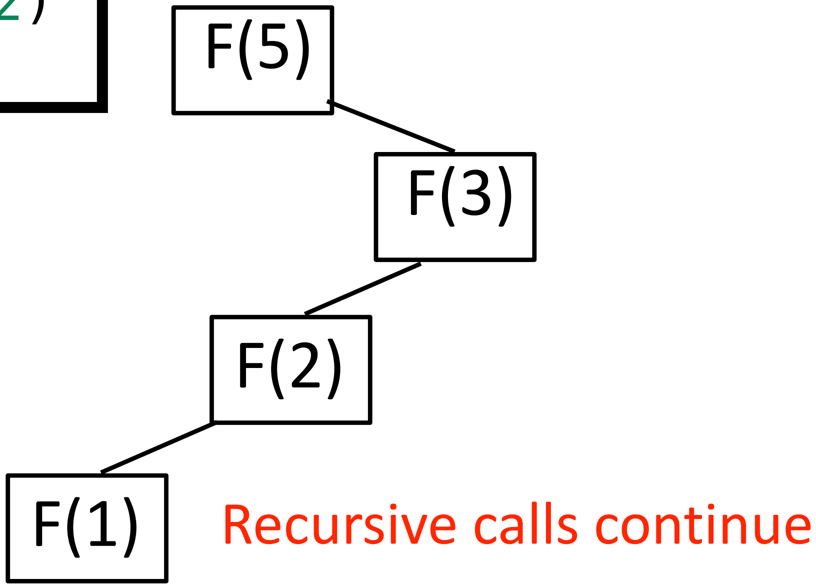
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



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 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