Running time of an algorithm

We have seen several algorithms so far in the course, and we have seen that some of them are faster than others. We would like to be able to characterize the time it takes to run an algorithm. At first glance it is not obvious how to do so, since there are several different questions one might ask.

- What operations are involved in each instruction? For example, in the instruction

  ```
  x = myVar.myMethod( 2.3*y + a[i], myOtherArg);
  ```

  there is an expression that is evaluated (which involves a floating point multiplication, an index into array, an addition), there is the call to a method (which involves some underlying bookkeeping), there is an assignment of a value (to `x`), etc. How many operations should we count here?

- What computer are we using? Some computers are "advertised as" faster than others. But such advertising hides important details: some computers may be faster at doing specific types of operations, but slower at doing other types of operations.

- What programming language are we using? The same algorithm might run differently when you code it up in Java than if you code it in C++ or C, even on the same computer.

It is quite common in computer science to analyze the time taken by an algorithm in a manner that is *independent of the above issues.* One tries to express the time taken only as a function of the "size n of the input". The size could be the number of items e.g. in a searching or sorting problem, or it could be the number of bits n in an operation e.g. integer multiplication (discussed in lecture 2), or a number to be computed e.g. the n-th Fibonacci number, or the n-th prime number.

Insertion sort revisited

Let's recall the insertion sort algorithm, which I write below using "pseudocode".

ALGORITHM: insertion sort
input: an array $a[]$ with n elements
outputs: the array with elements in non-increasing order

```
maxval ← a[0]
for i = 1 to n − 1 do
    tmp ← a[i]
    cur ← i
    while (cur > 0) & (tmp > a[cur − 1]) do
        a[cur] ← a[cur − 1]
        cur ← cur − 1
    end while
    a[cur] = tmp
end for
```

Notice that some of these operations are executed a constant amount of time (independent of n). Some are executed at most n times (depending on the values of the array $a[\,]$). Some are executed n^2 times (again, depending on the values of $a[\,]$).

Best case

If the array is already sorted from largest to smallest, then the condition tested in the `while` loop will be false every time (since $tmp < a[cur-1]$), and so each time we hit the `while` statement, it will take a *constant* amount of time, rather than a time that depends on n. This is the *best case* scenario, in the sense that the algorithm executes the fewest operations in this case. Since there are n passes through the `for` loop, the time taken is proportional to n.

Worst case

The worst case scenario is that the array is already sorted, but it is sorted from smallest to largest. In this case, the i^{th} pass through the `for` loop will cause the *while* loop to be executed i times. The total amount of times the i loop will be executed is therefore

$$\sum_{i=1}^{n-1}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

which is a quadratic function of n. This is much slower than the best case, which grew linearly with n. (If you don't know how I got the expression $n(n-1)/2$, then you should check out the notes from today's tutorial.)

"Asymptotic efficiency"

Big O - upper bound

Let $t(n)$ represent the "time" it takes an algorithm to run, as a function of the positive integer variable n. We will be more concrete than that, and define $t(n)$ to be a particular sequence of positive numbers. Let $g(n)$ be some other sequence of positive numbers (not necessarily integer values, though). Typically, we will consider $g(n)$ to be one of the following functions

$$g(n) \in \{\ 1, \log n,\ n,\ n\log n,\ n^2,\ n^3,\ 2^n, \dots\}.$$

Definition: We say that $t(n)$ is $O(g(n))$ – "big O order of $g(n)$" – if there exists a positive integer n_0 and a constant c such that

$$t(n)\ \leq\ c\ g(n)$$

for all $n > n_0$.

The idea is that $t(n)$ grows *no faster* than some constant times $g(n)$, for sufficiently large n. By comparing $t(n)$ to a simpler looking $g(n)$, such as listed above, we are ignoring the constant factors. This will make more sense once you have seen some examples. First, though, a few notes:

- For any $g(n)$, we can think of $O(g(n))$ as a set of sequences, namely those sequences $t(n)$ that satisfy the above definition. In this interpretation, we would say $t(n) \in O(g(n))$ rather than $t(n)$ "is" $O(g(n))$. With this 'set membership' interpretation, we would have (though its not obvious in each case):

  $$O(1) \subset O(\log n) \subset O(n) \subset O(n\log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \cdots \subset O(n!)$$

  for example, any function that is $O(\log n)$ automatically has to be $O(n)$, etc.

- Saying $f(n)$ is $O(1)$ is a bit strange, since there is no n dependence in the $g(n)$, and the definition of big-O requires one. But it does work: Applying the definition, we have that there exists a c and n_0 such that $f(n) < c$ for all $n \geq n_0$. Notice that the condition $n \geq n_0$ is rather unnecessary in the case of $O(1)$, since there are only finitely many n values less than n_0, so if the definition holds for all $n \geq n_0$ then we could find another c such that it would automatically hold for all values between $0 \leq n \leq c$ as well.
- In addition to saying "$t(n)$ is $O(g(n))$" and "$t(n) \in O(g(n))$", it is common to say "$t(n) = O(g(n))$". This seems like an abuse of notation. The reason people allow themselves to write this, though. In particular, it is sometimes useful to write things like "$t(n) = 3n^2 + O(n)$" which just means that $t(n)$ has a specific dependence on n^2 (namely a factor of 3), plus something which is of order n (which is strictly smaller than n^2, and so we don't care about it).

Example 1

The function $t(n) = 5 + 100n$ is $O(n)$. To prove this, we write:

$$\begin{aligned} t(n) &= 5 + 100n \\ &< 5n + 100n, \text{for} n > 1 \\ &= 105n \end{aligned}$$

and so $n_0 = 1$ and $c = 105$ satisfies the definition.

Note that $t(n)$ is also $O(n^2)$ since $t(n) \leq 105n \leq 105n^2$ for $n \geq 1$. However, this is not so interesting, since any function that is $O(n)$ is automatically $O(n^2)$.

Example 2

The function $t(n) = 17 - 46n + 8n^2$ is $O(n^2)$. To prove this, we want to show there exists a c and n_0 such that

$$17 - 46n + 8n^2 \leq cn^2 \ .$$

for all $n > n_0$. Dividing both sides by n^2, we now want to show that:

$$\frac{17}{n^2} - \frac{46}{n} + 8 \leq c.$$

But the first term is at most 17 and the second term is negative, so the inequality will hold when $c = 25$ and for all $n \geq 1$.