Worksheet

Let’s talk efficiency

Preliminaries

- Make sure to review content slides thoroughly
  - Time complexity is a topic that takes time to fully grasp and understand
  - Will be an important topic on exams
- For this homework, all that is required of you is to write down the time complexity of **each** function
  - Don’t need to show work for it but if you’re unsure and show work for it, might be eligible for **partial credit**
- Double check your work!
  - The smallest things can change the time complexity of a function!
- No resubmission for this homework, you only get one shot at it!
  - Make it count
- This video is not going to go over the answers for each function, will focus more on content review and examples.

Goals for this assignment

- Start getting practice with reading and identifying time complexities of functions
  - Good topic to understand know for exams as well as future homework assignments
    - “Stress test”
- Important for technical interviews for software engineering roles

In terms of goals, this assignment is relatively straight forward, it’s only testing one of the most important skills and concepts in your CS knowledge base. Make sure you get a good grasp of this!

Content Review: Time Complexity and Big-O Notation

What is “Time Complexity/Big-O Notation”?

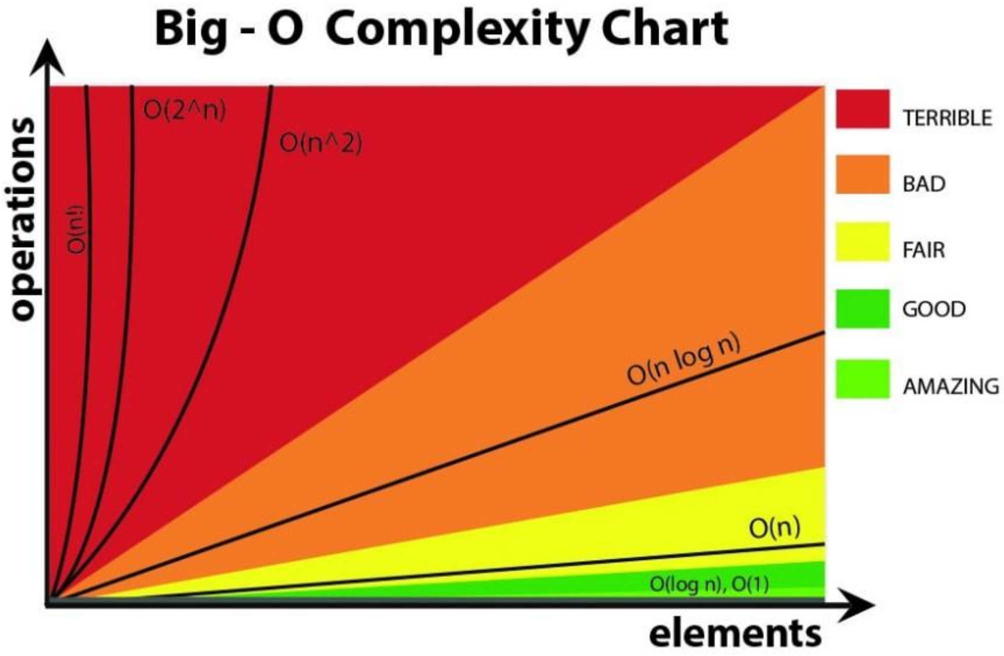
- “*Simplified analysis of an algorithm’s efficiency*”
  - Complexity is measured in terms of input size, N
  - Machine-independent
    - You may have an MBP, but let's assume all computers are equal for now
- Ways of measuring time complexity
  - **Worst-case**
  - best-case
  - Average-case
  - Later in this course: Amortized cost
- Calculating time complexity looks like the following

Time_complexity = time(statement1) + time(statement2) + time(statementN)

General Rules for analyzing time complexity

- Ignore constants
  - 3n is the same as n
    - So a function that has a time complexity of 3n is just O(n), not O(3n)
- Certain terms overtake others

$O(1) < O(\log n) < O(n) < O(n\log n) < O(N^2) < (2^n) < O(n!)$



Example 1

- Time Complexity: O(1)

Let's see why

- Each statement (line) is a basic operation (some type of math operation and variable assignment). Each line in this function takes O(1) time. Since no matter what the input, n is, the number of operations is the same, the time complexity is O(1)

```
def simple_terms(n):

    x1 = n + n
    x2 = n - 1
    x3 = n + (16 * 100)
    x4 = n * n
    x5 = n + 100000000
    print(n + 1)
```

Total Time = O(1) + O(1) + O(1) + O(1) + O(1) + O(1) = O(1)

*6 * O(1)*

Example 2

- Time Complexity: O(n)

Let's see why

- The first line of the code are O(1). However, the for-loop is dependent on the input, N for how many times it will run. Since we drop lower order terms for calculating time complexity, the time complexity of this function is O(n)

```python
def example_2(n):

x = 15 + (n * 100)
for x in range(0, n):
    print(x)
```

Total Time = O(1) + O(n) = O(n)

Example 3

- Time Complexity: $O(n^3)$

Let's see why

- Remember that we consider the **worst-case** time complexity when analysis code. Since the worst case in this function is when the else if statement is evaluated, the time complexity is $O(n^3)$

```
def tricky(input):

if (isTrue):
# some operation that is O(nlogn)
elif (some_case == True):
# some some operation that is O(n^3)
else:
# some operation that is O(1)
```

Total Time = O(nlogn) + O(n^3) + O(1) = $O(n^3)$

Example 4

- Time Complexity: $O(n^2)$

Let's see why

- The first line is O(1). The first for-loop is O(n) since it its dependent on our input size, n.
- The two stacked for-loops are both O(n) respectively but combined, they are $O(n^2)$
- Since the two for-loops make our code slower, the overall worst-case time complexity is $O(n^2)$ after evaluating each operation

```
x = 10 + (15 * 20)

for i in range(0, n):
    print(x)

for j in range(0, n):
    for k in range(0,n)
        print(j * k)
```

Total Time = $O(1) + O(n) + O(n^2) = O(n^2)$

Example 5

- Time Complexity: $O(2^n)$

Let's see why

- Let say that n = 2, then we have to do 3 recursive calls. fn(2) -> fn(1) and fn(0)
- For n = 3, we have 5 recursive calls
- And n = 4, we have 9 recursive calls
- Since for each time we increase n by 1, we have do at most **double the operations** -> $O(2^n)$

```
def fn(n):

    if (n < 0): return 0
    if (n < 2): return n
    return fn(n-1) + fn(n-2)
```

That’s all for the Worksheet!