20200110

Determining "Big-O" Classification

In class I glossed over some basic details that relate to constructing a timing function for an algorithm. In these notes I cover these steps in some detail. They are very straightforward and should be already familiar.

To determine the timing function for an algorithm we count the fundamental operations as a function of the size of the input. But when we do this, we **usually** just count the operations that involve the actual data. (There are exceptions. For example, in Assignment 1 you are asked to count function calls.) In other words we ignore things like index variables and execution control operations. As we will see, we don't even need to be completely precise in our counting.

Consider this algorithm, which is written in pseudo-code that I just made up. Notice that I'm leaving out all declarations.

CODE	OPERATIONS
`A1: n = read()`	`2 (1 I/O and 1 assignment)`
`for i = 1 to n:`	
`A[i] = read()`	`2*n (1 I/O and 1 assignment, repeated n times)`

We don't count any of the operations relating to the loop management because they don't involve the data.

So we would write the timing function for A1 as $T_{A1}(n) = 2n + 2$

(Note for purists: the size of the input here is actually n+1 since that is the total number of read actions we execute. For our purposes here, calling it n is fine.)

Now two more simple algorithms:

CODE	OPERATIONS
`A2: n = read()`	`2 (1 I/O and 1 assignment)`
`for i = 1 to n:`	
`A[i] = read()`	`2*n (1 I/O and 1 assignment, repeated n times)`
`for i = 1 to n:`	
`for j = 1 to n:`	
`print A[i] + A[j]`	`2*n^2 (2 ops, n^2 times)`

So we would write the timing function for A2 as $T_{A2}(n) = 2n^2 + 2n + 2$

CODE		OPERATIONS	
A3:	n = read()	2	(1 I/O and 1 assignment)
	for i = 1 to n:		
	A[i] = read()	2*n	(1 I/O and 1 assignment, repeated n times)
	B[i] = 2*A[i]	2*n	(1 I/O and 1 assignment, repeated n times)

So we would write the timing function for A3 as $T_{A3}(n) = 4n + 2$

Our goal is to use the timing functions as a way of comparing the efficiency of algorithms. But as we have already seen, they are somewhat approximate because they don't count every single operation. So instead of comparing the explicit timing functions for different algorithms, we use the timing functions to collect algorithms into groups. Then to compare two algorithms, we compare the groups they are assigned to.

We group algorithms together based on the ***growth-rate*** of their timing functions. To illustrate this we can look at the the three algorithms above and see what happens when we repeatedly double the value of n (i.e. double the size of the input).

n	$T_{A1}(n)$	$T_{A2}(n)$	$T_{A3}(n)$
1	4	6	6
2	6	14	10
4	10	42	18
8	18	146	34
16	34	546	66
Etc.			

Now how fast are these timing functions growing? Let's look at the ratios for successive values in the columns. For A1, the ratios are $\frac{6}{4}, \frac{10}{6}, \frac{18}{10}, \frac{34}{18}$ etc. We can see that these ratios are getting closer and closer to to 2 ... can you see why they will never quite reach 2?

For A3, the sequence of ratios is almost identical (it's just missing the $\frac{6}{4}$ term) so it has the same behaviour.

For A2, the sequence of ratios is $\frac{14}{6}, \frac{42}{14}, \frac{146}{42}, \frac{546}{146}$... it's a bit harder to see the pattern.

The ratios work out to (approximately) 2.3, 3, 3.5, 3.7 ... and if we went further, we would see that the ratios approach 4 but never quite reach it.

So when we double the size of the input (ie. the size of the input increases by a factor of 2), T_{A1} and T_{A3} also increase by a factor of (slightly less than) 2, but T_{A2} increases by a factor of (slightly less than) 4.

Experiment: What if we try increasing the size of the input by a factor of 3? That is, start with n = 1, then n = 3, then n = 9, 27, 81, etc. You can work it out, but I'll jump to the results: T_{A1} and T_{A3} also increase by a factor of (slightly less than) 3, and T_{A2} increases by a factor of (slightly less than) 9.

In general, we find that if the input n increases by a factor of k, T_{A1} and T_{A3} also increase by (slightly less than) a factor of k. We can write this as

$$\frac{T_{A1}(k*n)}{T_{A1}(n)} \leq k \quad \text{and} \quad \frac{T_{A3}(k*n)}{T_{A3}(n)} \leq k$$

Similarly, we find that when n increases by a factor of k, T_{A2} increases by (slightly less than) a factor of k^2. We can write this as

$$\frac{T_{A2}(k*n)}{T_{A2}(n)} \leq k^2$$

We got to those conclusions by observation, but we can reach the same conclusion algebraically. For example, we can write

$$T_{A2}(n) = 2n^2 + 2n + 2$$
$$T_{A2}(k*n) = 2(k*n)^2 + 2(k*n) + 2$$
$$= k^2 * 2 * n^2 + k * 2 * n + 2$$

and we see that $\frac{T_{A2}(k*n)}{T_{A2}(n)}$ is always $\leq k^2$

Let's focus on $T_{A1}(n)$. We have seen that it grows linearly (ie at the same rate) as n grows. Can we use that information to give any information about the actual value of $T_{A1}(n)$?

Suppose there is some particular value n_0 for which we can determine that $T_{A1}(n_0) \leq c * n_0$ for some positive constant c. Now consider $T_{A1}(k * n_0)$ where $k \geq 1$

From our previous discussion, we know $\frac{T_{A1}(k*n_0)}{T_{A1}(n_0)} \leq k$

and from there it is a simple step to $T_{A1}(k * n_0) \leq c * (k * n_0)$

Now if we replace "$k * n_0$" by a generic "n", we get

$$T_{A1}(n) \leq c * n \quad \forall n \geq n_0$$

Are there such an n_0 and constant c? Yes! We can see that if we let $n_0 = 1$ and $c = 4$, the requirements are satisfied.

Now what about T_{A3}? You can work out that the same property holds (though you cannot use the same value for c)

But what about T_{A2}?

Suppose we start by establishing that for some value n_0, $T_{A2}(n_0) \leq c * n_0$ for some constant c

Now we can consider $T_{A2}(k * n_0)$. From our previous analysis, we know

$$\frac{T_{A2}(k * n_0)}{T_{A2}(n_0)} \leq k^2$$

which gives

$$T_{A2}(k * n_0) \leq c * k * (k * n_0)$$

Now if we replace "$k * n_0$" by "n" we get

$$T_{A2}(n) \leq c * k * n \qquad \forall n \geq n_0$$

.... which does not fit the same pattern as we saw for T_{A1} and T_{A3}. In fact it is kind of confusing because it still has a k in it ... but remember that we used n to replace $k * n_0$

and if $n = k * n_0$ then $k = \frac{n}{n_0}$ ... and we can use this to replace the k in the right hand side! This gives

$$T_{A2}(n) \leq c * \frac{n}{n_0} * n \quad \forall n \geq n_0$$

i.e. $T_{A2}(n) \leq \frac{c}{n_0} * n^2 \ \forall n \geq n_0$

Since c is a constant and n_0 is also a constant, $d = \frac{c}{n_0}$ is a constant. Thus

$$T_{A2}(n) \leq d * n^2 \qquad \forall n \geq n_0$$

Was there anything particular about the timing functions that we used? Not really.

Suppose an algorithm A has timing function

$$T_A(n) = a_t * n^t + a_{t-1} * n^{t-1} + \cdots + a_1 * n + a_0$$

where the a_i values are constants.

Claim: $\exists\ n_0$ such that $\forall n \geq n_0, \quad n^t \geq a_{t-1} * n^{t-1} + \cdots + a_1 * n + a_0$

Proof: Suppose not. Then $\forall n, \quad n^t < a_{t-1} * n^{t-1} + \cdots + a_1 * n + a_0$

$$\Rightarrow \quad 1 < \frac{a_{t-1}}{n} + \cdots + \frac{a_1}{n^{t-1}} + \frac{a_0}{n^t}$$

As n increases, each term in the sum on the right gets smaller, and in fact gets arbitrarily close to 0. Thus there is a value of n for which each term in the sum is $< \frac{1}{t}$. For this value of n the sum on the right hand side is < 1... which is a contradiction. Therefore such an n_0 exists.

$$\Rightarrow \forall n \geq n_0, \quad T_A(n) \leq a_t * n^t + n^t$$

ie $\forall n \geq n_0, \quad T_A(n) \leq (a_t + 1) * n^t$

Let $f(n)$ and $g(n)$ be non-negative valued functions on the set of non-negative numbers. If there are constants c and n_0 such that $f(n) \leq c * g(n) \quad \forall n \geq n_0$ then we say $f(n) \in O(g(n))$

The significance of this is that as n gets large, the **growth-rate** of f(n) is no greater than the **growth-rate** of g(n). In other words, the growth of g(n) is an **upper bound** on the growth of f(n).

Putting all of this together, we find that

$T_{A1} \in O(n)$ and

$T_{A3} \in O(n)$ and

$T_{A2} \in O(n^2)$

which looks like a pretty clear distinction between T_{A2} and the other two …

but is it? Can we be sure that T_{A2} is not also in $O(n)$? Let's check that out.

Suppose $T_{A2} \in O(n)$. Then there exist constants n_0 and c such that

$T_{A2}(n) \leq c * n \quad \forall n \geq n_0$

ie

$2n^2 + 2n + 2 \leq c * n \quad \forall n \geq n_0$

$2n^2 + (2 - c)n \leq -2 \quad \forall n \geq n_0$

$(2n + 2 - c)n \leq -2 \quad \forall n \geq n_0$

But $\forall n \geq c$, the left hand side is positive so the inequality *does not* hold for all $n \geq n_0$ (note that it makes no difference which of c and n_0 is larger)

Therefore $T_{A2} \notin O(n)$

And now, finally, we are sure that T_{A2} does not belong to the same class of function as T_{A1} and T_{A3}