Big-O Notation

Vivek Singh

Information Systems Decision Sciences (ISDS)
MUMA College of Business
University of South Florida
Tampa, Florida

2018

Objectives

- To understand the importance of algorithm analysis in computer science.
- To understand the usage of "Big-O" to describe execution time.
- To understand the "Big-O" analysis of common operations on Python's in-built Data Structures.
- To analyze certain Python data implementations.
- To learn how to analyze and benchmark simple Python programs.

Introduction to Algorithm Analysis

- There are many ways to solve a problem in computer science. The question here is how does one approach compare against another.
- The underlying logic that an approach represents is what that makes it distinct from another in terms of memory and computational efficiency and even readability and reusability.
- An algorithm is a generic, step-by-step list of instructions for solving a problem.
- A software program is the algorithm encoded into a programming language of choice.
- Algorithm analysis is concerned with the amount of computing resources your implementation or solution uses.

Contd..

- The solution that is efficient with the limited resources available is often the best solution to a problem.
- Sometimes, we may exchange performance for productivity,but that is a subject for a broader discussion.
- There are two different ways to analyze performance of an algorithm
  - The first way is to consider the amount of *memory* an algorithm requires to solve the problem.
    - Some have some specific memory requirements, as we will see in certain sorting algorithms.
  - The second way is to analyze the amount of *time* an algorithm requires to execute. This measure is sometimes knows as the "execution time" or "running time" of an algorithm.

1

[1] An interesting read on performance vs productivity `http://www.cs.cornell.edu/~bindel/class/cs5220-s14/lectures/lec19.pdf`

Execution Time Example

Lets consider the case of sum of first n integers. We will compute the execution time of our solution using Python's time module.

```
import time
def sum_of_n(n):
    start = time.time()
    n_sum = 0
    for i in range(1, n+1):
        n_sum = n_sum + i
    end = time.time()
    return n_sum,end-start
```

- The above code takes in a positive integer value and computes the sum of all integers upto n starting from 1 .
- We are recording the start time before the computation starts and also the end time when its done.
- The time difference will give us the full execution time of

Execution Time Example

Lets run the program and see how it performs.

```
for i in range(5):
    print("Sum is %d required %10.7f seconds"
     % sum_of_n(10000))
```

Sum is 50005000 required 0.0005009 seconds
Sum is 50005000 required 0.0005014 seconds
Sum is 50005000 required 0.0005009 seconds
Sum is 50005000 required 0.0010028 seconds
Sum is 50005000 required 0.0005014 seconds
The results are fairly consistent for the first 10000 integers.

Execution Time Example

Lets compute the sum of the first 1000000 integers using the same program and see how that performs.

```
for i in range(5):
    print("Sum is %d required %10.7f seconds"
     %sum_of_n_2(1000000))
```

Sum is 500000500000 required 0.0782065 seconds
Sum is 500000500000 required 0.0646460 seconds
Sum is 500000500000 required 0.0611508 seconds
Sum is 500000500000 required 0.0601616 seconds
Sum is 500000500000 required 0.0656435 seconds
We can see the execution time has gone up 10 times or more than that as compared to the previous computation.

Intuitions and Conclusions

- Intuitively we can say that the performance of the program seems to reduce especially with larger integers.
- But that is only expected as there is more number of computations to be done.
- However, this is not uniform as we could see that the same program produces different results in different execution times on different machines, and different programming languages.
- Remember that we are only concerned with the performance of the algorithm and not the Python implementation of it.
- Therefore, execution time is not the correct parameter to analyze the performance of algorithms uniformly.
- *We need to characterize performance that is independent of the program or the machine being used.*

Big-O Notation

- To characterize an algorithm's performance in terms of execution time, independent of other factors such as language or machine's ability, it is important to quantify the number of operations or steps that the algorithm will require.
- When each step accounts for a basic unit of computation, then the execution time for an algorithm can be expressed as the number of steps required to solve the problem.
- But again, deciding on an appropriate basic unit of computation can be a complicated problem in itself and depends on the algorithm's implementation.
- In some problems, it is the number of assignments done, sometimes its the number of comparisons made between two values.

Big-O Notation

- For our problem, where we calculate the sum of n numbers, we have an initial assignment in n_sum = 0.
- It is followed by n number of assignments for adding every other element to the n_sum variable.
- Now, we denote this by a function, which we call T, where T(n) = 1 + n.
- The parameter n is called the size of the problem and we can read this is as "T(n) is the time it takes to solve a problem of size n, which is 1+n steps."
- Computer scientists prefer to take this analysis technique one step further. As problems become more complex, the exact number of operations is not that important as compared to the *most dominant part of the T(n) function.*
- The "order of magnitude" function describes the part of T(n) that increases the fastest as value of n increases.

Order of Magnitude

- Order of magnitude is often called Big-O notation (for "order") and written as O(f(n)).
- The representation provides a useful approximation to the actual number of steps in the computation.
- The function f(n) provides a simple representation of the *dominant* part of the original T(n).
- For example, for our sum of n integers problem, as n gets larger,the constant 1 becomes less significant and we could drop it and safely say that the execution time is O(n).
- Lets imagine another example takes exactly T(n) = $2n^2$ + 30n + 50 steps. But as n increases, the n^2 term gets larger and at some point it becomes the most important.
- When it becomes very large, we could perhaps ignore the coefficient 2 as well and say the function T(n) has an order of magnitude f(n) = n^2 or O(n^2).

Different Approaches to Performance Evaluation

- There are different measurements taken to analyze the performance of an algorithm in different cases.
- We usually characterize the performance in terms of *best case*, *worst case* and *average case* performances.
- There are situations in which an algorithm performs especially bad but this depends entirely on the data that it is working with.
- Going by that definition, for a different data set, the algorithm might perform exceptionally well.
- Most of the time, the performance lies in between the best and worst case and we call it the average case.

Evaluating Performance with Example

Lets consider this example below and see how to analyze the performance in terms of the number of assignment expressions executed. Although the code actually doesn't solve anything, it is a convenient example to learn how evaluation is done.

```
n= 2; a = 5; b = 6; c = 10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a * k + 45
    v = b * b
d = 33
```

Evaluating Performance with Example

- Initially we have 4 assignment statements which we will consider as constant '4' in our function.
- In the next block of code, we see there is a nested for loop which accounts for n^2 steps and since there are 3 assignment statements enclosed in it, they totally account for $3n^2$ assignments.
- That is followed by another for loop which has 2 assignment statements and therefore, 2n steps.
- At the end there is an assignment again and so it is 1 step and hence the final function is T(n) = 4 + $3n^2$ + 2n + 1
- But in the event, n gets larger, the $3n^2$ part will be dominating and hence, we can approximate this as O(n^2) algorithm.

Anagram Detection Example

Lets discuss Big-O Notation further with the Anagram example.

- A string is an anagram of another if the second string is simply a rearrangement of the first string.
- For example, School Master is a rearrangement of The Classroom. Likewise "Astronomer" is an anagram of "Moon Starrer".
- If the two strings are of the same length and are made up of a combination of the 26 alphabets, we could easily analyze them.
- Lets discuss two solutions to identifying an anagram and analyze their performances.
- Code link to be shared.

The Analysis of the Solution #1

- At first glance you we see straightforward assignments and hence we might be tempted to think this O(n).
- But the calls to the sort method comes with a price.
- We will see in later exercises that the sort methods are either O(nlog n) or O(n^2).
- And for larger n values, the sorting operation will dominate the performance.
- Therefore, this algorithm will have the same order of magnitude as that of the sorting process.

The Analysis of the Solution #2

- The solution has a number of iterations. But, unlike the first solution, none of them are nested.
- The first two iterations used to count the characters are both based on n.
- The third iteration that compares the two lists of counts, always takes 26 steps since there are 26 possible characters in the strings.
- Adding them up, it gives us T(n) = 2n + 26 steps. That is O(n).
- We have thus achieved a linear order of magnitude algorithm for solving this problem.
- Note - Although this solution achieved a better performance, it is important that we made some memory sacrifices to attain it.
- The solution could only work by using additional storage to keep the two lists of character counts.

Key Take Away

- In computer science, it is often left to the nature of the problem, the programmer and the final output that is expected of it, to make decisions on performance and space requirements.
- In the example that we just discussed, although the second solution appears to be the better one, it could create serious strain on the memory if the strings to be compared were large.
- Usually, the problems and data are quite complex and large and it is up to us find a balance between performance and resource constraints.

Summary

- We learned the importance of algorithm analysis in computer science.
- We learned the use of Big-O Notation to represent the computational effort that goes behind running an algorithm programmatically.
- We learned how different algorithms could be compared against each other using the Big-O Notation.
- We have understood how to make choices when we are presented with a dilemma of choosing betw een space requirements and execution time requirements.