CS 228: Introduction to Data Structures
Lecture 9

Introduction to the Analysis of Algorithms

There are several ways to compare algorithms. Possible criteria are:

- Speed
- Amount of memory
- Amount of network bandwidth use
- Ease of implementation
- Adaptability for reuse

The most significant for us are the first two, and we will concentrate on the first one.

A naïve way to compare the speed of two algorithms is to implement both, and measure them with a stopwatch ("wall clock time") or count CPU cycles. Here are some problems with this approach.

- It could actually be measuring aspects of the implementation details and runtime environment of each algorithm; e.g., CPU speed, memory speed, cache “locality”, whether just-in-time (JIT) compilation occurs, speed of the memory allocator, and the impact of garbage collection.

- It assumes that it trivial to implement both algorithms. It would be much better to have a scientific means to decide which algorithm to implement ***before*** committing resources (time and money) to coding.

- It says little about how an algorithm ***scales*** when we get a faster machine or increase the input size by an order of magnitude.

So, instead, our goal is to abstract the parts of an algorithm that are machine-dependent and express only those that depend on the algorithm itself.

The ***time complexity*** (or ***running time***) of an algorithm is a function that describes the number of basic execution steps in terms of the ***input size***. The time complexity abstracts the components of an algorithm’s performance that depend on the algorithm itself away from those components that are machine- and implementation-dependent.

Example: Sequential Search

The following basic problem comes up, for instance, when implementing the body of the **for** loop in Algorithm 1.

SEARCH
Input: An array A of length n and a value v.
Problem: Determine whether A contains v.

One solution is ***sequential search***:

Start at the beginning, scanning each successive element of A. Stop when v is found or no more elements are left.

The ***pseudocode*** for sequential search is given below. Pseudocode is not tied to any particular language. It just gives enough detail so that it is straightforward to translate it into any language.

`i = 0;`	*assignment:* 1 step
`while i < n`	*test:* n + 1 steps
`  if A[i] == v`	*test:* 1 step * n iterations of the loop
`     return true`	*return:* 1 step (at most once!)
`  i++`	*increment:* n times
`return false`	*return:* 1 step (at most once!)

The running time of sequential search depends on the particular values in A. If v is the first element, the algorithm finishes much more quickly than if v is not present. Let us assume the ***worst* case**, which is that we never find v. Then, the total number of steps is

$$T(n) = 3n + 3.$$

We do not know exactly how long it takes to execute any given step, but we can say this: The time complexity — i.e., the execution time — for an input of length n is ***proportional to*** T(n).

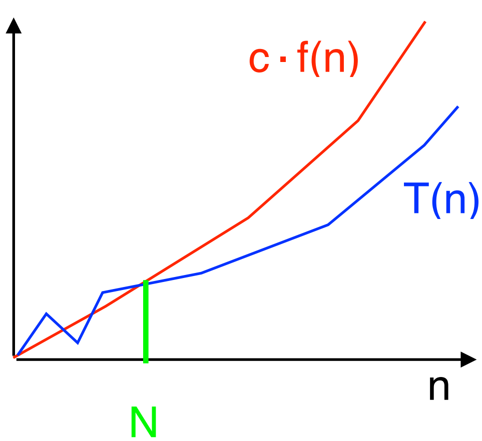
Big-O Notation

As n gets larger, the "+3" in T(n) = 3n + 3 becomes relatively insignificant, so the time complexity of sequential search is roughly proportional to 3n. We can simplify this statement further and say that T(n) is proportional to n or ***linear*** in n. So, instead of using the function T(n) = 3n + 3, we can use the simpler function f(n) = n. Yet another way to say it — the way computer scientists say it — is that the ***worst-case time complexity*** of this algorithm is O(n), or "big-O of n".

Definition. T(n) is O(f(n)) if and only if there exist positive constants c and N such that, for all $n \geq N$,

$$T(n) \leq c\, f(n)$$

Thus, T(n) is O(f(n)) if you can multiply f(n) by a (possibly large) constant c so that, ***asymptotically*** (as n shoots off to infinity), T(n) is ***completely underneath*** c f(n).



If you know calculus, you will notice that T = O(f) if and only if the limit T(n)/f(n) as n approaches infinity is finite.

Although the formal definition of big-O might be hard to digest at first, its intuitive meaning is easy to grasp. In fact, it is often easy to guess the correct O-bound for the worst-case running time of the algorithm, even if proving it might be more challenging.

Examples

Claim 1.

$$T(n) = 3n + 3 \text{ is } O(n)$$

Proof: Choose c = 4 and N = 3. Then, for any $n \geq 3$,

$$3n + 3 \leq 3n + n \leq 4n,$$

as required.

Claim 2.

$$T(n) = 42n + 17 \text{ is } O(n).$$

Proof: Choose c = 43 and N = 17.

These two examples illustrate a more general principle.

Fact. Every function T(n) of the form T(n) = an + b is O(n).

In particular, $T(n) = 10^9 n + 10^9$ is O(n). Set $c = 2*10^9$ and N = 1.

Now, you could also write T(n) = O(2n). While this is not incorrect, the constant is unnecessary. We want to make the simplest, cleanest statement possible about the running time, so we leave the constant factor 2 out.

Fact. When using-O notation we can ignore constant (multiplicative) factors!

You can think of O(n) as the ***class*** of all functions that do not grow any faster than a linear function, at least for large values of n.

Array Equality, Revisited

Let us return to the problem of determining whether two arrays, each consisting of distinct elements have the same elements, which we studied last time. We can write Algorithm 1 like this:

```
for i = 0 to n – 1:
    sequentially search for a[i] among b[0], . . . , b[n-1]
```

Here is more detailed pseudocode.

```
i = 0
while i < n
    found = false
    j = 0
    while j < n
        if a[i] == b[j]           (*)
            found = true
            break
        ++j
    if !found
        return false
    ++i
return true
```

The algorithm has a “loop within a loop” structure. In the worst case, the outer **`while`** loop goes through each of the n elements of a, and, for each one, the inner **`while`** loop goes through all n elements of b: a total of n × n pairs (`a[i]`, `b[j]`) of elements are compared in line (*). This happens even if a and b are identical and in the same order. This observation suggests that the time complexity of the algorithm is $O(n^2)$. This is indeed correct. To verify this, let us annotate the steps of the algorithm with the number of times they are performed.

	#Times performed	
`i = 0`	1	
`while` `i < n`	$n + 1$	
`found =` *`false`*	n	
`j = 0`	n	
`while` `j < n`	$n \times (n + 1)$	*at most*
`if` `a[i] == b[j]`	$n \times n$	*at most*
`found =` *`true`*	$n \times 1$	
`break`	$n \times 1$	
`++j`	$n \times n$	*at most*
`if` `!found`	n	
`return` *`false`*	0	
`++i`	n	
`return` *`true`*	1	
Total	$3n^2 + 8n + 3$	*at most*

The total number of steps is $T(n) = 3n^2 + 8n + 3$. Intuitively, when n is large, $8n + 3$ is negligible compared to n^2, so $T(n)$ scales like n^2. Next time, we will prove formally that, indeed, $T(n)$ is $O(n^2)$.