App Academy

Big O Analysis

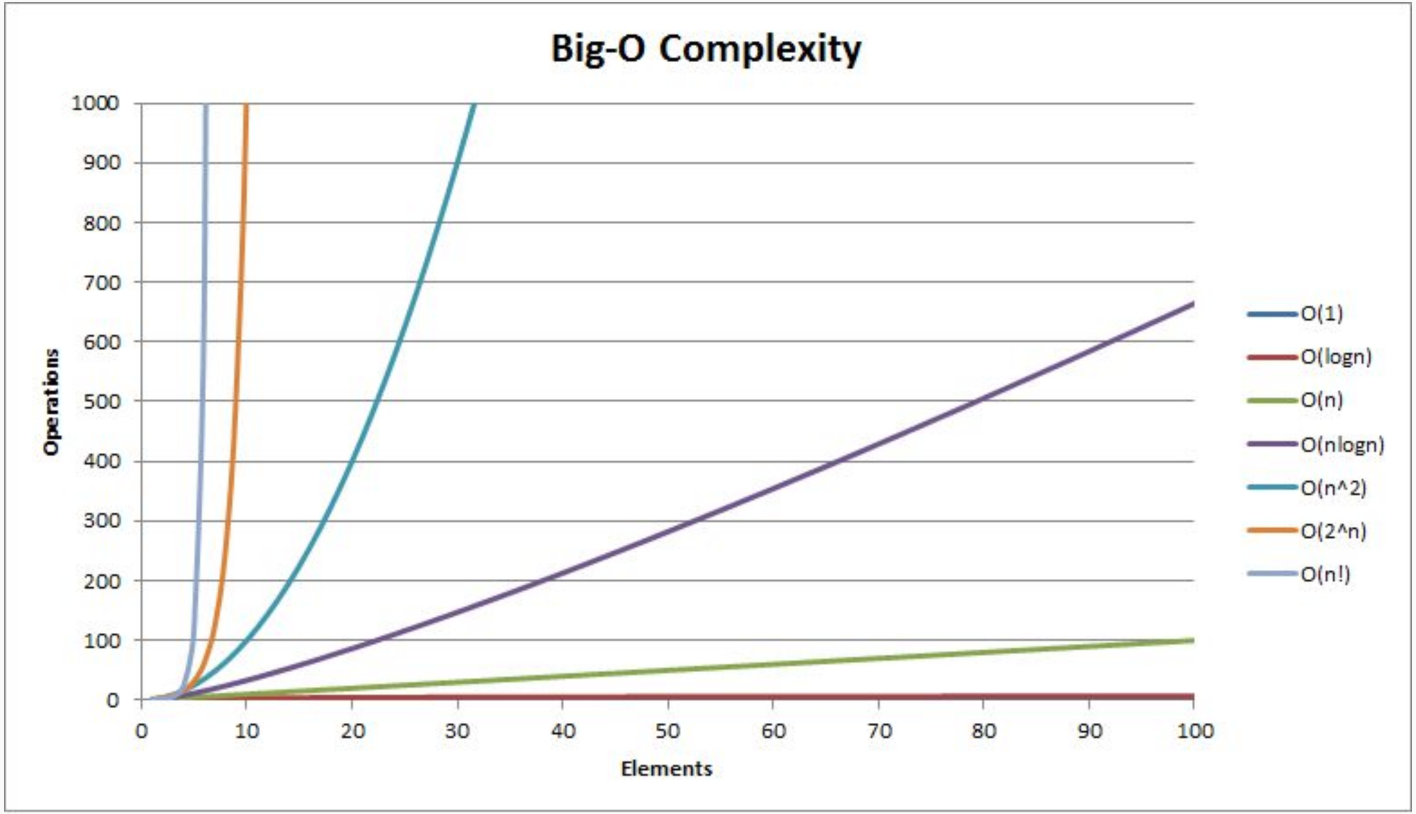
Big O Analysis

- Big O is how we analyze how efficient our solutions are
- This helps us understand how well our solutions work at scale
- There are two things we want to analyze:
  - time
  - space
- Generally, we are more concerned with time, but it is still important to understand how to analyze space efficiency as well.
- We are usually talking about worst case scenario
  - There are a few niche cases where we take the average into consideration instead. This is called “amortized complexity”. Inserting elements into dynamic arrays are an example of this.

Big O Rules

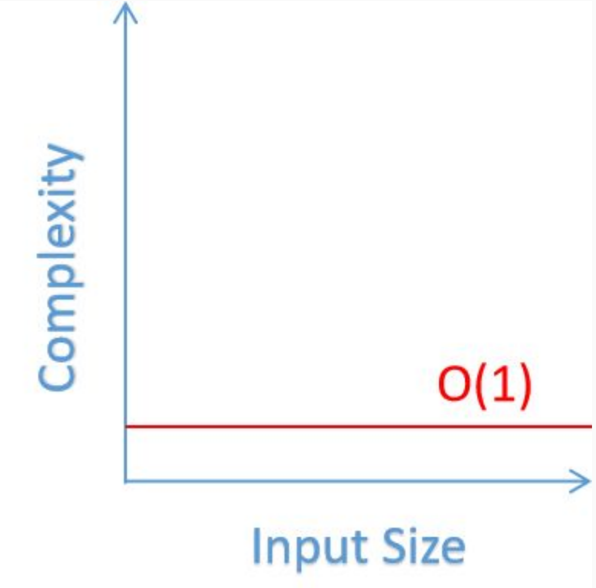
- Dropping constants
  - O(2n) is effectively the same as O(n) at large inputs
- Dropping non-dominant terms
  - O(n + log(n)) can be simplified to O(n) since O(n) is slower than O(log(n))
- Amortized time
  - Dynamic Array Example:
    - Arrays must be resized when reaching capacity, which is an O(n) operation.
    - However, this doesn't happen often.
    - If the vast majority of the time we can expect a O(1) push time complexity, we can amortize the runtime to O(1) even if the worst case is O(n).
- When figuring out complexities, ask yourself: How do my operations scale as my input gets larger?
- Always always ALWAYS define your variables. The correctness of your analysis is dependent on how you define your variables.

Big O Chart


Big-O Complexity
Operations
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

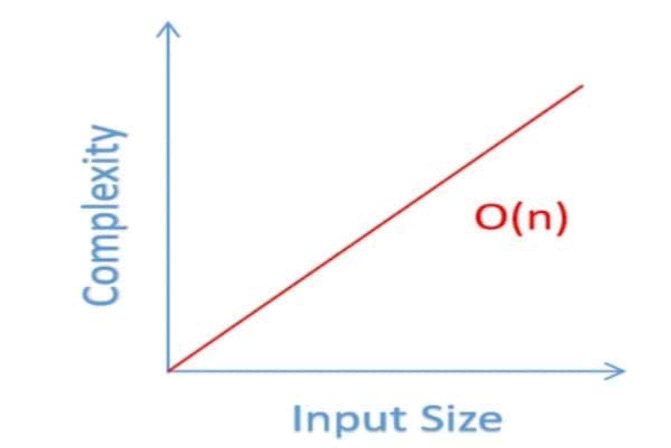
O(1) - constant

- O(1) is the most optimal complexity possible
- No matter how our input grows, our number of operations is the same
- This is represented by a flat line
- Examples:
    - Arrays
        - nums = [1,2,3]
        - nums.append(4) // pushing to end of array
        - nums.pop() // popping from end of array
        - nums[0]   // lookup
    - Hash Maps / Sets
        - hash = {}
        - hash["key"] = 10 // insert
        - "key" in hash // lookup
        - hash["key"] // lookup



O(n) - linear

- As our input size grows, our operations grow proportionally
- Note: nested loops can still be O(n); e.g. sliding window
- Examples:
    - Arrays
        - nums = [1, 2, 3]
        - for num in nums: // looping
        - 2 in nums // searching
        - nums.pop(0) // removing from beginning



$O(n^2)$ - quadratic

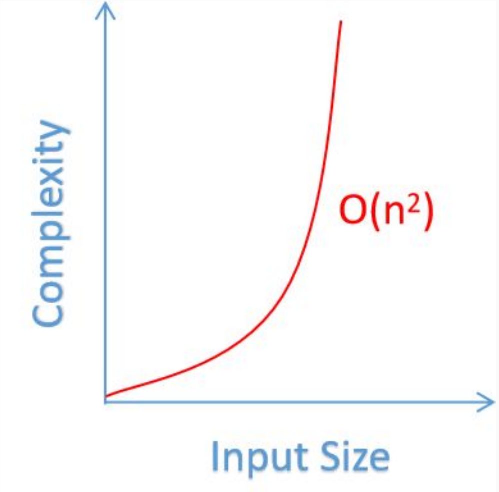
- As our input grows, our operations grow quadratically
- Examples:
  - Traversing a square matrix
    - nums = [[1,2,3], [4,5,6], [7,8,9]]
    - for row in range(len(nums)):

      for col in range(len(nums[0])):

      print(nums[row][col])
  - Get every pair of elements in array
    - nums = [1, 2, 3]
    - for i in range(len(nums)):

      for j in range(i + 1, len(nums)):

      print(nums[i], nums[j])



O(n*m)

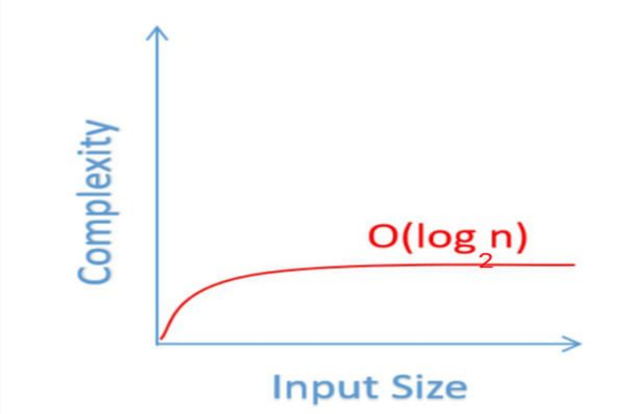
- Just because we have nested loops does not mean O(n^2)!!!
- WE MUST DEFINE OUR VARIABLES
- Example
    - Traversing a rectangle matrix
        - nums = [[1,2,3,4], [5,6,7,8], [9,10,11,12]]
        - ```
          for row in range(len(nums)):
                  for col in range(len(nums[0])):
                          print(nums[row][col])
          ```
          ```

$O(n^3)$

- Examples
  - Getting triplets of elements in an array
    - ```
      for i in range(len(nums)):
          for j in range(i + 1, len(nums)):
              for k in range(j + 1, len(nums)):
                  print(nums[i], nums[j], nums[k])
      ```
      ```

O(log(n)) - logrithmic

- As our input size grows, our operations grow logarithmically
- Marginally less efficient than O(1) but much more efficient than O(n)
- Examples
  - Binary Search
    - nums = [1, 2, 3, 4, 5], target = 6
    - l, r = 0, len(nums) - 1
    - while l <= r:

```
          m = (l + r) // 2
          if target < nums[m]:
                r = m - 1
          elif target > nums[m]:
                l = m + 1
          else:
                print(m)
                break
```

  - Pushing and popping from heaps
    - import heapq
    - minHeap = []
    - heapq.heappush(minHeap, 5)
    - heapq.heappop(minHeap)



$O(n*\log(n))$

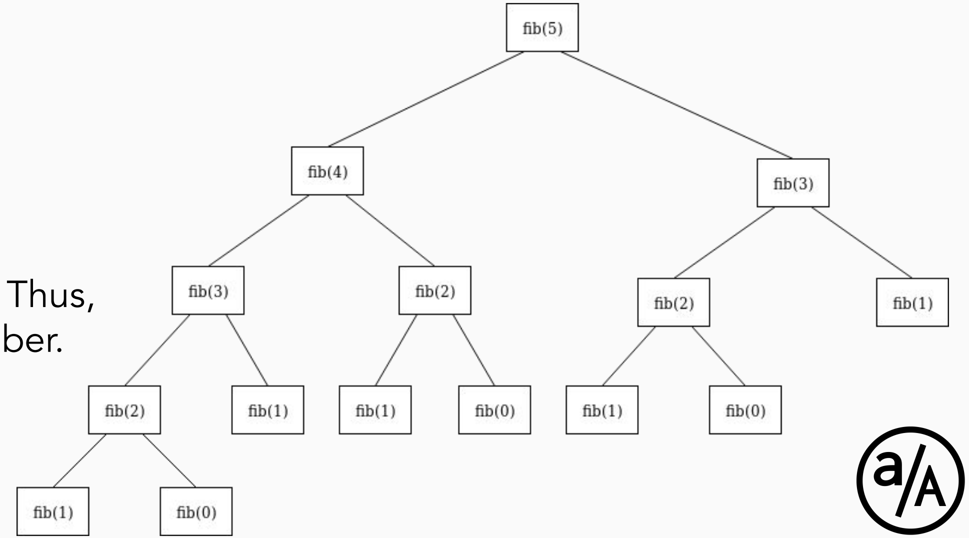
- Marginally less inefficient than $O(n)$ but much more efficient than $O(n^2)$.
- Examples:
  - Merge sort
  - Most built in sorting functions

$O(2^n)$ - exponential

- Most common with multi-path recursion
- Multi-path recursive functions can be analyzed with the following formula:

$$(\text{\# of paths})^{(\text{depth of stack})}$$

- The tree on the right represents a recursive Fibonacci function.
- Each node has at most 2 branching paths.
- The depth of the stack is 5, which is the input. Thus, we can call this $O(2^n)$ where n is our input number.



O(n!)

- Very rare and is unlikely to come up with an interview problem
- Horribly inefficient- the worst of the worst case scenarios
- The only problems that require this are very difficult problems that are considered "NP-complete" - nondeterministic polynomial-time complete
- Not necessary to put too much focus here, but you can read this article if you are interested.
- Examples
  - finding all permutations
  - "traveling salesman problem"

Questions?