Runtime Analysis

Erich Wellinger

08/09/2016

Goals

- Understand why runtime analysis is important.
- Understand what Big Oh Notation is.
  - Common runtimes from fast to slow.
- Big Oh execution time of common list and dictionary operations in Python.
- Example questions.

Why do we care?

Beyond the fact that we might be asked an interview question about how fast our code runs (hint: the answer is not an actual time), there is real value in being able to compare two algorithms (or functions, scripts, programs, & c.) to see which one is better. While the aesthetics of a solution might be preferable to another simply because of readability, we also would like a means of comparing the two based solely on efficiency.

Consider the following example you are all too familiar with: Your boss comes to you with an urgent matter; all the text in your company's database has been downloaded, tokenized, and it's your job to pick out all the anagrams that exist. Given that we are experienced programmers, several solutions to this pickle immediately come to mind. Maybe this is our first attempt:

```python
def get_anagrams(lst):
    result = []
    for wrd1 in lst:
        for wrd2 in lst:
            if wrd1 != wrd2 and sorted(wrd1) == sorted(wrd2):
                result.append(wrd1)
                break
    return result
```

Now maybe this is working pretty good but your boss would still like it to run faster. One solution to this might be throwing your code onto a more powerful machine. While this solution might minimize the execution time of your code, did it make your code any more efficient? Our code never actually changed and so it's clear this scenario is no more efficient despite getting an answer faster.

Runtime analysis allows us to evaluate how efficiently code runs regardless of the system that is running it or even the programming language it's written in.

Big Oh Notation

Big Oh notation gives us the means of quantifying how efficient a piece of code is. We do this by breaking the code into steps, each of which represent a basic until of computation. If we had the following piece of code:

```
def worthless_func(n):
    for _ in range(n):
        a = 12
    return 42
```

We would say that this code was $O(n)$ (pronounced "big oh of n") because for every element in n we are assigning 12 to a. As n increases, the time it takes the function to run will scale linearly. Big Oh notation serves to provide us an approximation of the actual number of steps in the computation. The following lists some common runtimes from fast to slow:

$f(n)$	Name
1	Constant
$log\ n$	Logarithmic
n	Linear
$n\ log\ n$	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

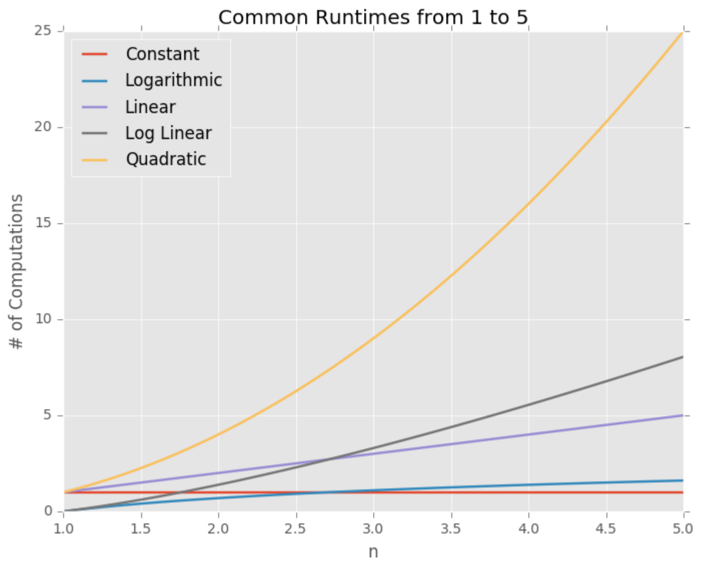

Figure 1: Breakdown of common runtimes from 1 to 5

An important factor to note is that we are also only interested in the worst case scenario. If we have to check the membership of an element in a list of length n, the runtime is $O(n)$ even if we find the element in the first position. That way we can take into account how well our function will run

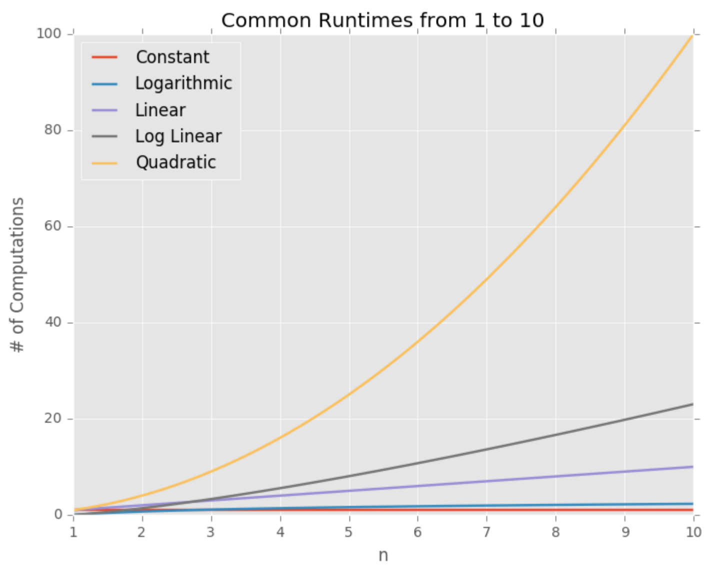

Figure 2: Breakdown of common runtimes from 1 to 10

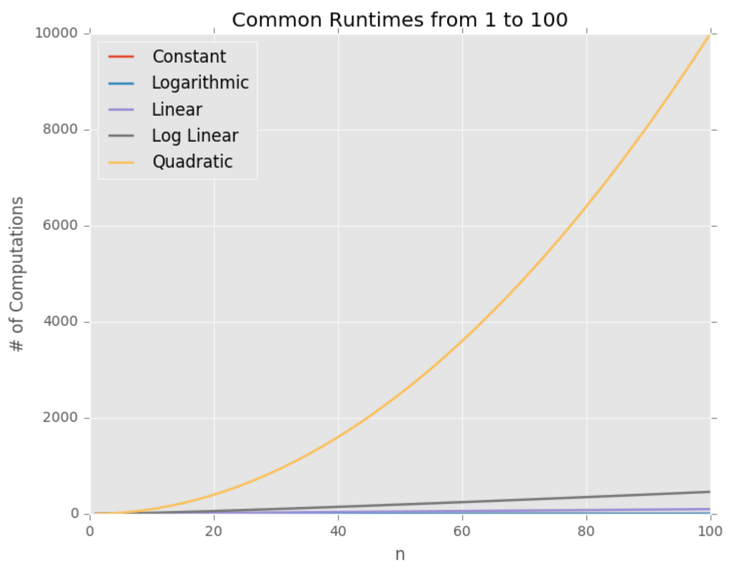

Figure 3: Breakdown of common runtimes from 1 to 100

regardless of the circumstances (performance can be broken down further into best case, worst case, and average case).

Furthermore, we also don't care about constant factors. We never say $O(n+2)$ because the added two steps will be insignificant given a large enough n. Also, if you have two parts to your function, one which takes $O(n)$ and one which takes $O(n^2)$, we say the total runtime is $O(n^2)$ because this is the larger factor.

Does this *really* matter?

So does this runtime really matter? Is n really that much faster than n^2? Let's assume that each computation step takes 1/100th of a second to perform. Let's see how two algorithms compare as we scale up n:

n	$O(n)$ algorithm	$O(n^2)$ algorithm
100	1 sec	1 sec
1,000	10 sec	100 sec
10,000	100 sec	167 min
100,000	17 min	11 days

I don't know about you, but the difference between a speedy lunch and two weeks is pretty significant.

Anagrams

Let's look back to our solution to the anagrams problem and pick apart how efficient it was.

```
1   def get_anagrams(lst):
2       result = []
3       for wrd1 in lst:
4           for wrd2 in lst:
5               if wrd1 != wrd2 and sorted(wrd1) == sorted(wrd2):
6                   result.append(wrd1)
7                   break
8       return result
```

- **Line 5**: Sorting is generally $O(n\ log\ n)$. Since we are actually sorting the characters in a word, we would say that the runtime is $O(k\ log\ k)$ where k denotes the number of characters in the word. While we are sorting two separate words, we would drop the constant and represent this while line as just $O(k\ log\ k)$
- **Lines 3-4**: The for loop on line 3 is $O(n)$, but because we are considering every word for each iteration through, we need to multiply that by n, yielding $O(n^2)$

All together this gives us a runtime of roughly $O(n^2 * k\ log\ k)$

Try coding up some other solutions and analyze the associated Big-Oh efficiency! Once you have one or two under your belt, check out this breakdown of other solutions we might take to the anagrams problem (zipfian/interview-prep/interview_questions/coding/runtime_analysis.md)

Standard operations and their complexity

Lists

Operation	Big-O Efficiency
append	$O(1)$
index []	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
contains (in)	$O(n)$
sort	$O(n\ log\ n)$

Dictionarys

Operation	Big-O Efficiency
insert	$O(1)$
del operator	$O(1)$
lookup by key	$O(1)$
lookup by value	$O(n)$

Some Problems[1]

1. What is the worst-case running time using Big Oh notation?

```
def mystery(n):
    r = 0
    for i in range(1, n):
        for j in range(1, i):
            for k in range(j, i+j):
                r += 1
    return r
```

Answer: http://www.algorithm.cs.sunysb.edu/algowiki/index.php/TADM2E_2.2

2. True or False: is $2^{n+1} = O(2^n)$

Answer: True.

$$
\begin{aligned}
2^{n+1} &= 2 * 2^n \\
2^{n+1} &= C * 2^n \\
2^{n+1} &\approx 2^n \\
2^{n+1} &= O(2^n)
\end{aligned}
$$

http://www.algorithm.cs.sunysb.edu/algowiki/index.php/TADM2E_2.7

[1]Many questions and solutions taken from The Algorithm Design Manual by Steven S. Skiena

3. You are given a set S of n numbers. You must pick a subset S' of k numbers from S such that the probability of each element of S occurring in S' is equal (i.e. each is selected with the probability k/n). You only make one pass over the numbers and n is unknown.

Answer:

```
def random_sampling(stream, k):
    result = []
    elements_seen = 0
    for item in stream:
        elements_seen += 1
        if elements_seen <= k:
            result.append(item)
        elif random.random() <= float(k)/(elements_seen + 1):
            result[random.choice(range(k))] = item
    return result
```

4. You have a 100-story building and a couple of marbles. You must identify the lowest floor for which a marble will break if you drop it from this floor. How fast can you find this floor if you are given an infinite supply of marbles? What if you have only two marbles?

5. The Grinch is given the job of partitioning $2n$ players into two teams of n players each. Each player has a numerical rating that measures how good he/she is at the game. He seeks to divide the players as unfairly as possible, so as to create the biggest possible talent imbalance between team A and team B. How would you do this? Show how the Grinch can do the job in $O(n\ log\ n)$ time.

Answer: http://www.algorithm.cs.sunysb.edu/algowiki/index.php/TADM2E_4.1

6. Let S be an *unsorted* array of n integers. Give an algorithm that finds the pair $x, y \in S$ that *maximizes* $|x - y|$. Your algorithm should run in $O(n)$ worst-case time.
    7. How does assuming a *sorted* array affect your algorithm and the associated runtime?

Answer: See a) and b) http://www.algorithm.cs.sunysb.edu/algowiki/index.php/TADM2E_4.2

8. You are given a pile of thousands of telephone bills and thousands of checks sent in to pay the bills. Find out who did not pay. What is the worst-case complexity of your solution?

9. Describe advantages and disadvantages of the most popular sorting algorithms.

Answer:

- Merge Sort
  - Advantages: Suitable for linked lists and external sort.
  - Disadvantages: Needs extra buffer holding the merged data.
- Insertion/Selection Sort
  - Advantages: Easy to implement.
  - Disadvantages: Too slow and becomes impractical when data is huge.
- Heap Sort
  - Advantages: Don't need recursion. Suitable for large data.
  - Disadvantages: Usually slower than merge sort and quick sort.
- Quick Sort
  - Advantages: Practical fastest.
  - Disadvantages: Recursive, worst case is slow.

10. Given a search string of three words, find the smallest snippet of the document that contains all three of the search words—i.e., the snippet with smallest number of words in it. You are

given the index positions where these words occur in the document, such as word1: (1, 4, 5), word2: (3, 9, 10), and word3: (2, 6, 15). Each of the lists are in sorted order, as above.

Answer: http://www.algorithm.cs.sunysb.edu/algowiki/index.php/TADM2E_4.45

11. You are given 12 coins. One of them is heavier or lighter than the rest. Identify this coin in just three weighings. Note - weighings are with a balance, where you can have a greater than, equal to, or less than result. You can't do this with a digital scale.

Answer: http://www.algorithm.cs.sunysb.edu/algowiki/index.php/TADM2E_4.46