Algorithm Analysis

June 22, 2017

1 Introduction: Two algorithms for a simple problem

Your math teacher enters the class, and asks you to find a way to compute $1 + 2 + \cdots + N$, given the natural number N.
Ha, you write a simple program in your favourite language which runs a loop from 1 to N, and adds each value to a sum variable.
Feeling rather good about yourself, you decide to take a look over at your partner's work, and they've simply written the formula

$$\frac{N \times (N + 1)}{2}$$

Ah, well, at least your solution is still correct, maybe not as fancy as your partner's.
In a dramatic twist of events, the teacher decides to actually run your program on the computer, with $N = 10^{10}$.
You watch in pain as your program gets stuck for almost a minute before finally throwing out 50000000005000000000 ; Your partner's formula converted to code outputs the same value instantly.

What happened there?
Well, so far, we've always thought that computers are *fast*. Our piddly human brains are so far behind that not only can we not keep up with them, we can't even understand how far behind we are.
Except the last bit is false. We have a rough idea as to how fast *fast* is.
Your modern desktop PC can perform $\sim 10^8$ operations per second. While that is an insanely large number, it is *not* infinite.
So when you ran your program which does N additions to compute $1+2+\cdots+N$ with $N = 10^{10}$, you asked the computer to do 10^{10} operations, which, by the estimate above, should take ~ 100 seconds. Whereas your partner's idea does this in a single addition, followed by a multiplication, followed by a division by 2.
By looking at your code, it should be clear that when it is given N as input, it takes N additions to compute the required sum. And looking at your partner's

formular, it should be clear that given N as input, it takes 3 operations to compute the required sum. This is the fundamental idea of algorithm analysis. You *count* the number of steps your program takes to execute, and try to figure out how it grows as your input becomes larger.

2 A more complicated example

Let's look at a very popular traditional question.

You are given an array A of size N, and then asked Q questions, each of which gives two indices, i and j, such that $0 \leq i \leq j < N$.
For each question, you have to find the sum $A_i + A_{i+1} + \cdots + A_j$.
Take a minute to think about your solution.
.
.
.

2.1 The easy solution

Ah, we can just run a loop from i to j, and within the loop, add the corresponding element of A to a `sum` variable.
Well that was easy.

2.1.1 A small problem

What if we have a lot of questions, and a very big array? Say $N = 10^5$, and $Q = 10^5$. (you may wonder why anyone is so interested in a dumb array, but such questions are beyond the scope of this write-up)
Let's try to count the number of steps our solution would take.
For convenience, let's denote the $k^{\texttt{th}}$ query by (i_k, j_k). Then, to answer the first query, our solution takes $j_1 - i_1 + 1$ steps, to answer the second query, $j_2 - i_2 + 1$ steps, and so on.
To answer all the queries, our solution takes $(j_1 - i_1 + 1) + (j_2 - i_2 + 1) + \cdots + (j_Q - i_Q + 1)$ steps, which is $\sum_{k=1}^{Q}(j_k - i_k + 1)$ steps.
Well this is nice, but what information does it give us? Is our solution too slow? Is it ok? Is it too fast? (this doesn't happen very often $\ddot\smile$)
Of course, our user could be very benevolent, and always give us questions where $i = j$, in which case our solution will run in $\sum_{k=1}^{Q} 1 = Q$ steps (which is pretty great, right?).
On the other hand, our user could be our worst nightmare, and always ask questions where $i = 1$, and $j = N$, in which case our solution will run in $\sum_{k=1}^{Q} N = Q \times N$ steps, which will mean 10^{10} operations $\ddot\frown$
In general, the safe assumption to make is that the worst thing that could happen is going to happen (this is also true of life $\ddot\frown$).

So, giving up our exact formula for the number of steps, $\sum_{k=1}^{Q}(j_k - i_k + 1)$ and trading it in for an *upper bound*, $Q \times N$, actually gives us much more useful information (here the takeaway is that our algorithm is too slow).
This is an important principle of *algorithm analysis*; we dump the exact formulae, and try to get some *idea* of how the algorithm performs as the input size increases.

3 Big-O notation

Often, as in the solution in sub-section 2.1, we replace the exact formula for the number of steps an algorithm takes, and replace it by an upper bound.
So instead of $j - i + 1$, where j and i are part of questions, we instead note that $j - i + 1 \leq N$, and hence N is a suitable replacement.
This is also written as $j - i + 1 = O(N)$, and read as $j - i + 1$ is Big-O of N.
Now it would be rather useless to define a new symbol just to say that $j - i + 1 \leq N$; so we ask the following question.

Do we really care about constant factors? Is there really a huge difference in the time it takes a computer to take $2 \times N$ steps, as opposed to N, steps?
(ye gads, this is double that)
It turns out that we don't really care; what we are worried about is how the number of steps grows as the input becomes larger and larger. So even though $2N > N$ for all $N > 1$, we note that they are always within a constant factor of 2 of each other, and hence are essentially the same for us.
So we allow $2N = O(N)$ (and also $3N = O(N)$, and so on). Note that 2 and 3 are both constants, which do not change with the "input".
We cannot have $QN = O(N)$, because Q is a value that can increase as input becomes larger, and is hence not a *constant.*

Well there we have it; if *something* is $<= N$, or $<= k \times N$, where k is some constant, we say that that *something* is $O(N)$.
Why should we restrict ourselves to N? What about N^2, $N \log N$, and all of our polynomial friends?

Definition 1 (Big-O notation)**.** *We say that an expression (for example a summation, or a function) is $O(f)$, if the expression is $\leq k \times f(N)$.* [1]

Here, f can be any function, like $f(N) = N^2$, or some such.

[1] there is an additional technicality we are ignoring, feel free to read the correct definition on the Internet, if you are comfortable with all of this