Analysis of Algorithms

Henry Z. Lo

May 29, 2014

1 Motivation

How scalable are algorithms? This is answered by seeing how the amount of resources consumed f grows as a function of input size n. In general, programmers should consider both time and space - in this course, we focus almost exclusively on time complexity only.

We lay down some ground rules for analyzing time complexity:

1. We only consider the worst case of an algorithm.
2. We only consider how $f(n)$, the time spent, grows in the long run. Thus, we can ignore constant overhead, e.g. we do not differentiate between n and $n + 300$.
3. There are distinct classes of growth rates. We do not consider multiples of a given growth rate to be distinct, e.g. we do not differentiate between n and $2n$.

The last two rules naturally lead to Big O notation, which we describe in Section 3. First, let's analyze some algorithms.

2 Calculating $f(n)$

2.1 Why we ignore multipliers

To count computations, we first need to define what counts as a computation. Consider the following piece of code:

```
for (int i=0; i<n; i++) {
  function_a(i, i);
}
```

Is `function_a(i, i)` one computation? What if `function_a` was defined as follows?

```
function_a(b, c) {
  function_b(b);
  function_c(c);
}
```

Here we see `function_a` is actually two functions - should it count as two computations then? If we count it as one, then the loop above is n computations; if we count it as two, then the loop is $2n$.

Of course, `function_b` and `function_c` could also be composed of other functions. This loop really consists of xn for some unknown number x. This is why we can safely disregard multiples such as x, as long as x itself doesn't depend on n.

2.2 Examples

For collections such as lists, stacks and queues, n usually refers to the number of elements. For things like graphs, we often need to consider both the number of edges and the number of nodes. We restrict our discussion here to counting one object.

You may have learned the general rule of thumb of multiplying the number of loops to get the runtime. For example, the following block of code consists of n^2 computations, because there are two nested loops:

```
for (int i=0; i<n; i++) {
  for (int j=0; j<n; j++) {
    sum += i+j;
  }
}
```

Be careful when the number of iterations through the loop is not n. For example, the following block consists of n^3 computations, even though there are only two nested loops:

```
for (int i=0; i<n*n; i++) {
  for (int j=0; j<n; j++) {
    sum += i+j;
  }
}
```

It is very common for the inner loop to iterate based on the index of the outer loop, as in the next problem.

Problem 1. *How many computations are in the following piece of code?*

```
for (int i=0; i<n; i++) {
  for (int j=0; j<i; j++) {
    sum += i+j;
  }
}
```

	$i=0$	1	2	3	4	5
$j=0$		1	2	4	7	11
1			3	5	8	12
2				6	9	13
3					10	14
4						15

In this case, at $i = 0$, we have 0 computations; at $i = 1$, we have 1, at $i = 2$, we have 2, and so on. Succinctly, we have $\sum_{x=0}^{n-1} x$. We can visualize the computations in the table above. Numbers indicate which computation is taking place with the current i and j.

Notice that the number of computations is the area of the triangle, which is half the area of the rectangle. The size of the rectangle is $n * (n-1)$. In this example, we $x = 6$, and so we have $6 * 5/2 = 15$ computations.

3 The Big O Function

We can use Big O notation to simplify discussion.

3.1 Definition

For a given algorithm, let $f(n)$ be the number of computations required for an input of size n. Then we say $f(n) = O(g(n))$ if:

$$0 \leq \lim_{n\to+\infty} \frac{f(n)}{g(n)} < +\infty$$

As an example, consider the solution to Problem 1, $n * (n-1)/2$. We can show that this is $O(n^2)$:

$$\begin{aligned}
&\lim_{n\to+\infty} \frac{n*(n-1)/2}{n^2} \\
&= \lim_{n\to+\infty} \frac{n-1}{2n} \\
&= \lim_{n\to+\infty} \frac{1-1/n}{2} \\
&= \frac{1}{2}
\end{aligned}$$

Technically, we can say that $n*(n-1)/2 = O(n^3)$ also, or any higher growth rate for that matter. In other words, if we treat it as a set, $O(n^3)$ contains $O(n^2)$:

$$O(\log_n) \subset O(n) \subset O(n^2) \subset \ldots \subset O(2^n)$$

But saying everything is $O(n^n)$ is not really informative, so please don't say bubblesort is $O(2^n)$, even though it is technically true. For the rest of this class, please give the smallest class that the runtime belongs to, e.g. $O(n^2)$.

We can say that the complexity classes of f and g are exactly the same if:

$$0 < \lim_{n \to +\infty} \frac{f(n)}{g(n)} < +\infty$$

3.2 Examples

Problem 2. $2^n = O(2^{3n})$?

Yes.

$$\begin{aligned}
&\lim_{n \to +\infty} \frac{2^n}{2^{3n}} \\
&= \lim_{n \to +\infty} \frac{2^n}{2^n 2^n 2^n} \\
&= \frac{1}{2^{2n}} \\
&= 0
\end{aligned}$$

Problem 3. $2^{3n} = O(2^n)$?

No.

$$\begin{aligned}
&\lim_{n \to +\infty} \frac{2^{3n}}{2^n} \\
&= \lim_{n \to +\infty} 2^{2n} \\
&= +\infty
\end{aligned}$$

Problem 4. $\log_2 n = O(\log_3 n)$?

Yes. Recall the change of base formula for logarithms: $\log_2 n = \frac{\log_3 n}{\log_3 2}$. We can disregard $\log_3 2$ as a constant which does not depend on n.