Lecture 2: Performance Analysis

built on 2019/04/23 at 15:57:24

Consider the following function, written in Python:

```python
from typing import List
from copy import copy

def foobar(numbers: List[int]) -> List[int]:
    numbers = copy(list(numbers))
    for j in range(1, len(numbers)):
        key = numbers[j]
        i = j-1
        while i >= 0 and numbers[i] > key:
            numbers[i+1] = numbers[i]
            i = i-1
        numbers[i+1] = key
    return numbers
```

This is a typical algorithm implementation, and there are many questions that we generally ask about it:

- How fast is this algorithm? That is, how long does the algorithm take to finish running a given task?
- How much space does it use?

How do we study such questions?

- Experimental analysis
- Mathematical analysis

1 Experimental Studies

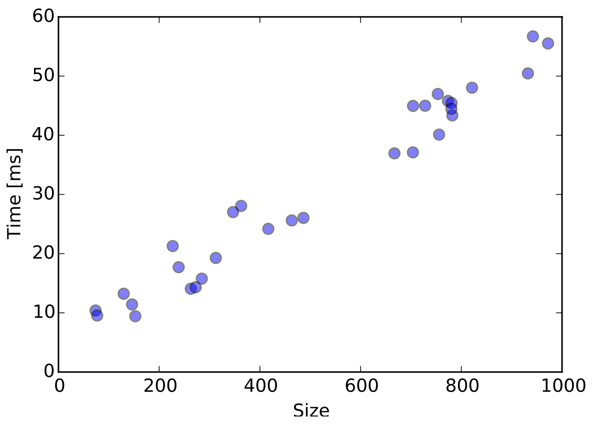
If we have an implementation of an algorithm, we can study its running time behavior by running it on multiple test inputs, recording how long each run takes.

This is quite easy to do with a stopwatch:

```
start = time_now();
// run your code
stop = time_now();
time_used = stop - start;
```

The time measured in this way is known as the *wall clock* time because it's the time that has passed in real life while the program does the work. This gives a reasonable sense of a program's performance, though we could see interferences from other things running on the same machine.

Often, we study this kind of things because we want to be able to predict the behavior of such a program in the future, on input that we haven't seen before. Therefore, we are interested in how the running time varies as the size and the structure of the input changes. To do this, just like in science experiments, we perform independent runs on many test inputs of various sizes and configurations. Plotting it on a time vs. input size



scale (like shown on the right) can provide us with a good sense of the program's overall behavior.

Play Time:

```
# ipython3 -i isort.py
                                              Table (range...)
%timeit _ = foobar(range(500))                |
%timeit _ = foobar(range(1000))               |
%timeit _ = foobar(range(2000))               |
%timeit _ = foobar(range(4000))               |

%timeit _ = foobar(range(500)[::-1])          Table (reversed range ...)

import numpy as np
%timeit _ = foobar(np.random.rand(500))
```

1.1 That's It?

This looks fine and dandy, especially when you have a properly implemented piece of software that needs fine-tuning. But there are a few major limitations to always conducting experiments:

- Experiments can only be done on a limited set of test inputs and may not reflect the behavior on all possible inputs.
- You need an implementation to run the experiment with. (This can be costly especially if all we want is to compare your options before we implement a solution.)

2 Beyond Experimental Analysis

To move beyond empirical analysis, we want an approach that

- evaluate the relative efficiency of algorithms that are independent of the hardware/software platforms.
- can be applied by looking at just the high-level description, without the need to fully implement it.
- take into account all possible inputs.

How can we satisfy all these requirements? As simple as this may sound, we will simply *count the number of primitive operations*. Quite surprisingly, on modern computer systems, the cost of primitive operations only vary a little (by constant factors) across hardware/software platforms—if we define the primitive operations well.

What're primitive operations? Things that can be done simply, in a constant amount of execution time. These include:

- Assigning an idenfier to an object
- Performing arithemtic operations on normal-sized numbers
- Accessing an element in an array (think Java array or Python list) by index
- Returning from/calling a function (excluding the work performed within the function)

Therefore, we could, in theory, come up with a function

$f(I) =$ the number of primitive operations performed on input I.

But that's cumbersome to talk about, so we want something more handy.

2.1 Function of Input Size

We settle to use $f(n)$ to characterize the number of primitive operations performed when the input has size n. This means regardless of the actual input, as long as the input has size n, we want it to behave more or less like $f(n)$. But then, two questions arise:

1. *How do we measure the input's size?* **Ans:** Usually, there's a natural measure for the problem. For example, if the input is a list, the number of elements in that list.
2. Algorithms can take drastically different amounts of time on different inputs even of the same size. *How do we handle that?* **Ans:** Worst-case behavior.

So why should we care about worst-case?

- Average case: difficult to characterize
- Worst case: it works well no matter what. Optimize for a stronger requirement :)

3 Asymptotic Analysis

Okay, where're we so far? We wanted to count the number of (primitive) operations an algorithm performs on the worst possible input of a certain size n.

There are a few problems with this: First, we're lazy—so finding the exact count is a nonstarter. Second, the exact count is generally useless because when mapped down the machine level, specifics of the platform matter more than off by 1 or so. Third, we want something clean to talk about.

Goal: first order approximation that has good predictive power.

How? The Big-Oh notation.

3.1 The Big-Oh notation

Let $f(n)$ and $g(n)$ be functions mapping positive reals to positive real numbers (i.e., $f, g : \mathbb{R}_+ \to \mathbb{R}_+$). Then:

We say that $f(n)$ is $O(g(n))$ (**read:** f is *big-O* of g) if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

Indeed, an alternate definition is

We say $f(n)$ is $O(g(n))$ if there's a real constant $c > 0$ and an integer constant $n_0 \geqslant 1$ such that $f(n) \leqslant c \cdot g(n)$ for all $n \geqslant n_0$.

The formal definition can be scary but the gist of it is simple:

As n grows very large, $f(n)$ is less than or equal to another function $g(n)$ up to a constant factor.

We'll leave the formal study of big-Oh (and their friends) to Discrete Math and revisit it later after you're more comfortable with it. For now, I want to offer some intuitive ways of thinking about big-Oh.

1. The Big-Oh of a function is the dominant term.
   - $10n^7 + n^2 + 900n\log n + 10^{100} \in O(n^7)$ as one can verify
   $$\lim_{n\to\infty} \frac{10n^7 + n^2 + 900n\log n + 10^{100}}{n^7} = 10 < \infty.$$
   - $n^2 + n\log n \in O(n^2)$
   - $2n + n\log n \in O(n\log n)$
2. They add and multiply in natural ways:
   - *Ex I:* If you perform n tasks, each taking $O(n)$ time, the total time spent is $O(n^2)$. More of this on Assignment 1.
   - *Ex II:* Say you perform two tasks: the first task takes $O(n^2)$ time and the second task takes $O(n\log n)$ time. Altogether, the time you spend equals the sum of the two tasks. It should be $O(n^2 + n\log n) = O(n^2)$ because that term dominates.

3. Fashion issue: characterize functions in simplest terms. By definition, $f(n) = 10n^3 + 2$ is $O(n^4)$, even $O(n^4)$; however, it's customary to write $O(n^3)$ because it's simplest and closest expression.
   Big-Oh is like this: How long does it take you to drive to MUIC? The exact answer: 1 hour, 3 minutes, 2 seconds, 15 milliseconds, and 5 nanoseconds. Big-Oh gives us a way to say "no more than 2 hours" ignoring the nitty-gritty detail. Now while it's also correct to say "no more than 12 hours" — you'd rather say "2" than "12" because it's a better approximation and as succinct.

3.2 Big-Theta Notation

In a strict sense, big-O says one function is "upperbounded" by another function. Although it's fashionable to give a tight upper bound, tightness is not a requirement. Mathematically, there is a related notion that precisely says a function f behaves in essentially the same way as g.

We say that $f(n)$ is $\Theta(g(n))$ (**read:** f is *theta* of g) if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

for some constant $c > 0$.

4 Examples

Let's warm up with two simple algorithms:

Consider the first algorithm:

```
int mooFest(int x) {
    return x + 2;
}
```

We see that if we need this function any number x, the only thing it will do is compute $x + 2$ and return that value. This requires a few (still constant) number of primitive operations. So we can make the following claim:

Claim 1: On input x a number, `moofest` takes at most c time for some constant c—or in Big-O notation $O(1)$ time.

To justify it, we argue that both computing $x + 2$ and returning that value can be done in a constant number of primitive operations, so that is clearly bounded by a constant.

As a second example, consider the following algorithm (once in Python/once in Java):

```
def foofest(lst):
    maxSoFar = lst[0]          # constant time - just primitive ops
    for i in range(1, len(lst)):     # repeat n - 1 times
        if maxSoFar < lst[i]:        # just a comparision [prim op] (inside loop)
            maxSoFar = lst[i]        # just assignment [prim op]
    return maxSoFar            # just a return
```

```
int fooFest(int[] lst) {
    int maxSoFar = lst[0];      // constant time - just primitive ops
    for (int i=1;i<lst.length;++i) {    // rep n - 1 times
        if (maxSoFar < lst[i])          // compare (prim op)
            maxSoFar = lst[i];          // assignment (prim op)
    }
    return maxSoFar             // just a return
}
```

To count steps, we can break down the time into:

- Work done on Line 2, which is constant $O(1)$
- Work done on Lines 3–5, which we'll analyze
- Work done "return", which is constant $O(1)$

Hence, the total time will be $O(1)$ steps plus however long the work done on Lines 3–5 takes. ...which brings us to the question: how costly are Lines 3–5?

The new challenge here is the presence of a loop. Let's write n to be the length of `lst` for notational convenience. Now as we know, a loop just says repeat what is inside it a number of times. We know that the inside of this loop does the same amount of work (more or less) in every iteration.

Iter. Count	Steps Taken
$i = 1$	some constant c
$i = 2$	some constant c
$\vdots$	
$i = n - 1$	some constant c

Therefore, Lines 3 – 5 take $c(n - 1)$ steps in all, which is $O(n)$. Overall, this is an $O(n)$ algorithm.

5 A More Involved Example

We now turn to the sequence uniqueness problem. In this problem, we want to find out whether all elements of a given list are distinct from each other. That is, we will write a function that takes a list and return True if there are no duplicates in the input (and False otherwise).

Example:

- `[1, 3, 2, 5, 1]` is *not* unique.
- `[1, 4, 3, 5, 7]` is unique.

Hence, we can imagine writing the following piece of code: we try all possible pairs and see if they have the same value:

```
function unique1(seq):
  let n = len(seq)
  for i = 0, 1, ..., n - 1:
    for j = i+1, ..., n - 1:
       if seq[i] == seq[j]:
           return False
  return True
```

As before, we'll quantify the time complexity of this algorithm by counting the number of primitive operations.

For any given i and j, what's the running time of the code `if seq[i] == seq[j]: return False`? These can be done using a constant number of primitive operations, so $O(1)$ time—or some constant k.

For notational convenience, we'll let $n = \texttt{len}(\texttt{seq})$.

Now consider that there are two nested loops. The outer loop goes over indicies $i = 0, 1, 2, \ldots, n - 1$. For each i, the inner loop goes over $j = i + 1, i + 2, ..., n - 1$. This means that for each i, there are $n - i$ values of j. And inside the inner loop, it does $O(1)$ work. More precisely, say that it is c steps for some constant $c > 0$.

Hence, for each i, we're talking about $c(n - 1 - i)$ operations. Across all i and together with the other cost, we have

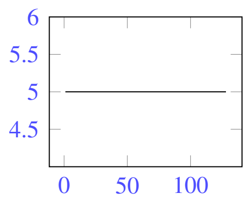
$$k + c\Big[(n-1) + (n-2) + ... + 2 + 1\Big] = k + c \cdot n(n-1)/2 \in O(n^2).$$

6 Digression: Seven Functions

We will review seven basic functions that will keep coming up in this class.

1. *The Constant Function:* This is the simplest function, the function $f(n) = c$ for some *fixed* constant c. For example, $c = 13, c = 17, c = 10^{10}$. What happens here is, *regardless of the input size n, the function remains at that constant c.*

   Let's plot it!
   The constant function is useful in algorithm analysis as it captures the running time of basic/primitive operations as well as a few other cheap operations.

2. *The Linear Function:* This is the function $f(n) = n$. Given an input of size n, this function registers the value n itself. As you can imagine, it comes up a lot in algorithm analysis. If the cost of touching each element is constant, the time to go over a sequence of n elements is n.
3. *The Logarithm Function:* This is the function $f(n) = \log_b n$ for some constant $b > 1$, where the meaning of the log function is given by

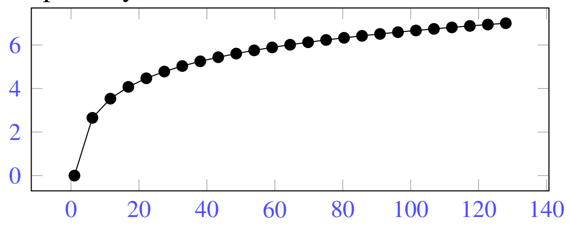
   $$x = \log_b n \iff b^x = n.$$

   In computer science, the most common base is 2, so we tend to write $\log n$ to mean $\log_2 n$.
   IMPORTANT: If this formality makes your head spin, the thing to remember is, for practical purposes of algorithm analysis, $\log n$ (or $\log_2 n$) is this Python function:

   ```
   def log(x):
     ans = 0
     while x > 1:
         ans += 1
         x /= 2
     return ans
   ```

   It tells you *how many times you can divide by* 2 *before x is at most* 1.

   Graphically, this is what it looks like:

   MATHY PEOPLE: Let's remind yourself the logrithm rules through this quick exercise:

   - $\log(4n) = \log 4 + \log n = 2 + \log n$
   - $\log(n/2) = \log n - \log 2 = \log n - 1$
   - $\log(n^2) = 2\log n; \log(n^{100}) = 100\log n$
   - $\log(2^n) = n$
   - $2^{\log n} = n$

4. *The N-Log-N Function:* That is, $f(n) = n\log n$. This is probably no good reason why this function should occur in nature, but it does come up often as the running time of many algorithms.
   The take away is it is slightly bigger than linear but not by much.
5. *The Quadratic Function:* The quadratic function is $f(n) = n^2$ (read: n squared).
   It appears frequently in programs with nested loops. For example, there are two loops:

   ```
   for i = 0, 1, 2, ..., n - 1:
     for j in 0, 1, 2, ..., n - 1:
       # do some constant work
   ```

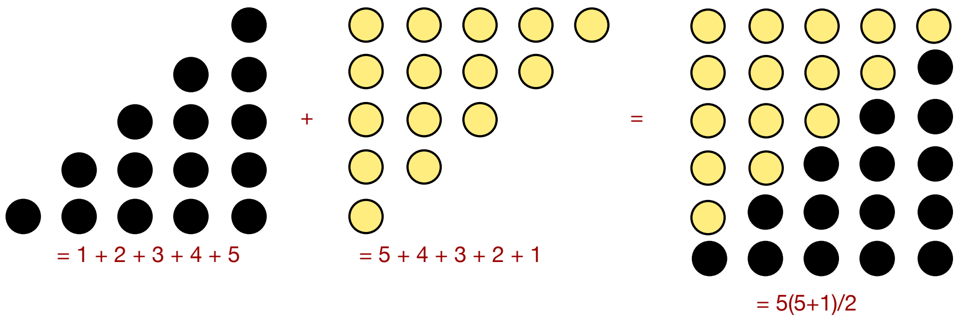
It also appears in the context of nested loops where:

```
for i in 0, 1, 2, ... , n - 1:
  for j in 0, 1, ..., i - 1:
    # do some constant work
```

In the first iteration, the inner loop does 1 unit of work. Then, 2 units, then 3 units, giving the total work:

$$1 + 2 + ... + n = \frac{n(n+1)}{2}$$

For a graphical derivation:



6. *The Cubic Function:* This is $f(n) = n^3$, which sometimes occurs.
7. *The Exponential Function:* These are $f(n) = b^n$ for a positive constant b (called the base). You should review the exponent rules.

I will state and prove one identity that will come up again and again.

Geometric Sum:

$$1 + 2 + 4 + ... + 2^n = 2^{n+1} - 1.$$

Here's an informal proof. Consider that if

$$N = (\underbrace{111111...1}_{n \text{ of } 1's})_2.$$

Then if we look at $N + 1$, we will see that

$$\begin{array}{cccccccc} & 1 & 1 & 1 & 1 & \dots & 1 & 1\,+ \\ & & & & & & & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ \hline\hline \end{array}$$

which is that $N + 1 = 2^{n+1}$. On the other side, it is not hard to see that N is $(111111\dots1)_2 = 2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^n$, so

$$\left(2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^n\right) + 1 = 2^{n+1} \implies 1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1.$$

You'll see a more formal way of proving this in Discrete Math.

6.1 Growth-wise: Can we order them?

As conventional wisdom would have it:

constant < log < n < n-log-n < quadratic < cubic < exponential

Just to see how that works:

N	Constant	Log	N	N-Log-N	Quadratic	Cubic	2^n
$N = 8$	1	3	8	24	64	512	256
$N = 16$	1	4	16	64	256	4096	65536
$N = 32$	1	5	32	160	1024	32768	4B
$N = 1024$	1	10	1024	10240	1M	1G	309 digits

7 Note on Formality

Remember that

A function $f(n)$ is $O(g(n))$—f is big-O of g—if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

Technically speaking, the big-O of a function, as in $O(n)$, is a set. In particular, for any function g,

$$O(g(n)) = \left\{ f(n) \;\middle|\; \lim_{n\to\infty} \tfrac{f(n)}{g(n)} < \infty \right\}.$$

Therefore, the proper of way is to say $f(n) \in O(g(n))$, but most people don't do that. They write $f(n) = O(g(n))$. Abusing notation, we also use Big-O in expressions such as $n + O(\log n)$. The presence of the Big-O in an expression such as $h(n) + O(g(n))$ simply means that expression is at most $h(n) + k \cdot g(n)$ for some constant $k > 0$. Hence, the expression $n + O(\log n)$ just means $n + k \log n$ for some $k > 0$.