Recursion

By

Dr. Tisha

What is Recursion?

- A recursive algorithm is one that defines a solution to a problem in terms of itself.
- A recursive method is one that calls itself
- Where/ why to use recursion?

Where to use recursion?

Recursion is useful when a problem can be broken down into smaller subproblems of the same type.
Common scenarios where recursion is beneficial in Java include:

1. **Mathematical Computations**
   - Factorial (n! = n * (n-1)!)
   - Fibonacci series
   - Greatest Common Divisor (GCD)
2. **Data Structure Operations**
   - **Tree Traversals** (Binary Tree, BST)
   - **Graph Traversal** (DFS)
   - **Linked List Operations** (reversing, searching)
3. **Divide and Conquer Algorithms**
   - Merge Sort
   - Quick Sort
   - Binary Search
4. **Backtracking Problems**
   - Solving a Maze
   - N-Queens Problem
   - Sudoku Solver
5. **Dynamic Programming**
   - Computing Fibonacci with Memoization
   - Knapsack Problem
   - Longest Common Subsequence

Why to use recursion?

1. **Simplifies Code**
   - It often leads to cleaner and more intuitive code, especially for problems that naturally fit recursive decomposition (e.g., tree traversal).
2. **Reduces Code Length**
   - Recursive implementations tend to be shorter compared to iterative versions.
3. **Better for Hierarchical Data**
   - Many problems like trees and graphs are naturally recursive in structure, making recursion a good fit.
4. **Solves Problems That Are Hard to Implement Iteratively**
   - Some problems (like backtracking) are complex to implement using loops but are straightforward with recursion.

Where no to use recursion

While recursion is powerful, it has some drawbacks:

- **High Memory Usage**: Each recursive call adds a new frame to the stack, leading to potential StackOverflowError for deep recursion.
- **Slower Execution**: Compared to loops, recursion has an overhead due to function calls.
- **Difficult Debugging**: Recursive functions can be tricky to debug due to multiple function calls in the stack.

Factorial Problem

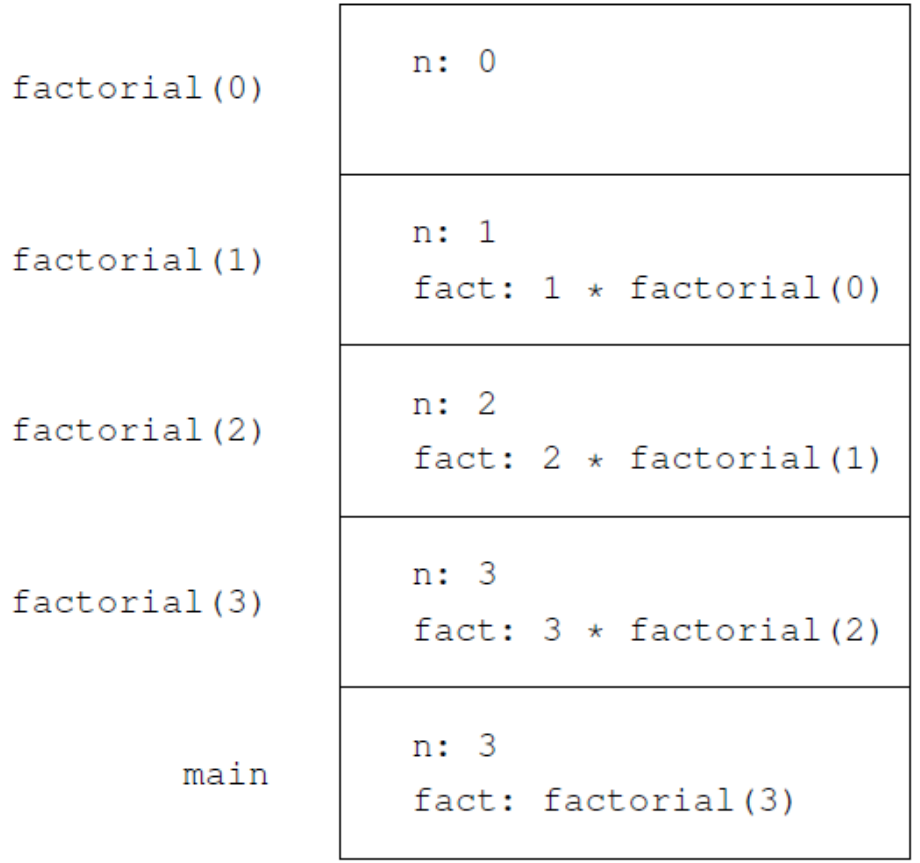
- **Write a code for factorial without recursion** **n * factorial(n-1);**

- **0! = 1**
- **1!= 1**
- **2!= 2*1**
- **5! = 5*4*3*2*1= 5*4*3!**
- **n!= n*(n-1)*(n-2)*…..1**
- **= n*(n-1)!**
- **F(n) = n*F(n-1)**

$$factorial(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times factorial(n-1), & \text{if } n > 0 \end{cases}$$

How recursion function executes

```java
/**
 * Executing a recursive method
 */
public class RecursiveExecution {

  /**
   * Calculate n! recursively
   */
  public static int factorial(int n) {
    // Base case
    if (n == 0) {
      return 1;
    }

    // Recursive case
    int f = n * factorial(n - 1);
    return f;
  }

  public static void main(String[] args) {
    int n = 3;
    int fact = factorial(n);
    System.out.println(fact);
  }
}
```



Memory Stack

Space Complexity

- The auxiliary space required to run a recursive function depends on the maximum depth of the memory stack used during execution.
- When a recursive function is called, each function call is added to the memory stack. Once the base case is reached, the function begins returning, and the calls are removed from the stack.
- The space complexity of the recursive function is determined by measuring the largest size of the stack during execution.
- In advanced coding, the constant space used by variable definitions and fixed-size arrays is typically not included when measuring space complexity. Instead, we focus on **auxiliary space**, which accounts for the additional memory required during execution, such as recursive stack frames and dynamically allocated structures. Since constant space remains unchanged regardless of input size, it is often ignored in asymptotic analysis, whereas recursive calls and dynamically expanding data structures significantly impact the overall space complexity.

ClassPractice 1(Write the recursive function and find out the space complexity)

- Write the power formula below as recursive function.

- $F(a^n) = a * F(a^{n-1})$
- Say, a = 5, n =3
- 5^3 = 5*5*5 = 5*5^2

- Write the recursive function for this formula

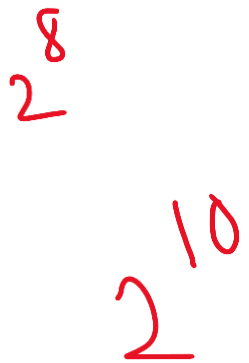
- Return a*power(a,n-1)

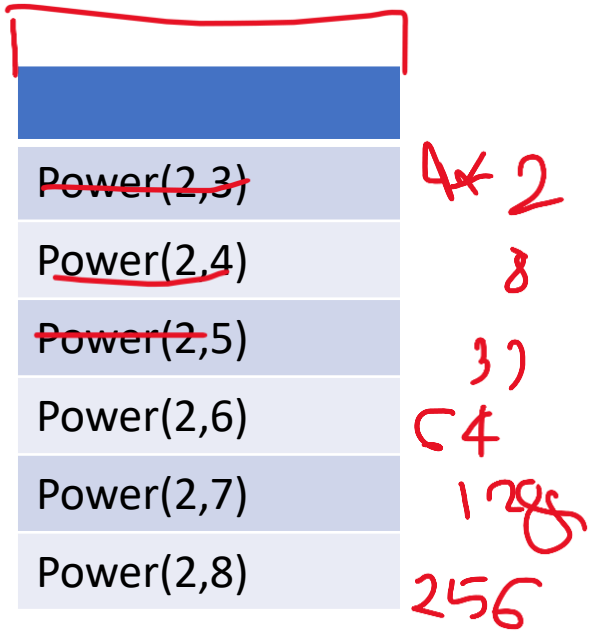
Formal mathematical formula definition

$$power(a,b) = \begin{cases} 1, & \text{if } b = 0 \\ a \times power(a, b-1), & \text{if } b > 0 \\ \frac{1}{power(a,-b)}, & \text{if } b < 0 \end{cases}$$

Space complexity

else return a*power(a,n-1);





O(n)

Class Practice 2

Like the builders of old, I enjoy constructing stone ziggurats in my back yard. A ziggurat of n levels starts with an $n \times n$ square of stones, then a layer of $(n-1) \times (n-1)$ stones, and so forth, until I finally place a single stone on top. Let $s(n)$ represent the number of stones in an n-layer ziggurat.

$$s(n) = \begin{cases} 1 & n = 1 \\ n^2 + s(n-1) & n > 1 \end{cases}$$

That is, a single layer ziggurat is just one stone, and an n-layer one has a base of n^2 stones plus the stones for the $n-1$ higher layers. Convert the definition of $s(n)$ to a `static` Java method called `stones`. The method will take a single `int n` as input and return an `int`.

Fibonacci number

- 0 1 1 2 3 5 8 13….

- https://www.youtube.com/watch?v=v6PTrc0z4w4
- https://www.youtube.com/watch?v=SjSHVDfXHQ4

Fibonacci series with recursion

- Consider the Fibonacci sequence, where each number is the sum of the two previous numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, . .
- If $F(n)$ is the n th Fibonacci number, then $F(n) = F(n-1) + F(n-2)$

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F(n-2), & \text{if } n \geq 2 \end{cases}$$

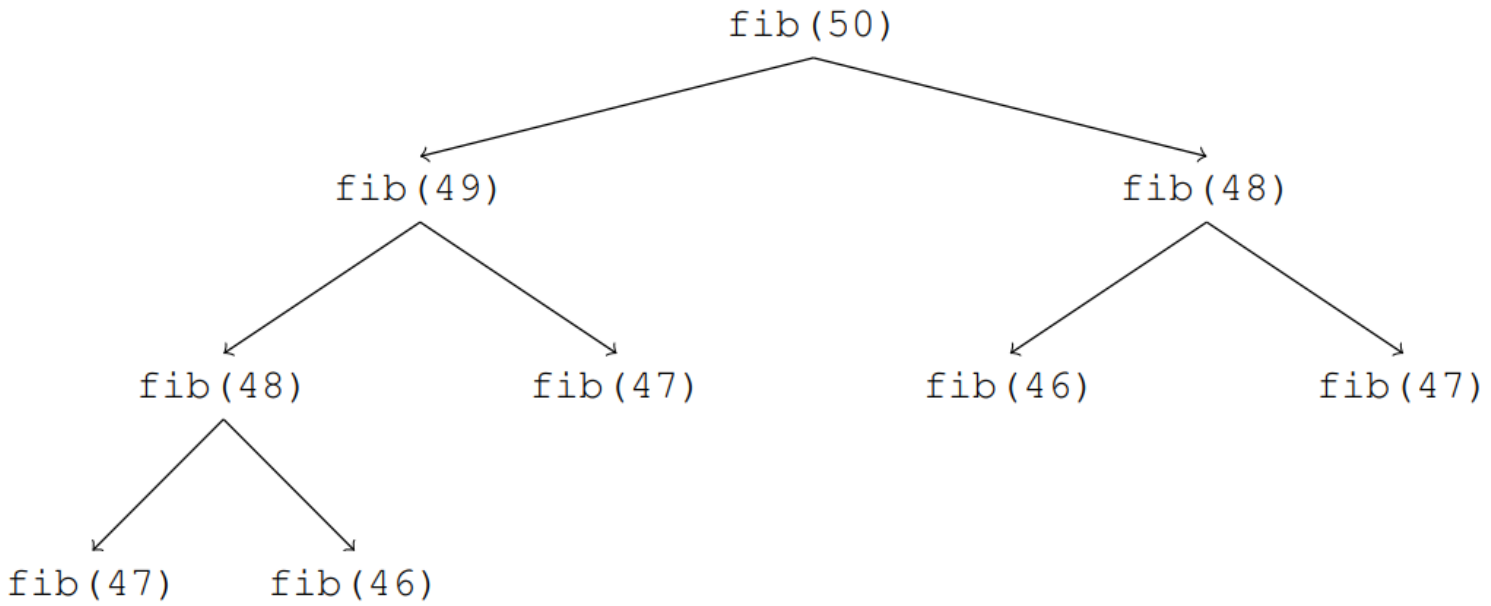
```
Fib(n){


Return Fib(n-1)+Fib(n-2);
}
```

Fibonacci code (make it work)

```java
/**
 * Recursive Fibonacci
 *
 * @param  n  number of the sequence
 * @return  F_n
 */
public static int fib(int n) {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}
```

What is the runtime of the Fibonacci code with recursion?

- This method seems perfectly fine when n is small, but its performance rapidly degrades for values in the range 50-100. In fact, although it seems simple, this implementation is terrible: its running time grows exponentially with the size of n. To understand the issue, consider just the top of the tree of recursive calls.



Practice Problem

- Finding GCD(Greatest Common Divisor) of two numbers using recursion with Euclidean algorithm.

- The Euclidean Algorithm for finding GCD(A,B) is as follows:
- If A = 0 then GCD(A,B)=B, since the GCD(0,B)=B, and we can stop.
- If B = 0 then GCD(A,B)=A, since the GCD(A,0)=A, and we can stop.
- Write A in quotient remainder form (A = B·Q + R)
- And R = A%B
- Find GCD(B,R) using the Euclidean Algorithm since GCD(A,B) = GCD(B,R)
- Read https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm

Finding GCD

```
public class GCD {
    public static int gcd(int a, int b) {
        // Base case: GCD(a, 0) = a
        if (b == 0)
            return a;

        // Recursive case: GCD(a, b) = GCD(b, a % b)
        return gcd(b, a % b);
    }

    public static void main(String[] args) {
        int a = 0;
        int b = 48;

        int result = gcd(a, b);
        System.out.println("GCD of " + a + " and " + b + " is: " + result);
    }
}
```

Practice at Home

- 1+2+3+……+n
- This sum can also be calculated recursively by adding n to the sum of integers up to $n-1$
- Base Case:
  - If n=1, return 1.
- Recursive Case: Otherwise,
  - return n+sum_of_numbers($n-1$)

Home Read

- https://www.geeksforgeeks.org/worst-average-and-best-case-analysis-of-algorithms/

Analyzing Recursive algorithm

- Recursive algorithms, pose a problem for the statement-counting approach to analysis: it's not easy to figure out how many times a recursive method can execute before it terminates.
- a general method for analyzing recursive algorithms.
  - **recurrence relation**: an equation that expresses the amount of recursive work the problem requires as a function of the problem size n.
  - Given a recurrence relation, we can then apply standard mathematical techniques to derive an equation that satisfies the relationship,
  - then convert that result to Big-O notation.

Time Complexity of factorial problem

```
fact (n){
    if(n==0)
        return 1;
    else
        return n*fact(n-1);
}
```

Recurrence relation: Recurrence relations are often used to model the cost of recursive functions.

T(n) = T(n-1) +3 (or C) where n>0
T(0) =1

If we want to convert T(n) in terms of T(0)

T(n) = T(n-2) +3+3 extra work
= T(n-3)+9
= T(n-4)+12
....
= T(n-k)+3k
n-k= 0 => k=n
T(n) = T(0) +3n => 1+3n

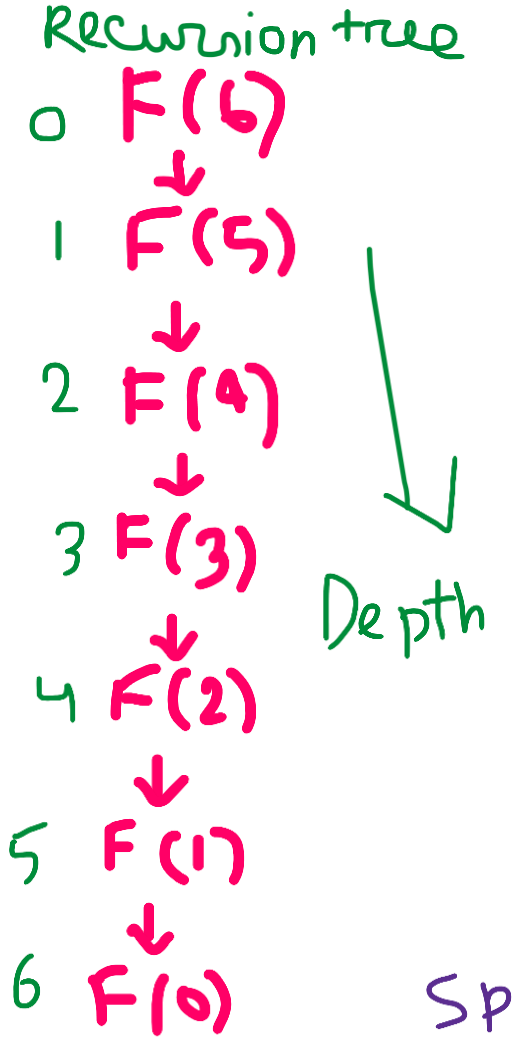
Time complexity will be O(n);

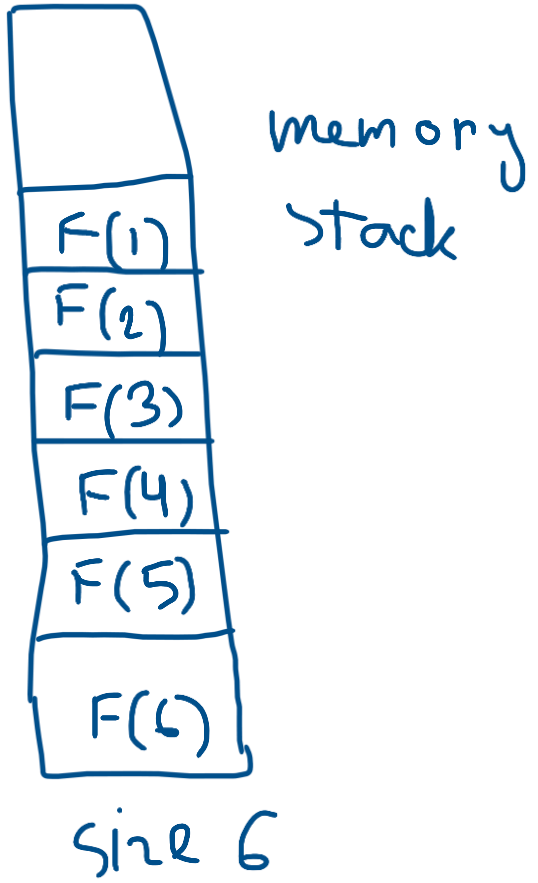
https://www.youtube.com/watch?v=ncpTxqK35PI

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/RecurrenceIntro.html#:~:text=A%20recurrence%20relation%20defines%20a,%3D(n%E2%88%921)!

Space Complexity of factorial problem

```
fact (n){
    if(n==0)
            return 1;
    else
            return n*fact(n-1);
}
```





Space $\propto n$

$O(n)$

Practice Problem

A recursive function code for power functions is as follows:

```
int power(int x, int n) {
        if (n == 0)
                return 1;
                        else return x * power(x, n-1);
}
```

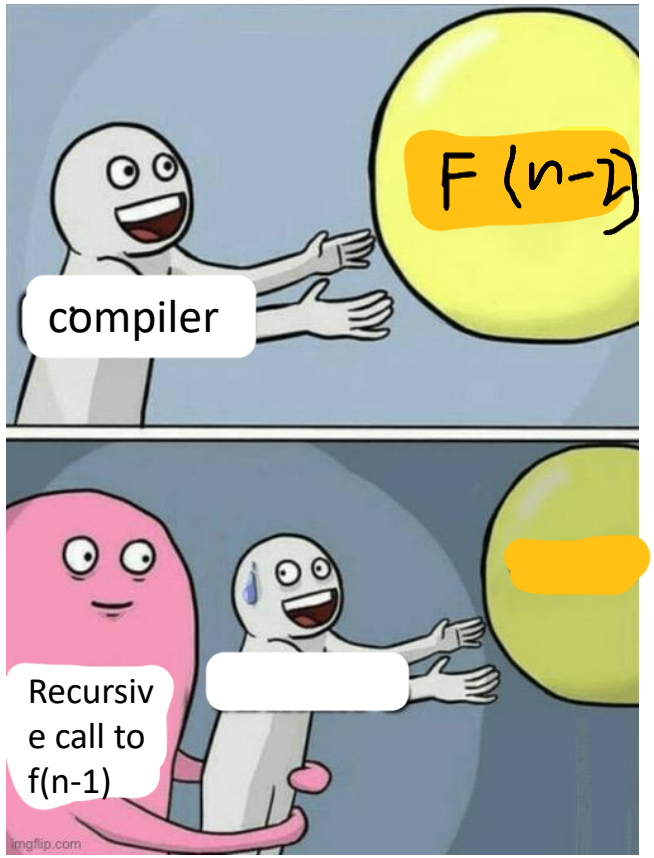
Given T(0)=1, What is the recurrence relation T(n)?

Let’s think about time complexity for Fibonacci series

Consider the Fibonacci sequence, where each number is the sum of the two previous numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, . .
If $F(n)$ is the n th Fibonacci number, then $F(n) = F(n-1) + F(n-2)$

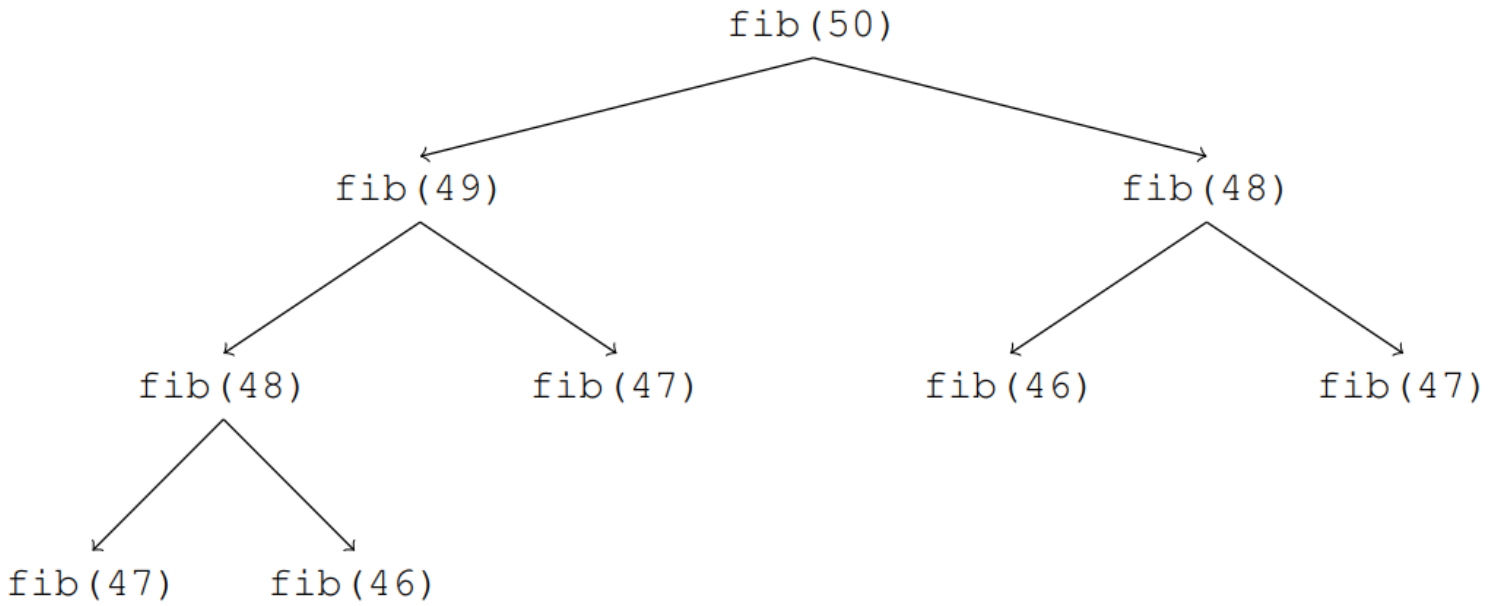
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
         return Fib(n-1)+Fib(n-2);
```



What is the runtime of the Fibonacci code with recursion?

- This method seems perfectly fine when n is small, but its performance rapidly degrades for values in the range 50-100. In fact, although it seems simple, this implementation is terrible: its running time grows exponentially with the size of n. To understand the issue, consider just the top of the tree of recursive calls.



Time complexity of Fibonacci recursion

0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recurrence relation:

T(n) = T(n-1) +T(n-2)+ C where n>1 and C=4

T(0) =T(1)=1

If we want to convert T(n) in terms of T(0)

For a lower bound we can say that T(n-1)~T(n-2)

T(n) = 2T(n-2) +C --- 1st run

= 2{2T(n-4)+C}+C --- 2nd run

= 4T(n-4)+3C

= 8T(n-6)+7C --- 3rd run

= 16T(n-8)+15C --- 4th run

….

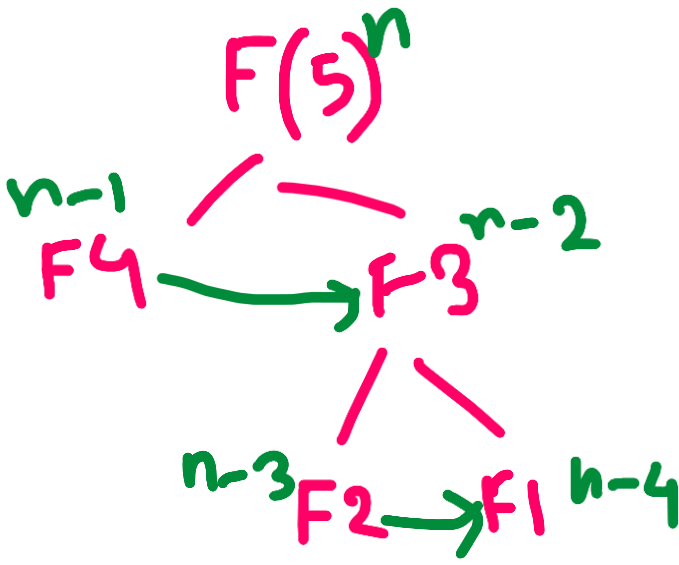
$= 2^k T(n-2k) + (2^k - 1)C$

Now, n-2k= 0 => k=n/2

So, $T(n) = 2^{\frac{n}{2}} T(n-n) + \left(2^{\frac{n}{2}} - 1\right) C$

$= 2^{\frac{n}{2}} T(0) + \left(2^{\frac{n}{2}} - 1\right) C$

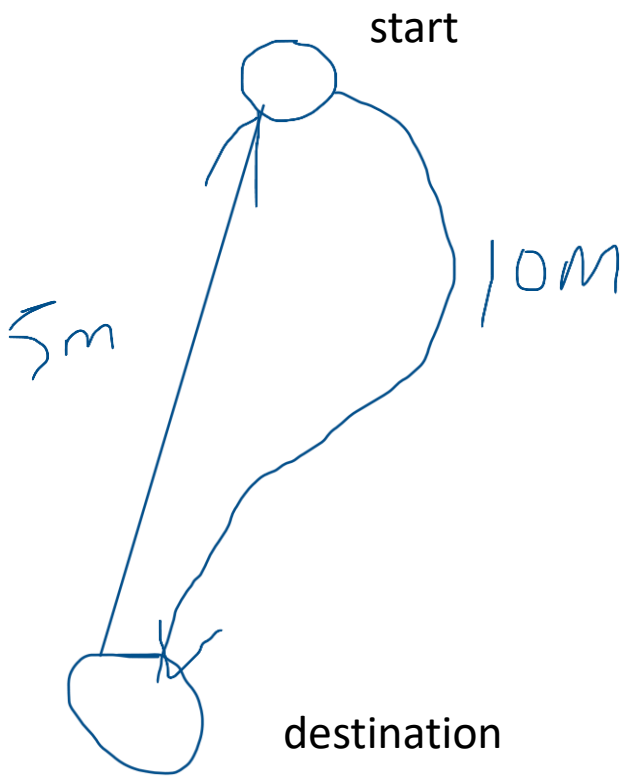
$= (1+C)\ 2^{\frac{n}{2}} - C$

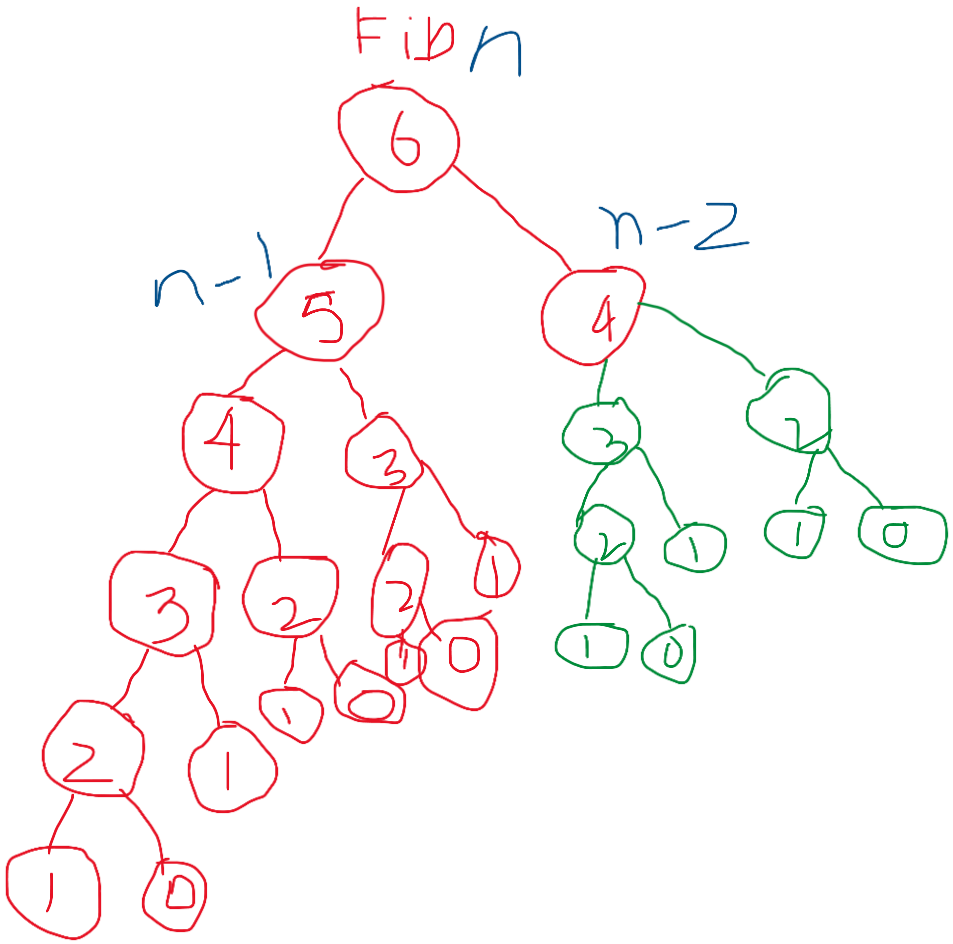


$\approx \Omega(2^{n/2})$

Lower bound

Think of a one-way road





Time complexity of Fibonacci recursion upper bound

0 1 1 2 3 5 …

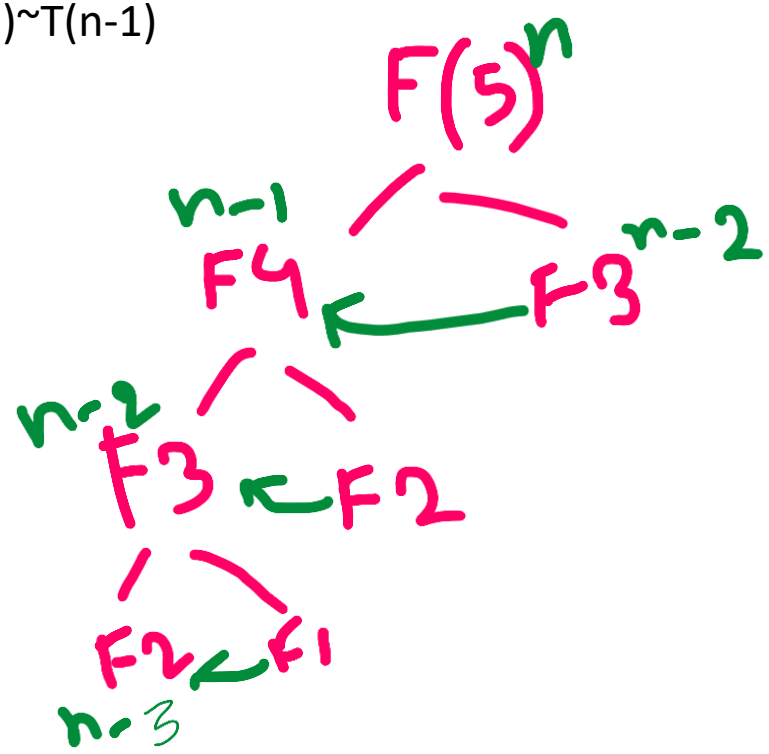
```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recurrence relation:
T(n) = T(n-1) +T(n-2)+ C where n>1 and C=4
T(0) =T(1)=1
If we want to convert T(n) in terms of T(0)

For a upper bound we can say that T(n-2)~T(n-1)

T(n) = 2T(n-1) +C --- 1st run
= 2{2T(n-2)+C}+C --- 2nd run
= 4T(n-2)+3C
= 8T(n-3)+7C --- 3rd run
= 16T(n-4)+15C --- 4th run
….
= $2^k T(n-k) + (2^k - 1)C$

Now, n-k= 0 => k=n
So, T(n) = $2^n T(n-n) + (2^n - 1)C$
= $2^n T(0) + (2^n - 1)C$
= (1+C) 2^n -C



$\approx O(2^n)$

Space complexity of Fibonacci recursive function

0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree

Memory stack

Space complexity of Fibonacci recursive function

0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree

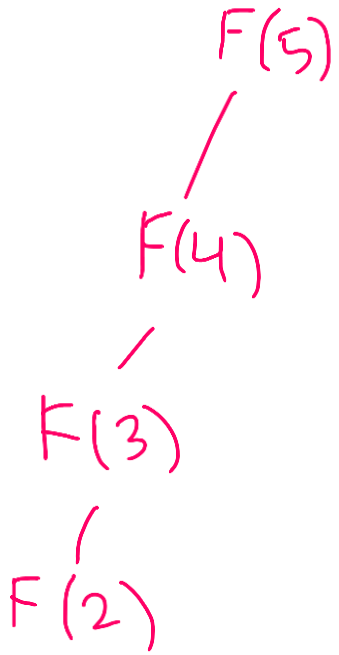
Memory stack

Space complexity of Fibonacci recursive function

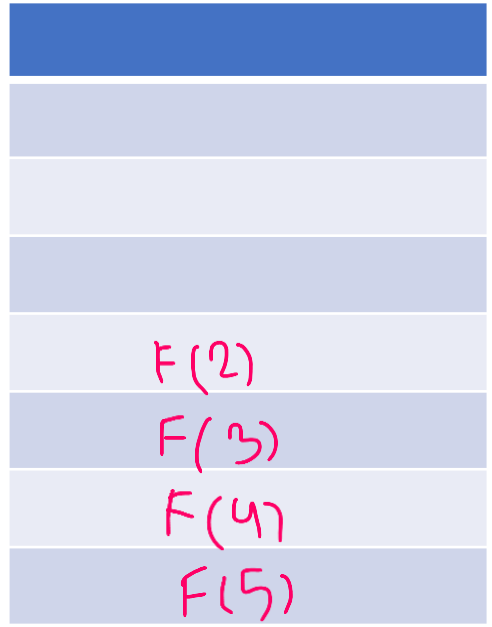
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

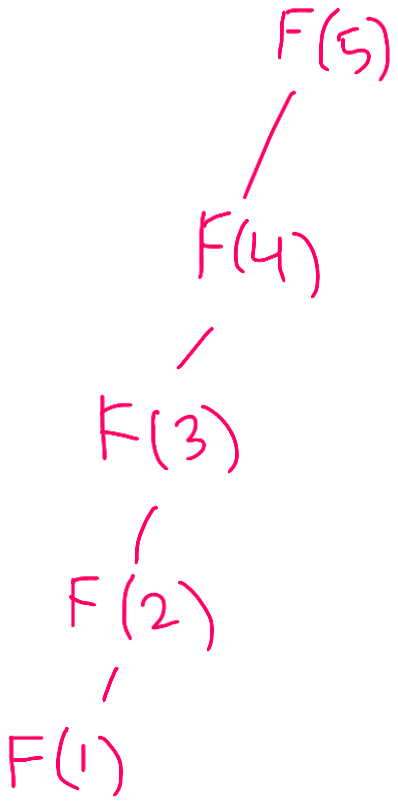


Space complexity of Fibonacci recursive function

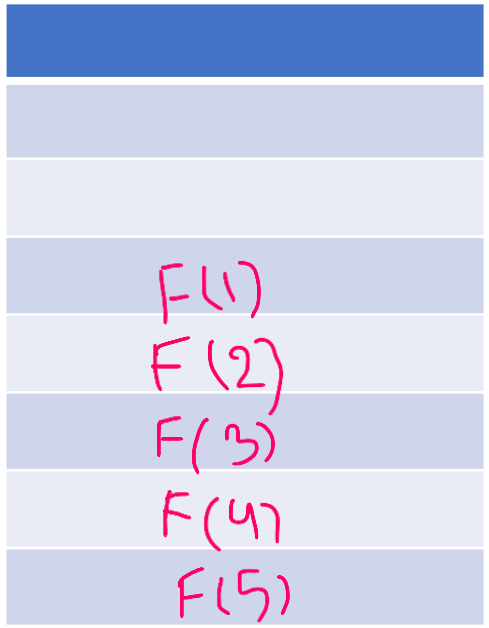
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

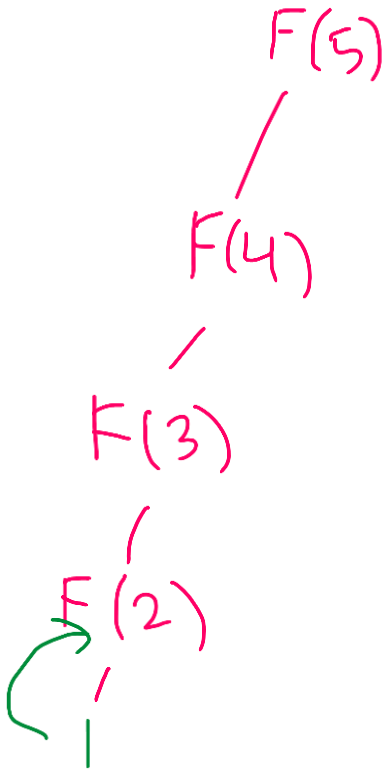


Space complexity of Fibonacci recursive function

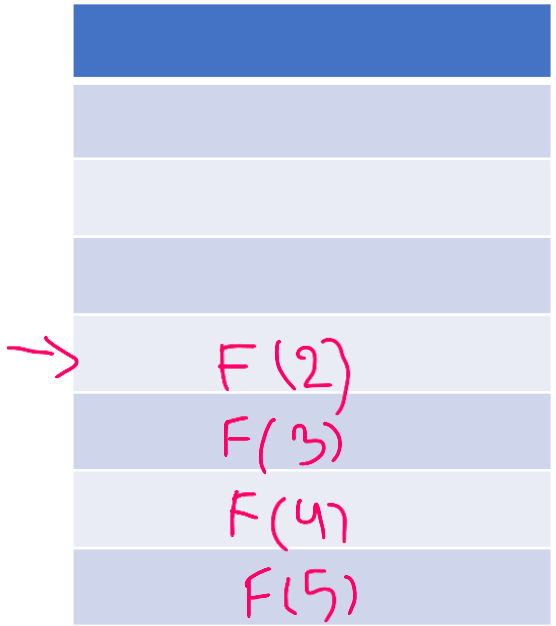
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

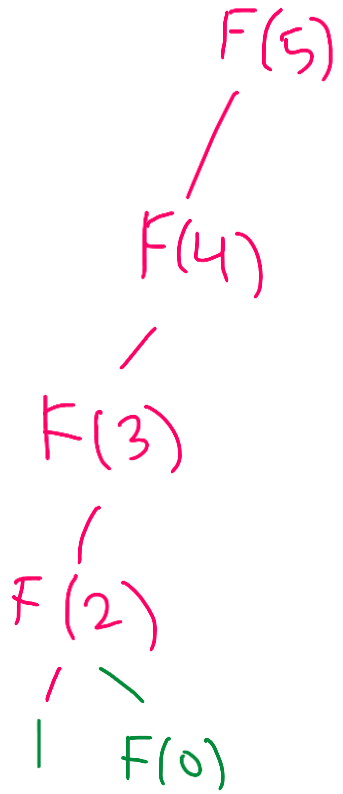


Space complexity of Fibonacci recursive function

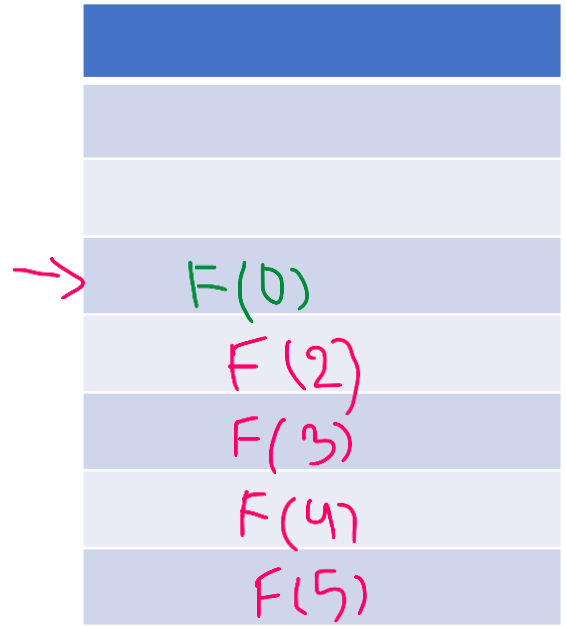
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

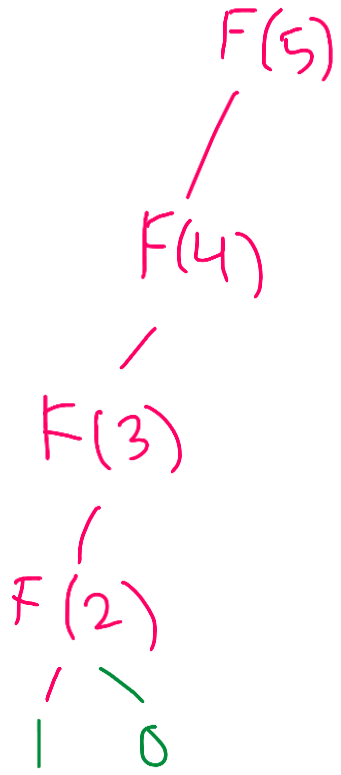


Space complexity of Fibonacci recursive function

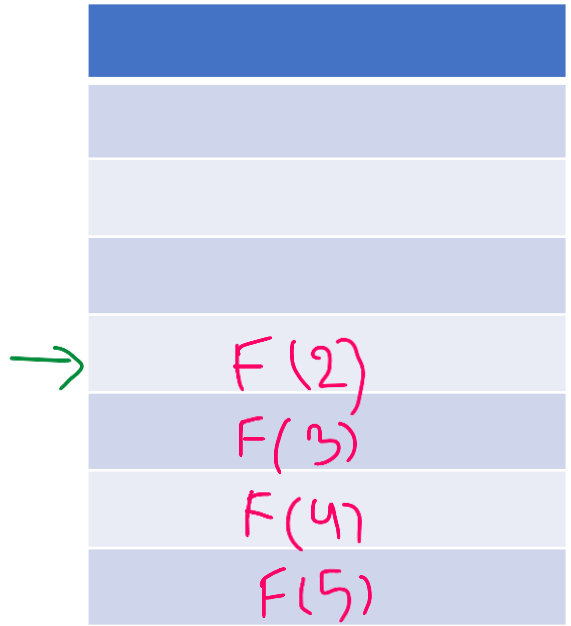
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

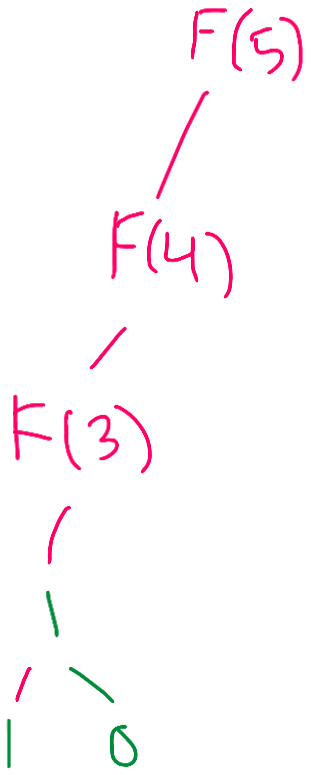


Space complexity of Fibonacci recursive function

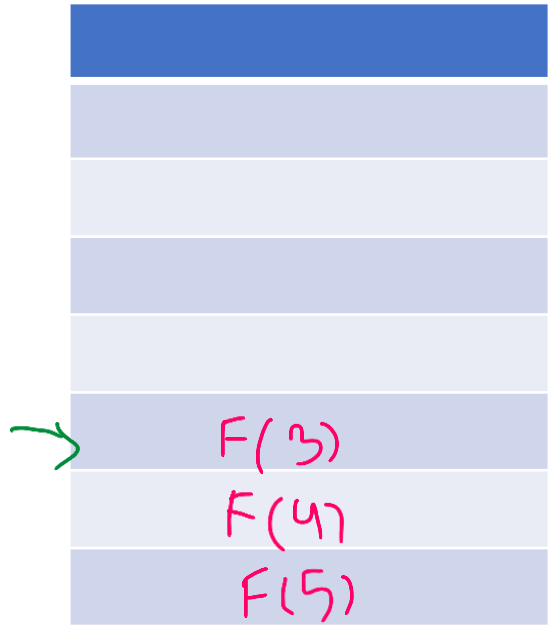
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

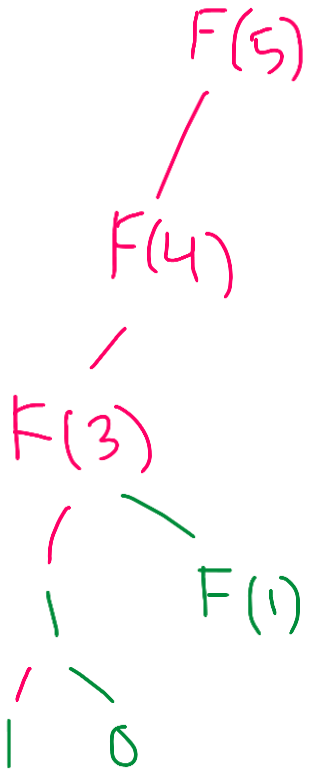


Space complexity of Fibonacci recursive function

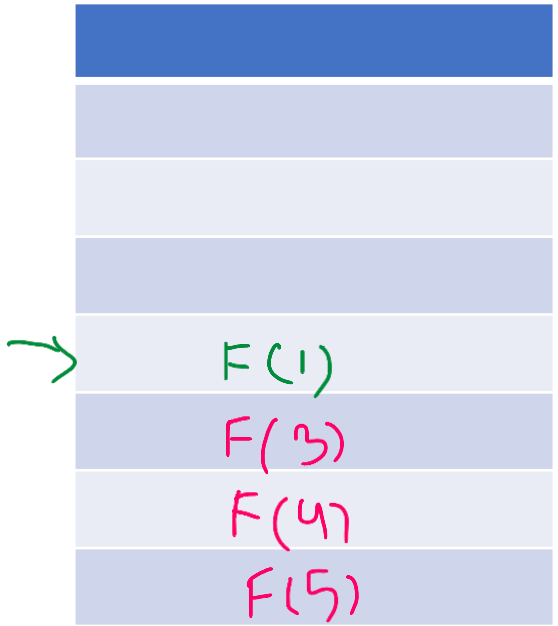
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

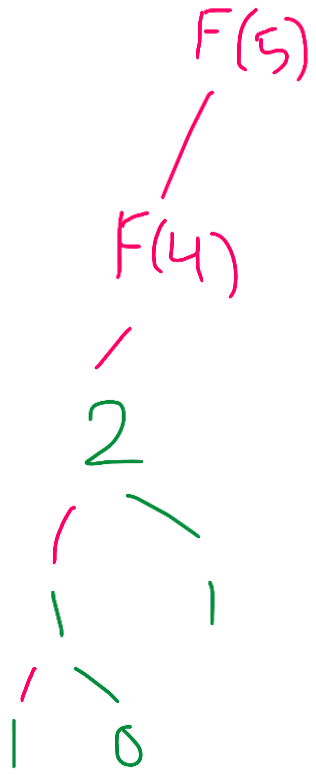


Space complexity of Fibonacci recursive function

0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



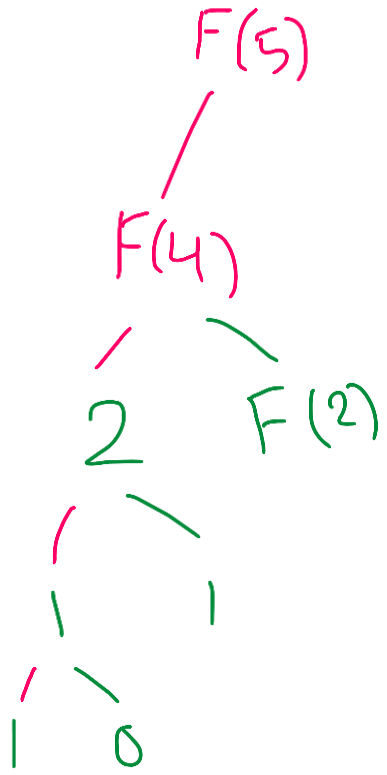
Memory stack

Space complexity of Fibonacci recursive function

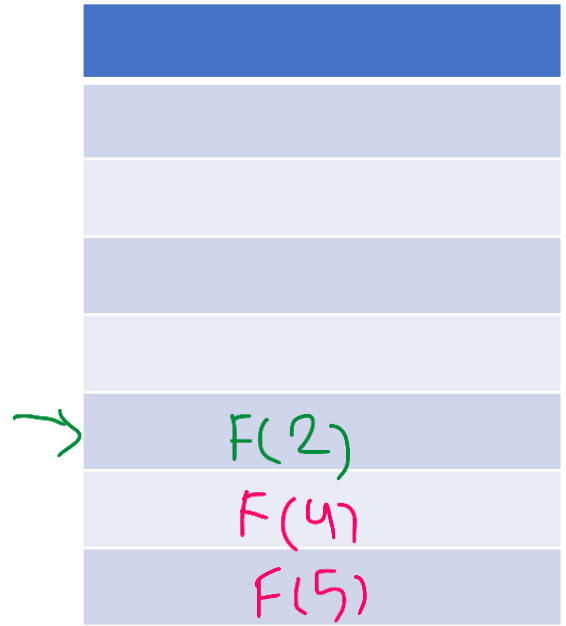
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

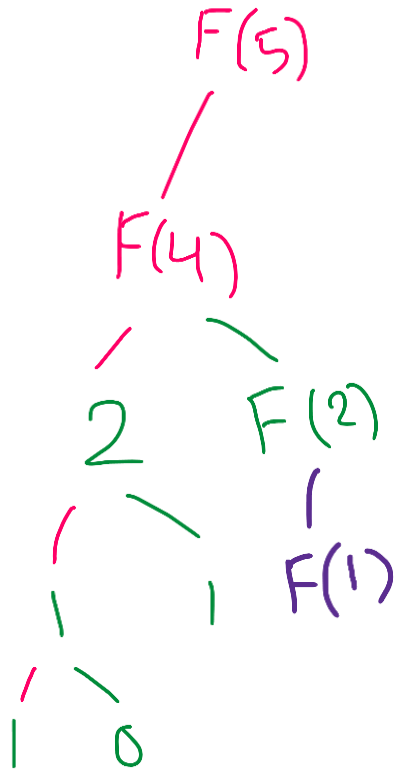


Space complexity of Fibonacci recursive function

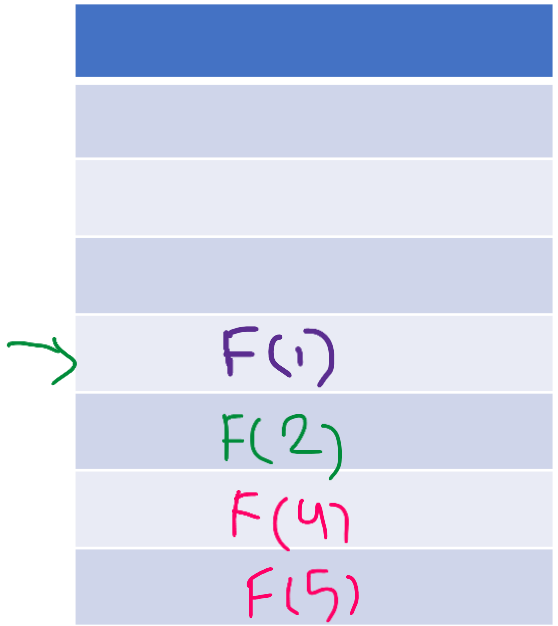
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

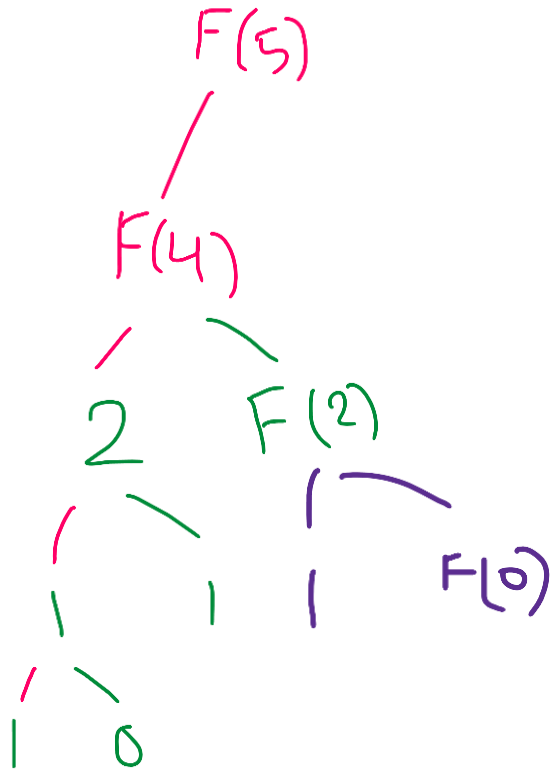


Space complexity of Fibonacci recursive function

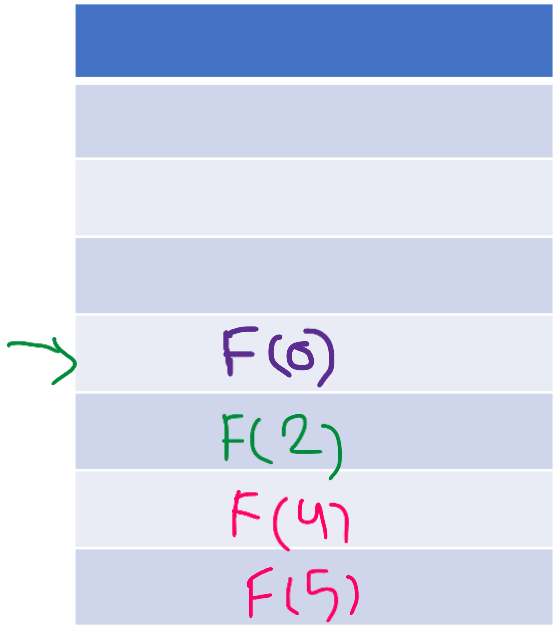
0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

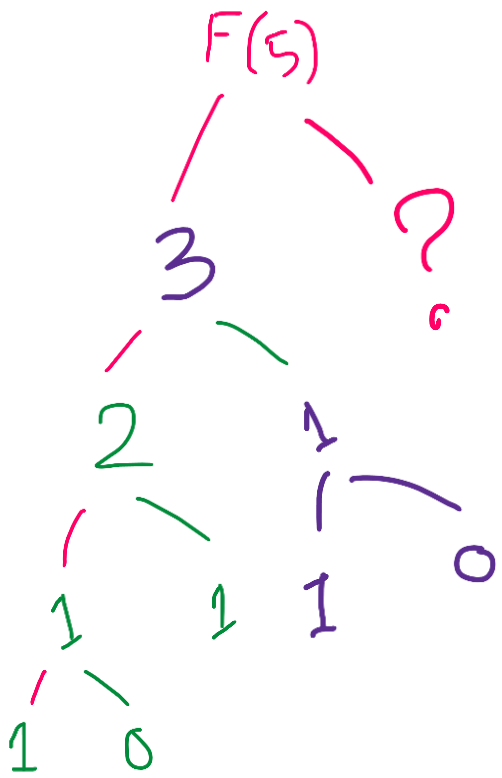


Space complexity of Fibonacci recursive function (Bonus: Show the detailed tree and the memory stack) Check on the Canvas for details.

0 1 1 2 3 5 …

```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Memory stack

What would be the space complexity of Fibonacci recursion of Fib(5)?

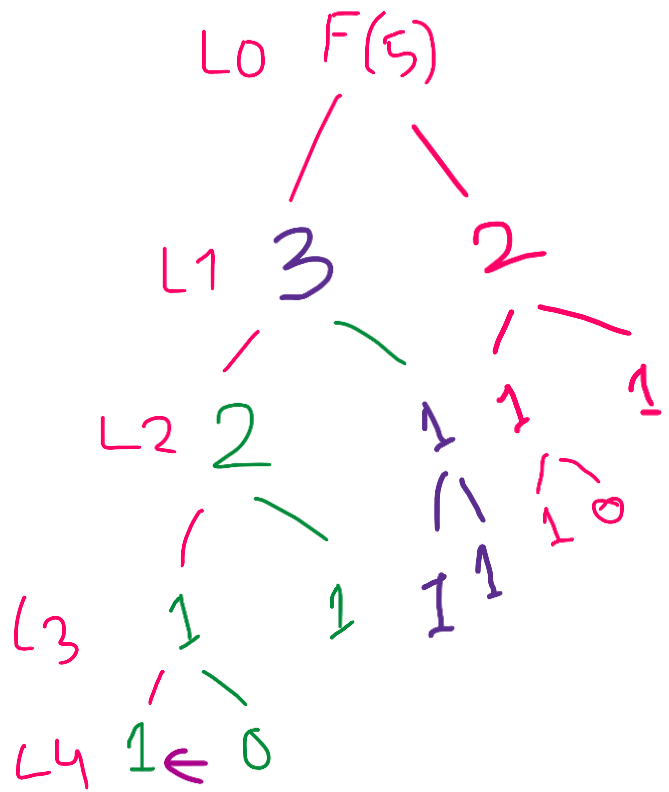
https://www.youtube.com/watch?v=dxyYP3BSdcQ

Space complexity of Fibonacci recursive function

0 1 1 2 3 5 …

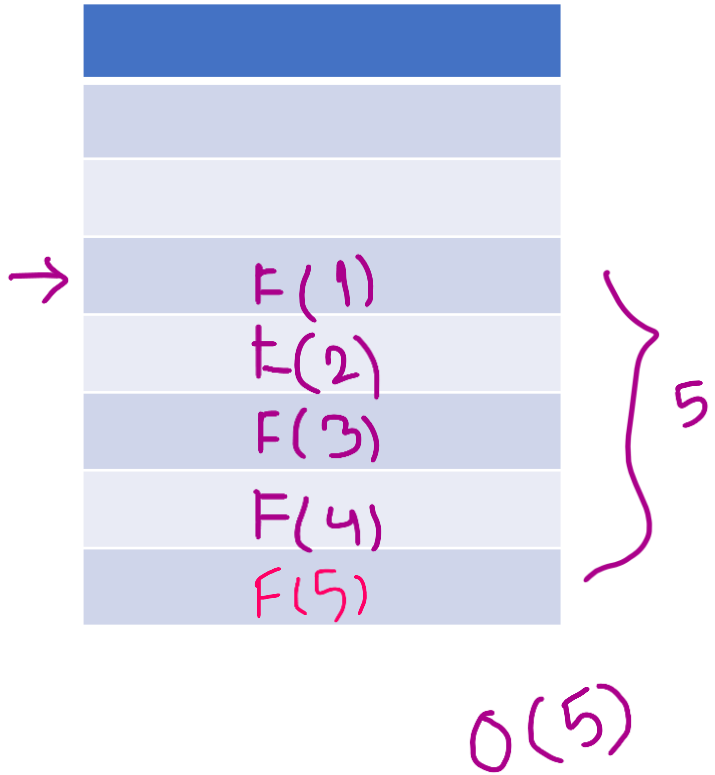
```
Fib(n){
   if (n<=1)
            return n;
    else
        return Fib(n-1)+Fib(n-2);
```

Recursion tree



Depth 4

Memory stack



O(5)

What would be the space complexity of Fibonacci recursion of Fib(5)?

https://www.youtube.com/watch?v=dxyYP3BSdcQ

Cont…

- This example illustrates how recursion is only useful if the underlying computational structure
- Could we improve fib? Yes: one standard strategy is to save the results of each recursive evaluation so that we only need to consider each value of n a single time; this approach is called **memoization**.

Memoization code: https://www.javabrahman.com/gen-java-programs/recursive-fibonacci-in-java-with-memoization/

Ask AI (5 mins time)

- Fibonacci memoization and Fibonacci iterative approach which one is better?

So, which is better?

Feature	Iterative For-Loop 🏆	Memoized Recursion
Time	O(n)	O(n)
Space	O(1)	O(n)
Simplicity	Very simple	Slightly more complex
Stack Safety	No recursion	Recursive (risk of stack overflow for large n)
Flexibility	Less flexible	Can easily adapt to other DP problems

Both has time complexity O(n)! Then why iterative is better?

- **Memoization** uses **recursion**, which involves:
  - Repeated function calls
  - Maintaining the **call stack**
  - Extra time for checking and storing values in a dictionary
- **Iterative** version avoids all that:
  - Just a simple loop and a few variables.
  - No recursion, no dictionary lookups.

- Memoization stores all previous results in a dictionary → uses more memory.Dictionary operations (hashing, lookup, insert) are fast but not as fast as basic variable assignments.
- Iterative version only keeps two variables (a and b), so it's leaner and more cache-friendly.

Binary Search

- Let’s watch the video https://www.youtube.com/watch?v=xrMppTpoqdw

Time complexity calculation of Binary Search

```java
public static int binarySearch(int[] a, int v, int left, int right) {

  // Base case -- terminate the search with no result
  if (left > right) {
    return -1;
  }

  // Calculate the middle element
  int mid = (left + right) / 2;

  // Success
  if (a[mid] == v) {
    return mid;
  }

  // Recursive case #1 -- left half
  else if (a[mid] > v) {
    return binarySearch(a, v, left, mid - 1);
  }

  // Recursive case #2 -- right half
  else {
    return binarySearch(a, v, mid + 1, right);
  }
}
```

Input size n

Input size n/2

Input size n/2

Recurrence relation:

T(n) = T(n/2)+C where n>1
T(1)=1

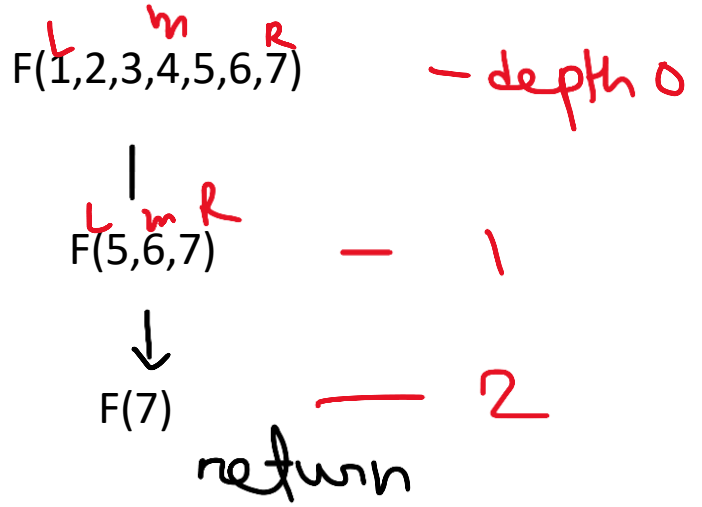
T(n) = T(n/2)+C
= T(n/4)+C +C
= T(n/8)+ 3C
.....
= $T(n/2^k) + KC$

Now $n/2^k=1$
=> $n = 2^k$
=> k = log n (2 base log)

So, T(n) = T(1) +C*log n
~ O(log n)

Space complexity of binary search is also O(log n)

- Input array 1,2,3,4,5,6,7 and our target is 7



Memory stack
F(7)
F(5,6,7)
F(1,2,3,4,5,6,7)

3 ≃ n/2

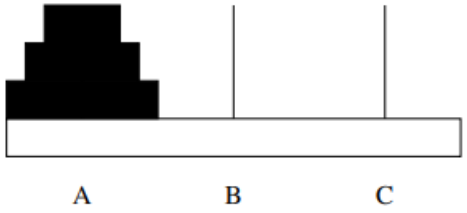
≃ O(log n)

Tower of Hanoi

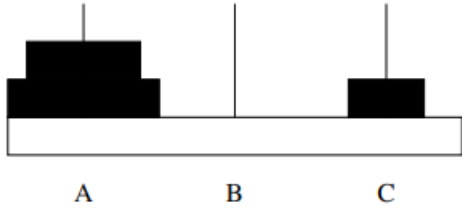
The puzzle has the following two rules:

1. You can’t place a larger disk onto a smaller disk
2. Only one disk can be moved at a time

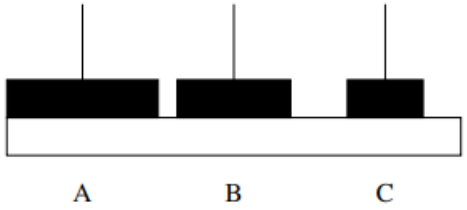
Tower of Hanoi for 3 disk



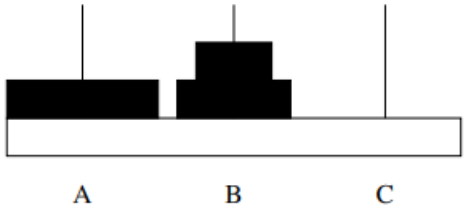
1. Move the small disk from A to C.



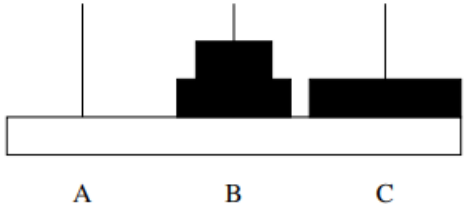
2. Move the middle disk to the middle peg (B).



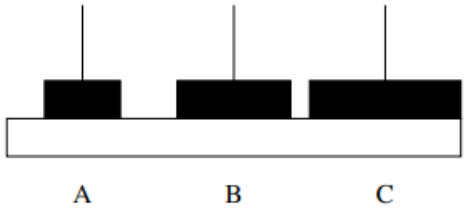
3. Move the small disk from C to B.



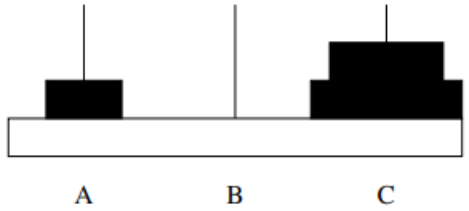
4. Move the largest disk from A to C.



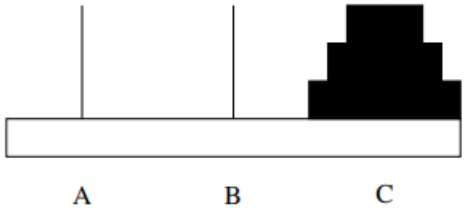
5. Move the small disk from B to A.



6. Move the middle disk from B to C.



7. Move the small disk from A to C.



Therefore, the full solution for three disks requires seven moves.

Let’s play Tower of Hanoi with 4 discs and write the number of moves

Move	Disk	From	To
1	1	A	B
2	2	A	C
3	1	B	C

Complete the moves until all the discs are transferred in order from Peg A to C

Results

Move	Disk	From	To
1	1	A	B
2	2	A	C
3	1	B	C
4	3	A	B
5	1	C	A
6	2	C	B
7	1	A	B
8	4	A	C
9	1	B	C
10	2	B	A
11	1	C	A
12	3	B	C
13	1	A	B
14	2	A	C
15	1	B	C

step 1: move top n - 1 disks from source peg to storage peg - repetitive

step 2: move one disk to destination – one step

step 3: move top n - 1 disks from storage peg to destination peg - repetitive

Go to this link for code
https://www.geeksforgeeks.org/java-program-for-tower-of-hanoi/#

Time Complexity

```
towerOfHanoi(n,…){
        if (n==1)
                return;
        towerOfHanoi(n-1, …)
        print (…)
        towerOfHanoi(n-1, …)
}
```

Recursive equation
T(n) = 2T(n-1)+1 and T(1)=1; 1 disk requires 1 movement,
And we already saw T(2)=3 and T(3)=7

T(1) = 2T(1-1)+1
=> 1 = 2T(0)+1
=> T(0)=0

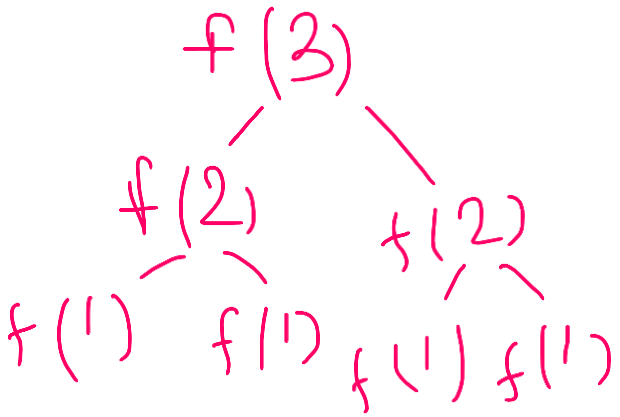
T(n) = 2(2T(n-2)+1)+1 = 4T(n-2)+3
=8T(n-3)+7
= 2*(8T(n-4)+7) +1 = 16T(n-4)+15
……
= $2^k T(n-k) + \left(2^k - 1\right)$
n-k = 0, k=n

T(n) = $2^n - 1$ so O(2^n)

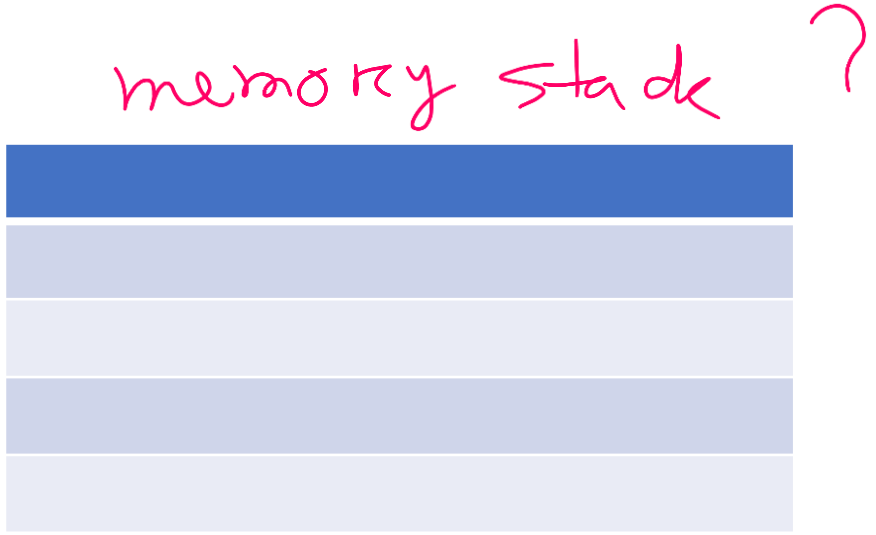
Space Complexity

Recursion tree for moving 3 disks

Recursion calls



O(3)



For n disk to move it would be O(n)