Programming Club Meeting 16 Slides

Big O Notation

Description

- Big O notation is used to represent the complexity of code and how it scales
- Describes the way that the number of steps involved increases as the input size does
- Very broad system that puts algorithms into rough categories
- Because it looks at the scalability of an algorithm, a “slower” algorithm may be faster with smaller inputs

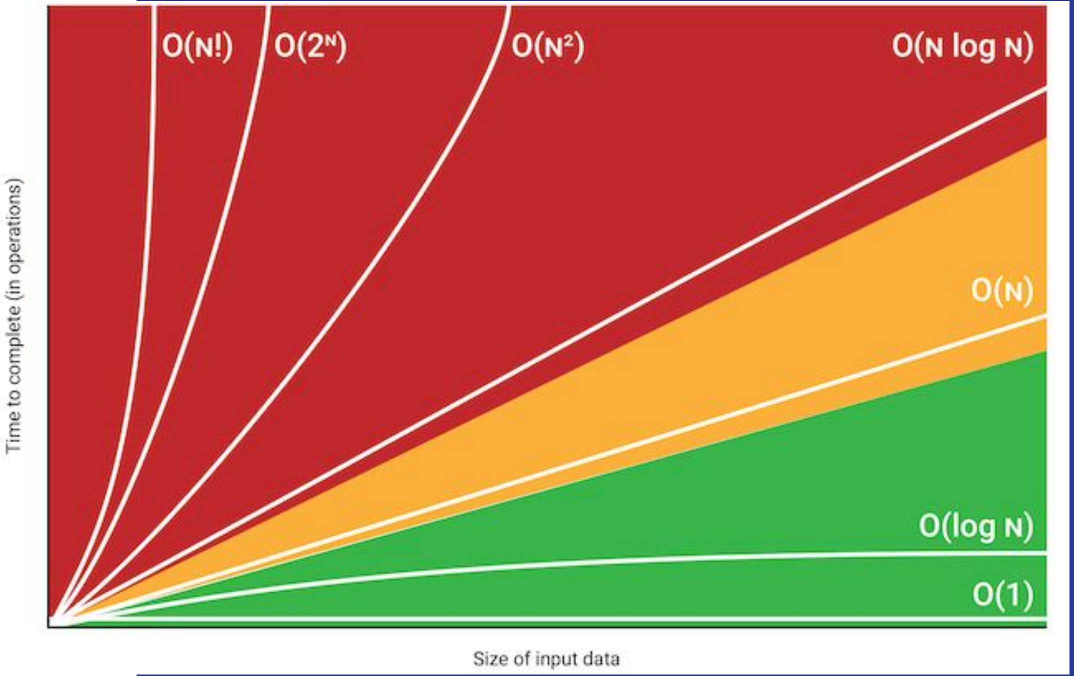
Basics

- Looks something like “O(n)”
- Constants and coefficients are removed as are lower values
  - O(3n) → O(n)
  - O(n^2 + n + 1) → O(n^2)
  - Because what qualifies as a step varies
- Can look at the best, average, or worst complexity
  - Generally look at the average because it’s often the most important
- “n” represents the input size
- Won’t really need a complex understanding until significantly later, hoping to give you a general idea of the topic

Graphic

Ex:

- Mowing the lawn (by area vs. side length)
- Sorting a deck of cards



Complexities P1

- O(1) - constant time, usually ideal complexity though often not possible, usually only for simple algorithms
- O(log n) - logarithmic time, often related to divide and conquer algorithms, pretty good complexity
- O(n) - linear time, usually a good complexity but sometimes undideal, often things that have to go through each element in a list like linear search

Code

```python
# O(1)
print("1")

print()

# O(log n)
def binarySearch(lst: list, target: int) -> int:
    """
    Returns the index of the target if present,
    otherwise returns -1.
    """
    low = 0
    high = len(lst) - 1
    mid = 0

    while low <= high:
        mid = (high + low) // 2

        if lst[mid] < target: # ignore left
            low = mid + 1
        elif lst[mid] > target: # ignore right
            high = mid - 1
        else: # found target
            return mid

    return -1 # target not in list
print(binarySearch([1, 2, 3, 4, 5, 6, 7, 8], 8))

print()

# O(n)
n = 5
for i in range(n):
    print(i)
```

Output

```
1

7

0
1
2
3
4
```

Complexities P2

Code

```python
# O(n log n)

# O(n^2)
n = 2
for i in range(n):
    for j in range(n):
        print(i*n+j)

print()

# O(n^3)
n = 2
for i in range(n):
    for j in range(n):
        for k in range(n):
            print(i * n**2 + j * n + k)

print()

# O(2^n)

# O(n!)
print("a: a: 1! = 1")
print("ab: ab, ba: 2! = 2")
print("abc: abc, acb, bab, bca, cab, cba: 3! = 6")
```

Output

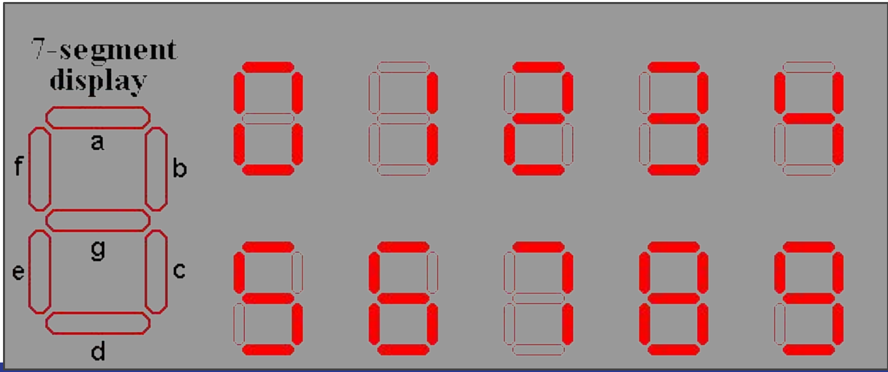
```
0
1
2
3

0
1
2
3
4
5
6
7


a: a: 1! = 1
ab: ab, ba: 2! = 2
abc: abc, acb, bab, bca, cab, cba: 3! = 6
```

- O(n log n) - pretty complicated algorithms, I don't have a good simple example, can find one here though
- O(n^2) - polynomial time, this is usually caused by loops inside of each other, can have "n" to something besides 2 depending on how many loops are nested, usually the limit for how complex an algorithm should be
- O(2^n) - exponential time, pretty complicated algorithms, I don't have a good simple example, can find one here though
- O(n!) - factorial time, often comes up in looking for all possible combinations

Practice Problems

Practice Problem 1: Matchstick Display

- Src: https://www.hackerearth.com/practice/basic-programming/input-output/basics-of-input-output/practice-problems/algorithm/seven-segment-display-nov-easy-e7f87ce0/
- Goal: Write a Python program that will output the largest value that can be created on a 7-segment display when each lit segment is represented by a matchstick. Note that the total number of matchsticks that can be used is limited and to be inputted by the user.
- Relevant Information:
    - Ex 1 - Input: 2, Output: 1
    - Ex 2 - Input: 6, Output: 111
    - Ex 3 - Input 11, Output: 71111



Practice Problem 2:

Random MAC Address

- Src: https://www.101computing.net/ip-addresses-ipv4-ipv6-mac-addresses-urls/
- Goal: Write a Python program that will generate a random MAC address.
- Relevant Information:
  - A MAC address has 6 segments separated by colons
  - Each segments holds a 2 digit hexadecimal number (meaning that each digit could be 0-9 or A-F)
  - Ex: 40:BC:06:6C:29:D7

Practice Problem 3: Estimate Square Root

- Src: https://www.101computing.net/square-root-estimation-algorithms/
- Goal: Write a Python program that will estimate the square root of an inputted number via the Babylonian method.
- Relevant Information:
  - The Babylonian method is as follows:
  - Set the variable ‘x’ to 1 with ‘number’ representing the number whose square root you are estimating
  - Set ‘x’ to (x + number / x) / 2, repeat this step 99 more times
  - The final ‘x’ value is your estimated square root

CFG Weekly Contest

- Src: https://practice.geeksforgeeks.org/events/rec/gfg-weekly-coding-contest
- Sundays 9:30am-11:00am
- 2-3 questions, answer as many as possible
- Looks like individual challenge
- Practices DSA skills

Next Meeting: Recursion