Algorithms for Interview Preparation

Analysis Using Big O notation

Allan A. Zea [1] Antonio Rueda-Toicen [2]

[1]Technical University of Berlin

[2]HomeToGo GmbH

April 9, 2020

Outline

What you will learn

- The concept of time complexity.
- The concept of space complexity.
- How to describe the time and space complexity of an algorithm using big O notation.
- How to apply this to real examples.

Why is this important?

- To better understand the efficiency of algorithms.
- To talk about the scalability of programs in a sound way.
- To know how ‘fast’ or ‘slow’ a program is.

Measuring execution time

How to measure the execution time of a program?

We could, for example, count the number of operations performed by the program, but this approach depends heavily on:

- The input data.
- The compiler.
- Our computer.

In general, we just want to study how the complexity of our program *grows* as the input size n increases.

Measuring execution time

The goal: To find the number N of elementary operations needed to run the program as a function of the input size n.

Let's try to do this for a very simple task.

```
def two_sum(nums, target):
  for i in range(len(nums)):  # n=len(nums) times
    for j in range(len(nums)):  # n=len(nums) times
      if nums[i] + nums[j] == target: # constant 'c'
        return True
```

We can summarize the total amount of work done by `two_sum()` using the cost function $f(n) = cn^2$, where c is constant.

Measuring execution time

In the last slide, we arrived at the cost function

$$f(n) = cn^2,$$

which tells us that the number of operations grows quadratically as the input size n increases. But there are still a few things that we do not know, namely:

- What does the constant c mean?
- Is this constant significant at all?

Measuring execution time

Big O notation allows us to express the cost function in its ‘purest form’, showing us how the complexity of our program scales up as the input size grows. There are some simple rules to do this.

- Drop the constants.
- The costs of separate statements are added.
- The dominant term always wins.

After applying this to our example, we get $O(f(n)) = O(n^2)$.

Measuring execution time

Example: For the piece of code below, we have the total cost $f(n) = c_1 n^2 + c_2 n$, where c_1 and c_2 are constants.

```python
for i in range(len(nums)):  # n=len(nums) times
  for j in range(len(nums)):  # n=len(nums) times
    print(nums[i] + nums[j])  # constant 'c1'
for i in range(len(nums)):  # n=len(nums) times
  print(nums[i])  # constant 'c2'
```

After applying the set of rules we described earlier, we get:

$$O(f(n)) = O(c_1 n^2 + c_2 n) = O(n^2 + n) = O(n^2).$$

Measuring execution time

Observation: Saying that an algorithm runs in time $O(f(n))$ is just one way of saying that its time complexity is bounded above by $f(n)$, *i.e.* that it gets no worse than $f(n)$.

Let's examine the code from the previous slide.

```
for i in range(len(nums)):  # n=len(nums) times
  for j in range(len(nums)):  # n=len(nums) times
    print(nums[i] + nums[j])  # constant 'c1'
for i in range(len(nums)):  # n=len(nums) times
  print(nums[i])  # constant 'c2'
```

We saw that this algorithm is $O(n^2)$, so we can confidently say that it is also $O(n^3)$, $O(n^4)$, $O(n^5)$, $O(n^6)$, and so on.

It all comes down to the order of growth...

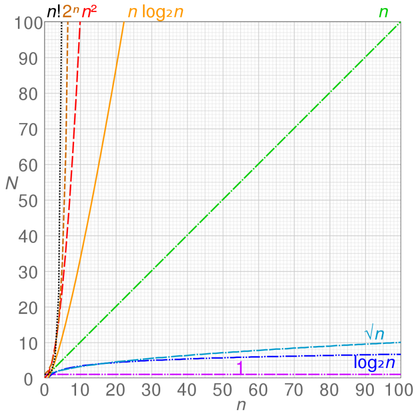

Figure 1: Graphs of number of operations, N vs input size, n for common complexities. Source: Wikipedia (Big O notation).

It all comes down to the order of growth...

Names for some common complexities.

- $O(1)$ – constant.
- $O(\log n)$ – logarithmic.
- $O(n)$ – linear.
- $O(n^2)$ – quadratic.
- $O(2^n)$ – exponential.

It all comes down to the order of growth...

Names for some common complexities.

- $O(1)$ – constant. 😎
- $O(\log n)$ – logarithmic. 😎
- $O(n)$ – linear. 😎
- $O(n^2)$ – quadratic.
- $O(2^n)$ – exponential.

How do we compare complexities?

Let f and g be cost functions. If $f(n) = O(g(n))$, then we can say f grows no faster than g.

- $f(n) = n,\ g(n) = n^2 \Rightarrow f(n) = O(g(n))$
- $f(n) = 5n,\ g(n) = 17n \Rightarrow f(n) = O(g(n))$ and $g(n) = O(f(n))$
- $f(n) = 10000000n,\ g(n) = n^2 + 2n + 1 \Rightarrow f(n) = O(g(n))$
- $f(n) = n,\ g(n) = \log n \Rightarrow g(n) = O(f(n))$

What is space complexity?

Space complexity is, shortly said, a measure of the amount of memory an algorithm needs for its execution.

How to measure space complexity?

To describe the space complexity of a program, we use similar rules to those we used for the time complexity.

```
def plus_one(nums):
    new_values = []
    for num in nums:
        new_values.append(num + 1)
    return new_values
```

The program needs to allocate memory for the same amount of elements in the input list. If $n =$`len(nums)`, then the space complexity would be $O(n)$.

Time to practice!