Week 12 - Efficiency

Updated November 26, 10pm

So far in this course, we've been concerned with learning about new ideas in making programs: recursion, object-oriented programming, and abstract data types. We've main been concerned with writing *correct* code, and understanding how algorithms work. For the last parts of the course, we're going to focus on some more technical details starting with *efficiency*.

This lecture, the fundamental question are going to be: How long does our code take to run? What makes some programs more efficient than others?

Review of Big-Oh

First, recall that for most algorithms, their running time depends on the size of the input -- as the input numbers or lists get larger, for example, we expect algorithms operating on them to increase as well. So when we measure efficiency, we really care about a *function* of the amount of time an algorithm takes to run in terms of the size of the input. We can write something like $T(n)$ to denote the runtime of a function of size n (but note that this isn't always necessarily n - see below).

How best to measure runtime? We might try to use tools like the `Timer` class we provided, but there's many factors: how powerful your machine is, how many other programs are running at the same time. As with Schrodinger's cat, even the act of observing the runtime can affect performance!

What about the number of basic steps an algorithm takes? This is a bit better, but still subtly misleading: do all "basic" operations take the same amount of time? What counts as a basic operation? Etc. etc.

This is where Big-Oh comes in: it allows an elegant way of roughly characterising the *type* of growth of the runtime of an algorithm, without actually worrying about things like how different CPUs implement different operations, whether a for loop is faster than a while loop, etc. Not that these things aren't important - see CSC369 and CSC488 - but they are much too technical for this course.

When we characterise the Big-Oh property of a function, we really are thinking about general terms like *linear*, *quadratic*, or *logarithmic* growth. For example, when we see a loop through a list like this:

```
for item in lst:
    # do something with item
```

we know that the runtime is proportional to n, the length of the list. If the list gets twice as long, we'd expect this algorithm to take twice as long. The runtime grows linearly with the size of the list, and we write that the runtime is $O(n)$.

Ignoring constants, focusing on long-term behaviour

In CSC165, you learn about the formal mathematical definition of Big-Oh notation, but this is not covered in this course. Intuitively, Big-Oh notation allows us to analyse the running time of algorithms while ignoring two details:

1. The constants and lower-order terms involved in the step counting: $5n$, $n + 10$, $19n - 10$, $0.05n$ are all $O(n)$ (they all grow linearly).
2. The algorithm's running time on small inputs. The key idea here is that an algorithm's behaviour as the input size gets very large is much more important than how quickly it runs on small inputs. (This is what's meant when we say that Big-Oh deals with that *asymptotic* runtime.)

Note that these points means that Big-Oh notation is not necessarily suitable for all purposes. For example, even though mergesort runs in time $O(n\log n)$ and insertion sort runs in time $O(n^2)$, for small inputs (e.g., lists of size <= 10), insertion sort runs significantly faster than mergesort in practice! And sometimes the constants are important too - even though quicksort is (on average) an $O(n\log n)$ algorithm, it has smaller constants than mergesort and so typically runs faster in practice. Neither of these practical observations are captured by the sorting algorithms' respective Big-Oh classes!

Input "size" very important!

Students learning about Big-Oh for the first time often get the impression that the *n* is extremely important, when nothing could be further from the truth. What's important when describing the asymptotic runtime is that we have a **clearly defined notion of input size**, no matter what variable names we choose! Here's a simple example:

```
def num_common(lst1, lst2):
    count = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                count += 1
    return count
```

Suppose `lst1` has length *a* and `lst2` has length *b*. Then the outer loop iterates *a* times, and for each of these iterations, the inner loop iterates *b* times. This means the `if` block is executed *ab* times, and since this is the part of the algorithm that takes the most time, this leads to a running time of $O(ab)$.

Note that the sizes of both lists must be taken into consideration, and that there's no "*n*" in the Big-Oh!

Worst-case vs. Best-case

Something mentioned last week when we discussed quicksort was that even if we fixed the input list size to be some constant *n*, the running time of this algorithm could still vary tremendously, from $O(n\log n)$ to $O(n^2)$!

In general, just knowing the size of the input does not tell the whole story for an algorithm; even inputs of the same size can have very different running times. Therefore in computer science we often talk about the *best-case* running time of an algorithm for a given size, and the *worst-case* running time of the same algorithm, and the two can be quite different!

As another example, consider doing linear search through a list of size *n*. In the best case, the item we're looking for is at the front of the list, and we can return `True` immediately after checking the first item in the list. This makes the best case running time $O(1)$, i.e. **constant time**, because it does not depend on the input size. On the other hand, the *worst-case* running time is $O(n)$, i.e., linear time (why?).

Trees trees trees

When we analyse the running time of our recursive algorithms on trees, there are two factors to keep in mind: the **size** of the tree (i.e., number of nodes), and the **height** of the tree.

As we saw in lecture, depending on how many recursive calls our algorithm makes, we can either get algorithms whose number of recursive calls is proportional to the tree's height, or proportional to the tree's size.

Even though it seems like the former situation is much better than the latter, this is not always the case. Take a tree with n nodes - depending on the structure of the tree, its height can be as small as $\log n$, and as large as n itself!

Thus for tree methods, a very important consideration is to keep the tree's height small even as the number of nodes grow. Though this is not at all the case for our BST mutating methods (why?), in CSC263 you'll learn about more copmlex algorithms that do achieve this goal, resulting in a very efficient data structure.

Study Tip!

As you review the past data structures of the course, practice using Big-Oh notation to describe each algorithm you saw. What is the asymptotic complexity of inserting an item into a heap? Why? What about searching in a linked list? What about searching in a general tree?

Note that you'll need to understand what is meant by the "input size" in each case, and also be able to distinguish between the best and worst cases for each method.

Yes, there are many methods; remember, we aren't asking you to memorize anything for the final, but we will expect you to be able to determine the asymptotic running times of algorithms with a similar complexity on the final exam!

Computer Science
UNIVERSITY OF TORONTO

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)