Complexity Crash Course

Thiru, IT1007 2017

What is Time and Space Complexity?

Most of the time, your functions interact with inputs. Eg,

```python
def print_every_element(lst):
  for elem in lst:
    print(elem)
```

What we want to know is for a given code, how does the **time and **space needed** __**scale with the input**__.

For the example above, the:

- time needed is O(n)
  - O(n) means it scales linearly with the input
- Space needed is O(1)
  - The space needed is **independent** of the input. What is space? Its memory that you need to store intermediate computations (working memory for computations)

IMPORTANT1: You only care about the **WORST-CASE complexity.** Read the general advice section at the bottom for more info.
IMPORTANT2: You ignore constants. See general advice section below.

Common Complexities you Might get Tested On

1) O(1) → constant time/space (independent of input)

__Time complexity example:__ finding an element in a dictionary.

```python
x = {"potato":3}

x["potato"] # This is time O(1) because i dont care how long the dictionary is
```

__Space complexity example:__ print_every_element. Print does not require space.

2) O(n) → linear

The complexity scales with the input.

Time complexity example: print_every_element. The length of the array is N, so as N increases, your time increases linearly.

Space complexity example: Factorial Recursion.

```
def fact_recursive(n):
  if n == 1:
    return 1
  return n + fact_recursive(n-1)
```

Recall: Recursion operates by holding the intermediate value in memory, then calling the other computations until everything is done, then adding all the results.

In this case, if i call fact_recursive(10), i will be **holding the results** of fact_recursive(9), fact_recursive(8)... fact_recursive(1) in **memory**, then finally summing them up in the end.

If i input n, i get n calls to fact_recursive, each call stores one item. Hence, the space complexity grows by n. → O(n)

3) O(n^2) / O(n*m) / O(n^3) - polynomial

You often get polynomial time when doing nested loops.

Time complexity example:

```
lst1 = [1,2,3]
lst2 = [3,4,7,9,10]

for entry in lst1:
  for entry2 in lst2:
    ## do something
```

Let's call length of lst1 as n, length of lst2 as m. You're nested looping through both, means if i print the outputs:

1,3

1,4

1,7

..

2,3

2,4

2,7

..

The time complexity in this case is n*m since the arrays are of different length. Hence, your complexity is polynomial time of **O(n*m). If n == m ,then this would be O(n^2).**

Time complexity of bubble sort is O(n^2). (why?)

<u>Space complexity example: </u>Matrix Multiplication.
Say you have a function:

```
def matmul(mat_a,mat_b):
    ##insert magic here
    return multiplied
```

In the worst case input, If your matrix A is 4x1 and matrix B is 1x4, → your new matrix is 4x4! You need to create a new matrix that's much bigger than A and B to store the results.

4) O(2^n) / O(a^n) - exponential

Best example for time complexity is recursive fibonacci.

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

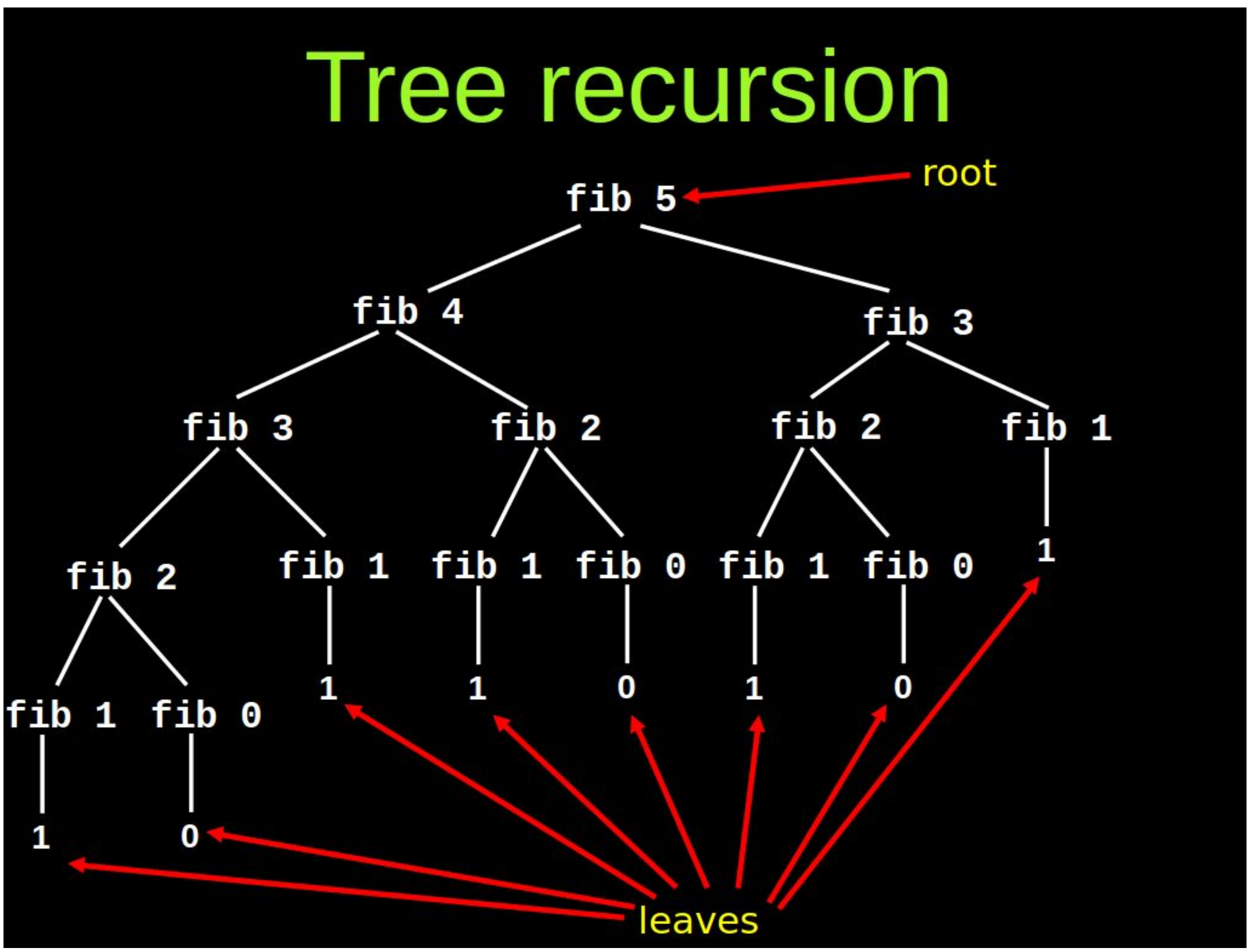
Each fib call calls 2 other fibs.

1 → 2 → 4 → 8 → 16

See the pattern? Hence, this grows exponentially in time. So time complexity is O(2^n)

However, each fib only returns 1 number, so for n fibs, you need n space (you only care about the height of the tree). Therefore, the space complexity is O(n).

Hopefully this diagram makes it clearer



Number of leaves → 2^n
Height of tree → n

Space complexity example: Likely not tested.

5) **O(logn) - logarithmic**

Logarithmic means that for input of size n, the number of computations you need to do is inverse exponential. Best example for time: binary search.

Assuming you have a sorted array:
Lst = [1,45,78,100,103]

Every iteration, you are cutting away half the array size. Eg, if i want to find 1,

First iteration: check middle element → 78 → dont bother checking second half.

new_lst= [1,45,78] → check middle element → 45 → cut away
new_lst = [1] → found!
For an array of length 5, i only did 2 cuts!

Space complexity for logn: Likely Not tested

6) **O(nlogn) - linear*logarithmic**

Best examples for time complexity of nlogn are the sorts: merge sort, quicksort, timsort. But you guys likely wont be tested on this. You can google it if you’re really interested.

How to Analyze?

General Advice

- You only care about WORST CASE complexity
    - eg , if you have one dual nested loop and one normal loop, the complexity is O(n^2) + O(n), But your answer is O(n^2) because you only care about the worst case
- You ignore constants
    - If you do a loop 5 times in a function across an array, the time complexity is O(5n). But you dont care about constants. So the answer is O(N).
    - Important: I dont know why your notes shows differently for space complexity, so follow what your notes says for space i guess.

- Easy way: Each nested loop increases complexity by n, if n is the length of the array and each loop is looping from beginning to end of array. Eg,

```
def nested_loop(lst):
  for i in range(len(lst)):
    for j in range(len(lst)):
      for k in range(len(lst)):
        print(i,j,k)
```

This has a time complexity of O(n^3) because for each loop, you are looping the inner value!

Eg:
i = 0, j = 0, k = 0
i = 0, j = 0, k = 1,
i = 0, j = 0, k = 2,
..
i = 0, j = 1, k = 0,
i = 0, j = 1, k = 1,
i = 0, j = 1, k = 2,
..

How many calls are made? n^3!

- For a more rigorous approach: You need to exactly see how your computation increases with your inputs across your function. You can see an example here https://stackoverflow.com/questions/360748/computational-complexity-of-fibonacci-sequence

Example Practice Q

Question 2: Alternating Series Galore [24 marks]

Consider the following alternating series s_{11}:

$$s_{11}(n) = 1 - 2 + 3 - 4 + \cdots n$$

A. **[Warm Up]** Write an **recursive** function `s11(n)` that returns the value for $s_{11}(n)$. [4 marks]

B. What is the order of growth in terms of time and space for the function you wrote in Part (A) in terms of n. Explain your answer. [4 marks]

C. Write an **iterative** function `s11(n)` that returns the value for $s_{11}(n)$. [4 marks]

D. What is the order of growth in terms of time and space for the function you wrote in Part (C) in terms of n. [2 marks]

Answers:

A. **[Warm Up]** Write an **recursive** function `s11(n)` that returns the value for $s_{11}(n)$. [4 marks]

```
def s11(n):
    if n == 1:
        return 1
    elif n%2 == 0:
        return s11(n-1) - n
    else:
        return s11(n-1) + n
```

B. What is the order of growth in terms of time and space for the function you wrote in Part (A) in terms of n. Explain your answer. [4 marks]

Time: $O(n)$
There is a constant number of basic steps in each recursive call, and there are n such recursive calls from s11(n) to s11(1).

Space: $O(n)$
To compute s11(n), there is n-1 pending operations: s11(n-1), ..., s11(1)

-1 for each missing (or incomplete) explanation.
-1 for incorrect big-O notation.

C. Write an **iterative** function `s11(n)` that returns the value for $s_{11}(n)$. [4 marks]

```
def s11(n):
    total = 0
    for i in range(1,n+1):
        if i%2 == 0:
            total -= i
        else:
            total += i
    return total
```

D. What is the order of growth in terms of time and space for the function you wrote in Part (C) in terms of n. [2 marks]

Time: $O(n)$

Space: $O(1)$