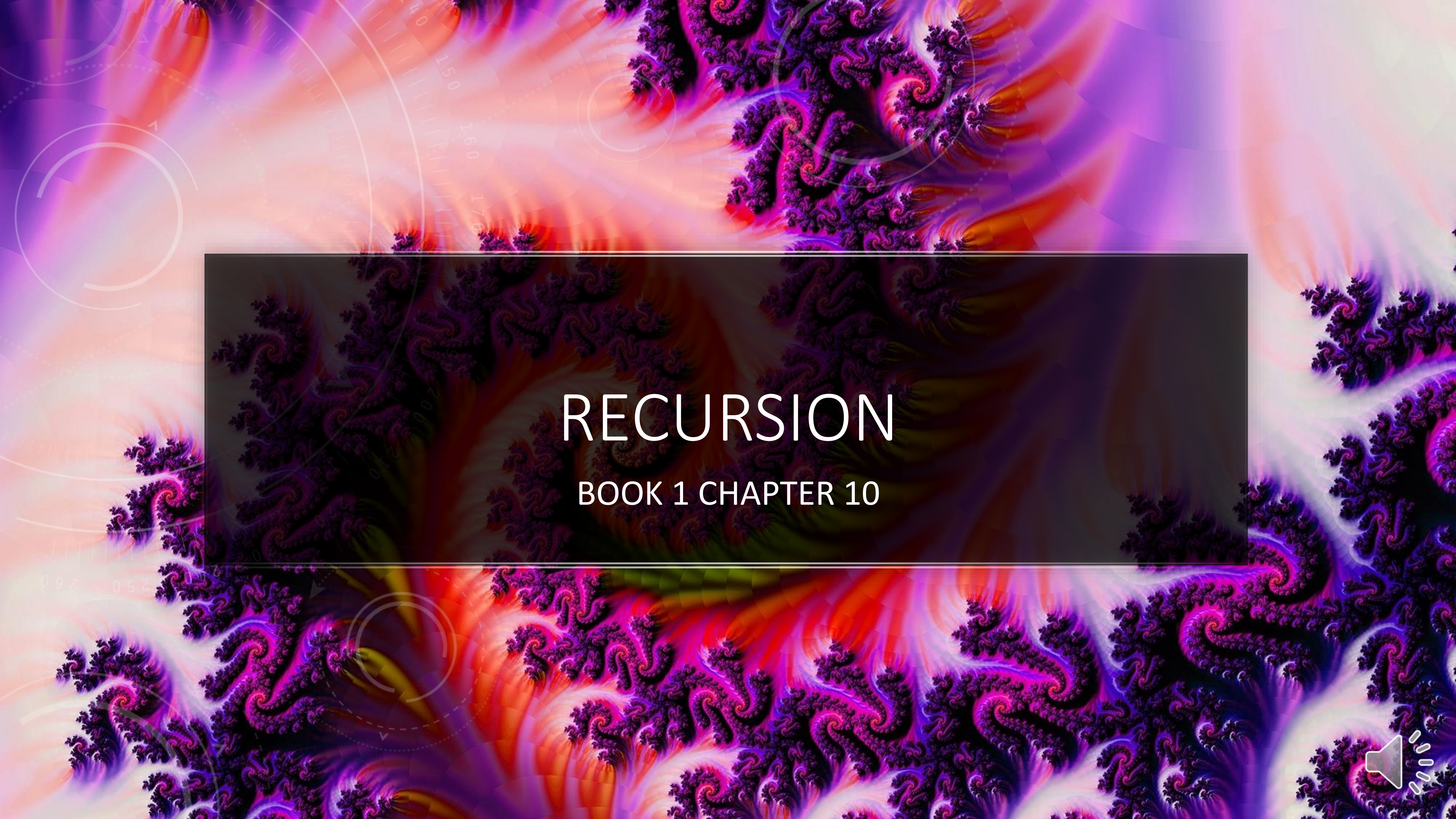
CS-218 DATA STRUCTURES AND ALGORITHMS

LECTURE 17

BY UROOJ AINUDDIN



RECURSION

BOOK 1 CHAPTER 10

IN THE LAST LECTURE…

We investigated the space complexity of quicksort in best and worst cases.

We looked at ways to optimize quicksort.

We were introduced to the run-time stack.

RECURSIVE FUNCTIONS

THE ULTIMATE APPLICATION OF STACKS

DEFINITION

- A function that calls itself is known as a **recursive** function.
- One instance of a function *func* calls another instance of the same function *func*.
- On the return path, one instance of the function *func* returns to another instance of the function *func*.
- Recursion cannot take place without stacks.
- The run-time stack is used to keep track of order of function calls in recursion too.

EXAMPLE #1: THE FACTORIAL FUNCTION

Iterative definition

$$n! = n.(n-1).(n-2)\ldots 2.1$$

Recursive definition

$$n! = \begin{cases} n.(n-1)!\ \forall n > 0 \\ 1, n = 0 \end{cases}$$

Code for iterative definition

```
def itfac(n):
    assert n > -1, "Negative number"
    prod = 1
    for i in range(2,n+1,1):
        prod *= i
    return prod
```

space: $O(1)$
time: $O(n)$

Code for recursive definition

```
def refac(n):
    assert n > -1, "Negative number"
    if n == 0:
        return 1
    return n * refac(n-1)
```

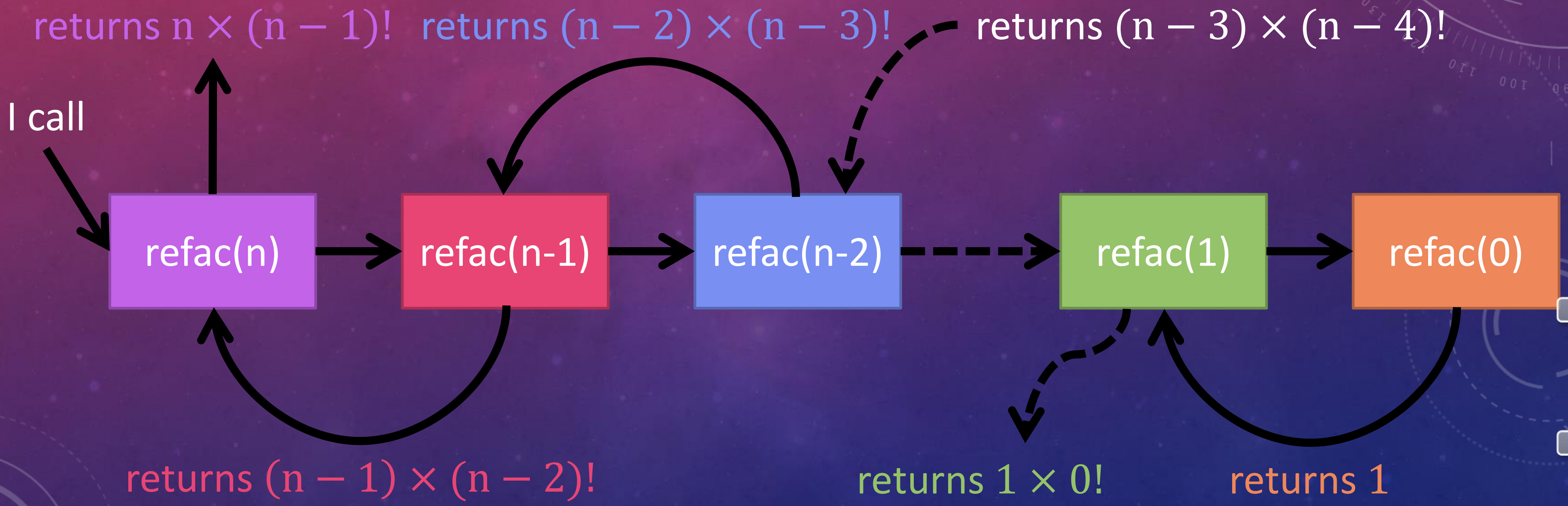
space: ?
time: ?

HOW MUCH SPACE AND TIME DOES THE RECURSIVE FACTORIAL TAKE?

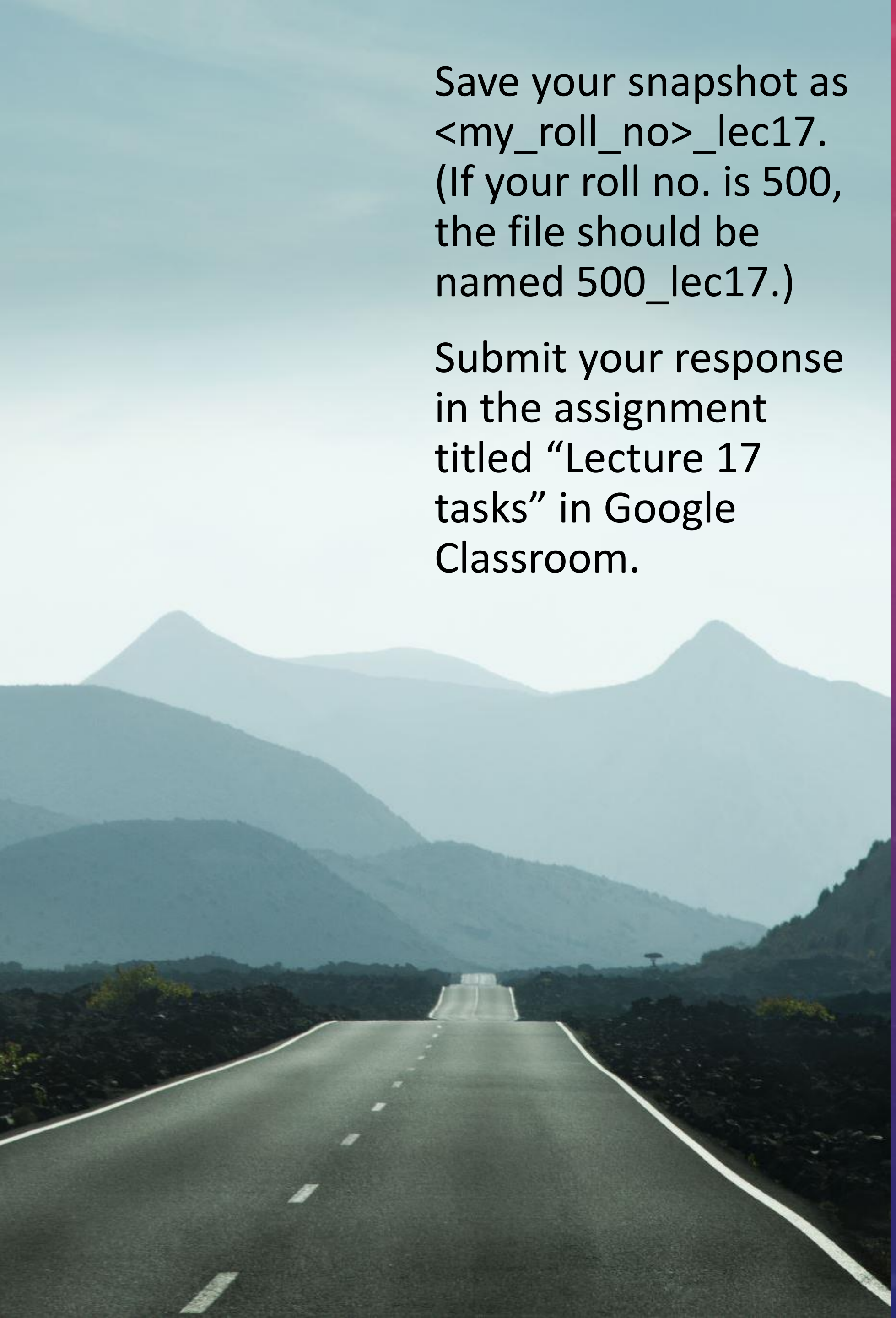
```
def refac(n):
    assert n > -1, "Negative number"
    if n == 0:
        return 1
    return n * refac(n-1)
```

time: $O(n)$

space: $O(n)$



returns $n \times (n-1)!$
returns $(n-2) \times (n-3)!$
returns $(n-3) \times (n-4)!$
I call
refac(n)
refac(n-1)
refac(n-2)
refac(1)
refac(0)
returns $(n-1) \times (n-2)!$
returns $1 \times 0!$
returns 1



Save your snapshot as <my_roll_no>_lec17. (If your roll no. is 500, the file should be named 500_lec17.)

Submit your response in the assignment titled “Lecture 17 tasks” in Google Classroom.

TASK

Which of the two implementations of the factorial function is better in your opinion. Why do you think so?

THE IMPORTANCE OF THE BASE CASE

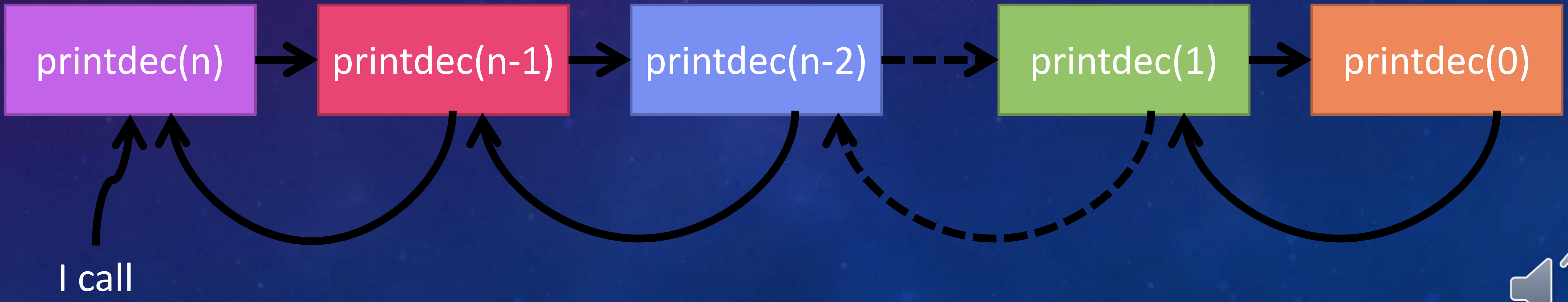
- A recursive function is incompletely defined if the base case is not mentioned, and cannot be implemented in code, as such a recursion will never end!
- There must be some input for which output can be furnished without recursion.

EXAMPLE #2: PRINTDEC

```
def printdec(n):
    if n > 0:
        print(n)
        printdec(n-1)
```

space: $O(n)$
time: $O(n)$

```
Output
n
n-1
n-2
⋮
1
```

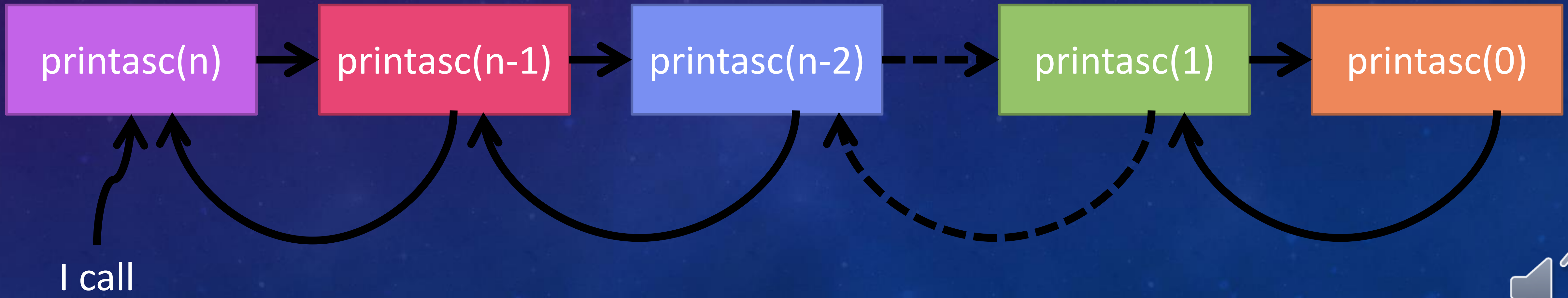


EXAMPLE #3: PRINTASC

```
def printasc(n):
    if n > 0:
        printasc(n-1)
        print(n)
```

space: $O(n)$
time: $O(n)$

```
Output
1
⋮
n-2
n-1
n
```



EXAMPLE #4: THE FIBONACCI SEQUENCE

Definition

$$F(n) = \begin{cases} F(n-1) + F(n-2), \forall n > 1 \\ 1, n = 0 \text{ and } n = 1 \end{cases}$$

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Iterative implementation

```
def fib(n):
    assert n >- 1
    fib=[1,1]
    for i in range(2,n+1,1):
        new=fib[i-1]+fib[i-2]
        fib.append(new)
    return fib[n]
```

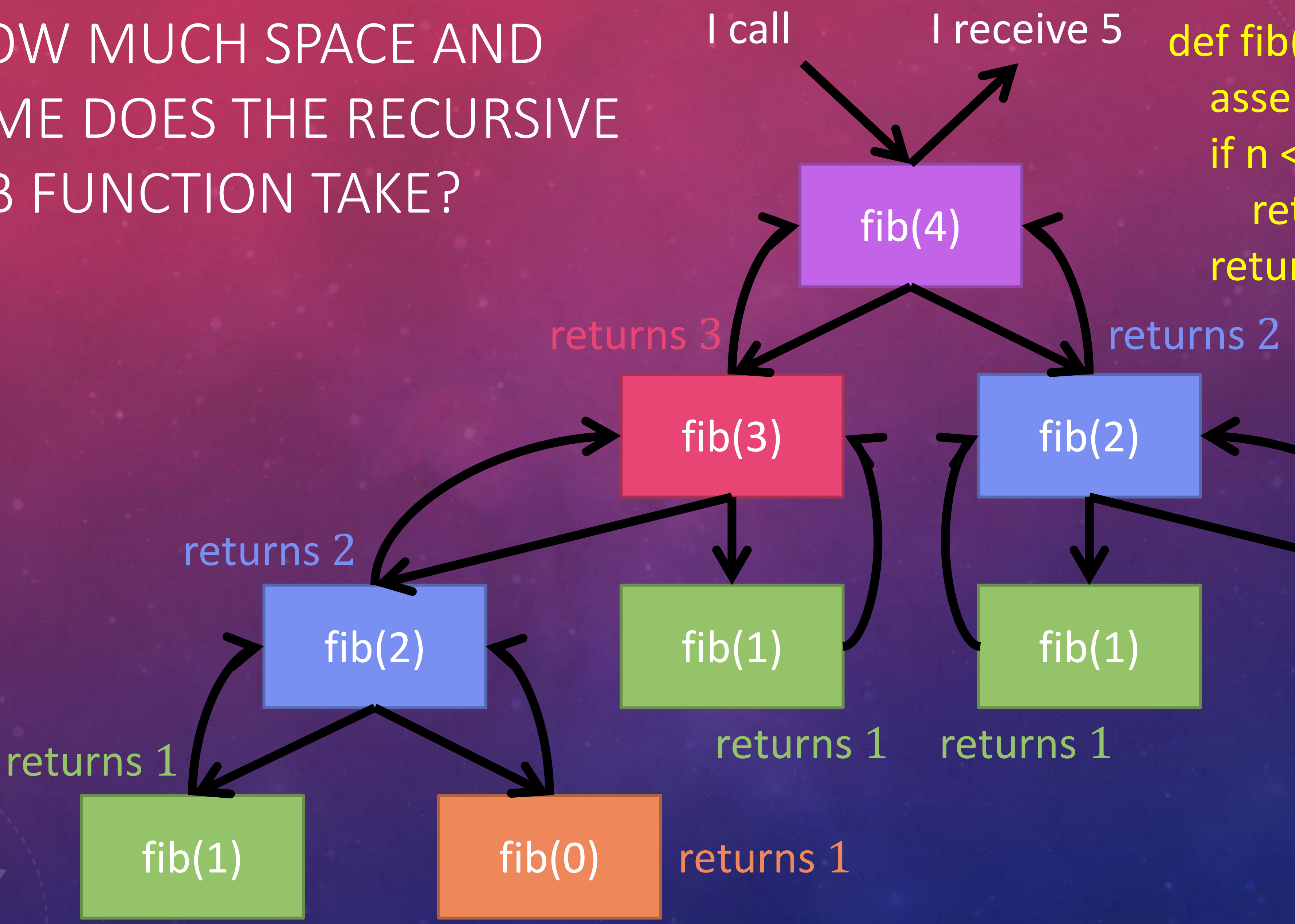
space: $O(n)$
time: $O(n)$

Recursive implementation

```
def fib(n):
    assert n >- 1
    if n <= 1:
        return 1
    return fib(n-1) + fib(n-2)
```

space: ?
time: ?

HOW MUCH SPACE AND TIME DOES THE RECURSIVE FIB FUNCTION TAKE?

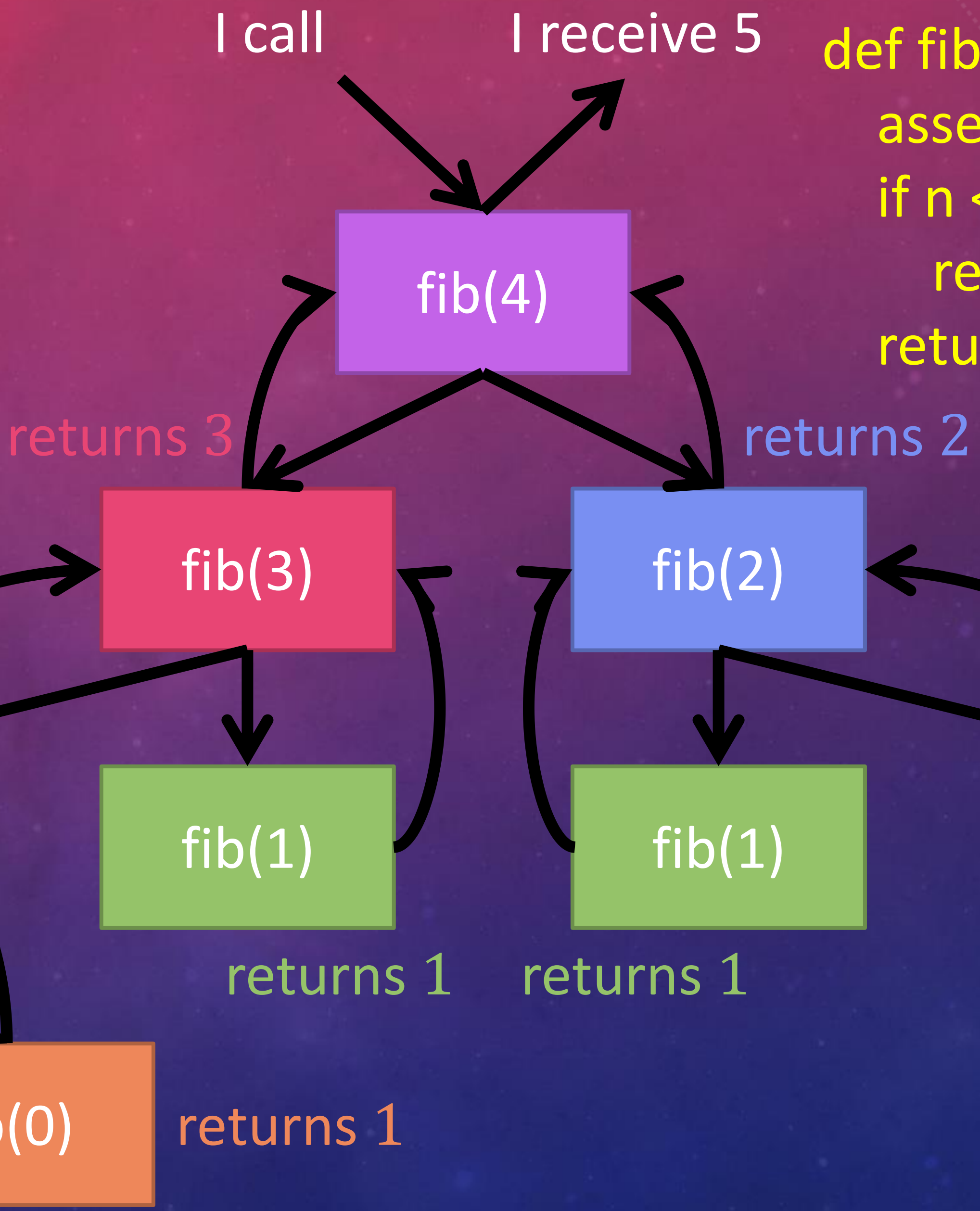


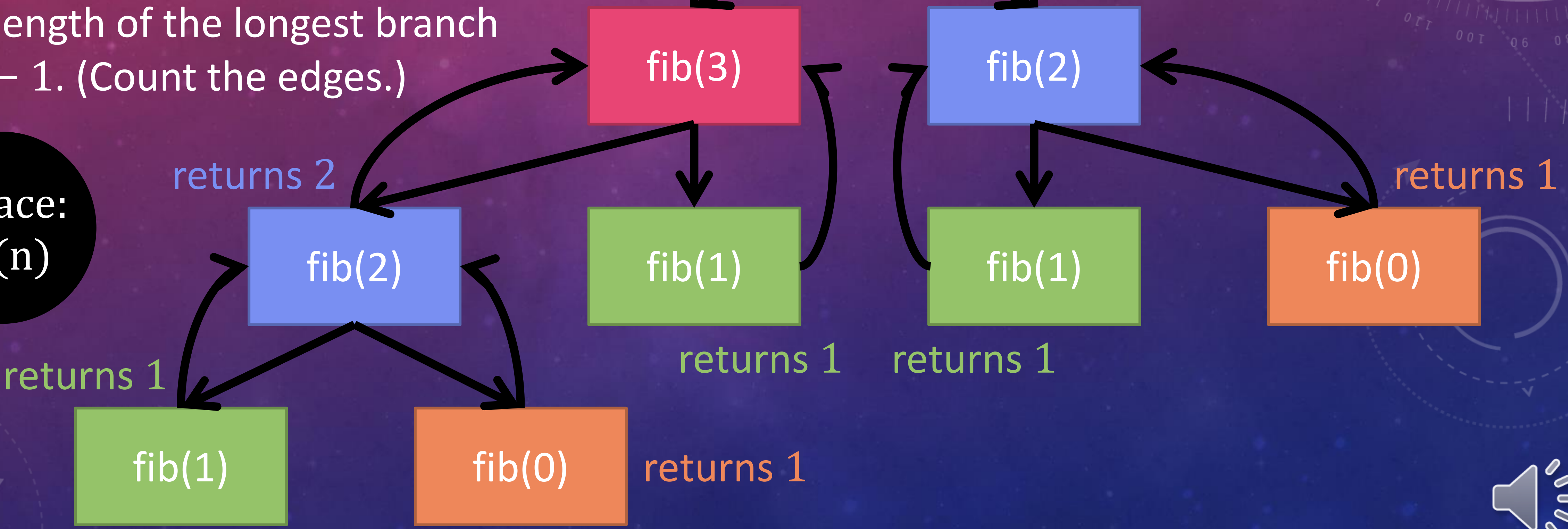
```
def fib(n):
    assert n >- 1
    if n <= 1:
        return 1
    return fib(n-1) + fib(n-2)
```

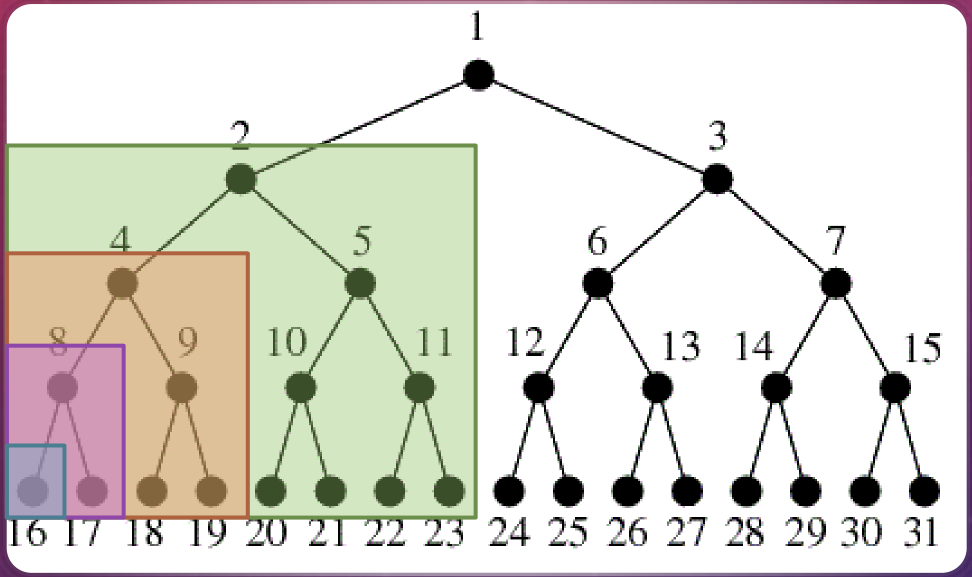
- The tree of recursive calls for fib(n) has n levels, so it has height n.
- The most pushes to the run-time stack happen in the longest branch of the tree.
- The length of the longest branch is $n - 1$. (Count the edges.)

space: $O(n)$

```
def fib(n):
    assert n >- 1
    if n <= 1:
        return 1
    return fib(n-1) + fib(n-2)
```







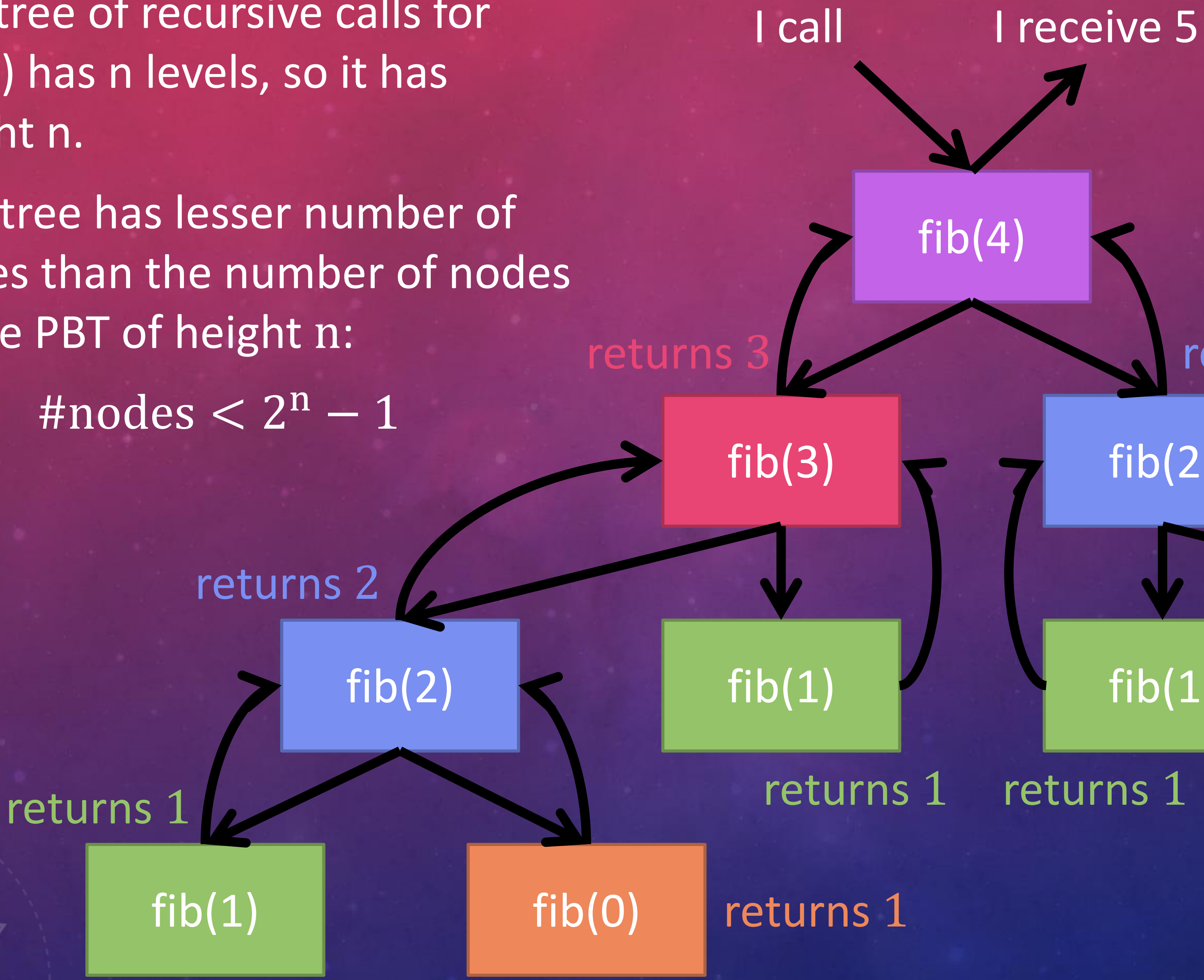
THE PERFECT BINARY TREE (PBT)

- The tree shown here is a perfect binary tree of height 5 because:
    - Each node in the last level has 0 children.
    - All other nodes have 2 children.
    - There are 5 levels in the tree.
- The left and right sub trees of the root node are perfect binary trees of height 4.
- The number of nodes x in a perfect binary tree of height h is:

$$x = 2^{h} - 1$$

- The tree of recursive calls for fib(n) has n levels, so it has height n.
- This tree has lesser number of nodes than the number of nodes in the PBT of height n:

$$\#\text{nodes} < 2^n - 1$$



```
def fib(n):
    assert n >- 1
    if n <= 1:
        return 1
    return fib(n-1) + fib(n-2)
```

- This tree has greater number of nodes than the number of nodes in the PBT of height $n - 1$:

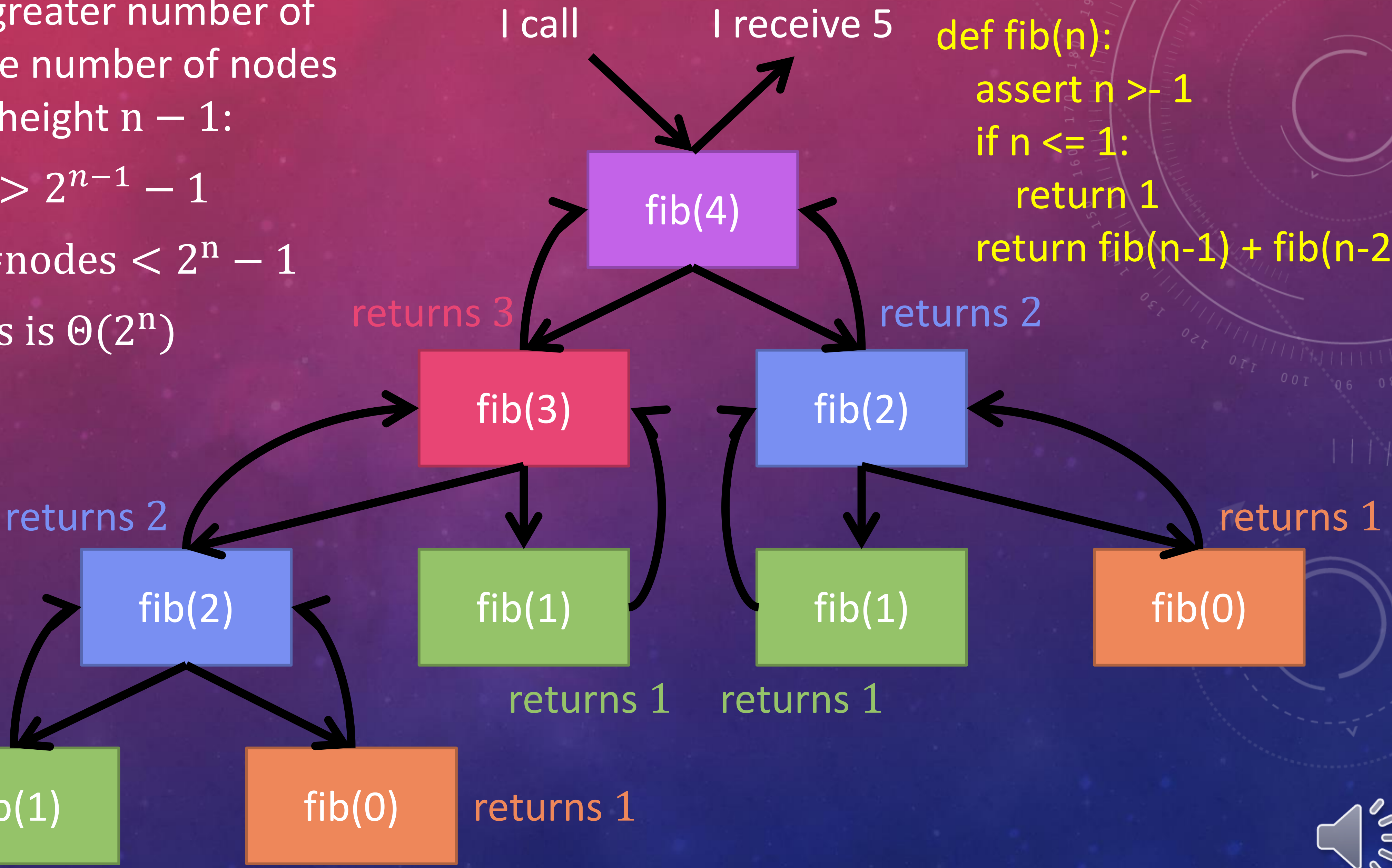
$$\#nodes > 2^{n-1} - 1$$

$$2^{n-1} - 1 < \#\text{nodes} < 2^{n} - 1$$

$$\#\text{nodes is } \Theta(2^{n})$$

time: $O(2^{n})$

```
def fib(n):
    assert n >- 1
    if n <= 1:
        return 1
    return fib(n-1) + fib(n-2)
```

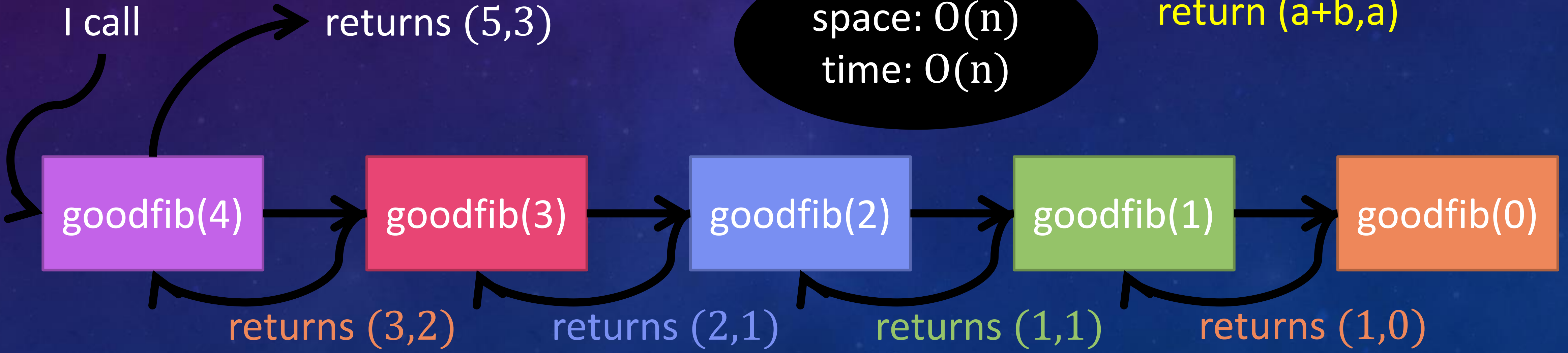


A BETTER FIBONACCI FUNCTION

- The first element of the returned tuple is the nth Fibonacci term.
- This function eliminates redundancy in calculations present in code of fib(n).

```
def goodfib(n):
    assert n>-1
    if n==0: return (1,0)
    (a,b)=goodfib(n-1)
    return (a+b,a)
```

space: $O(n)$
time: $O(n)$



SO WHAT DID WE LEARN TODAY?

We implemented the iterative and recursive versions of the factorial function.

We looked at two recursive functions for printing values.

We implemented the iterative and recursive versions of the function to find the nth term of the Fibonacci sequence.

We implemented a better function to find the nth term of the Fibonacci sequence.

We found time and space complexities of all codes.

THINGS TO DO

- Read the book!
- Submit your answers to the tasks in this lecture.
- Note your questions and put them up in the relevant online session.
- Email suggestions on content or quality of this lecture at uroojain@neduet.edu.pk