MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.081—Introduction to EECS I
Spring Semester, 2007

Lecture 4 Notes

Algorithms and Complexity

Problems and Algorithms

In computer science, we speak of problems, algorithms, and implementations. These things are all related, but not the same, and it's important to understand the difference and keep straight in our minds which one we're talking about.[1].

Generally speaking, a *problem* is defined by a goal: we'd like to have this list of numbers sorted, to map an image represented by an array of digital pictures into a string describing the image in English, or to compute the nth digit of π. The goals are about a pure computational problem, in which inputs are provided and an output is produced, rather than about an interactive process between computer and world. So, for example, we wouldn't consider "keep the robot driving down the center of the hallway" to be an algorithmic goal. (That's a great and important type of goal, but needs to be treated with a different view, as we shall see).

An *algorithm* is a step-by-step strategy for solving a problem. It's sometimes likened to a recipe, but the strategy can involve potentially unboundedly many steps, controlled by iterative or recursive contructs, like "do something until a condition happens." Generally, algorithms are deterministic, but there is an important theory and practice of randomized algorithms. An algorithm is correct if it terminates with an answer that satisfies the goal of the problem. There can be many different algorithms for solving a particular problem: you can sort numbers by finding the smallest, then the next smallest, etc; or your can sort them by dividing them into two piles (all the big ones in one, all the small ones in another), then dividing those piles, etc.. In the end these methods both solve the problem, but they involve very different sets of steps. There is a formal definition of the class of algorithms as being made up of basic computational steps, and a famous thesis, due to Alonzo Church and Alan Turing, that any function that could possibly be computed, can be done so by a *Turing machine*, which is equivalent to what we know of as a computer, but with a potentially infinite memory.

An *implementation* is an actual physical instantiation of an algorithm. It could be a particular computer program, in Python or Scheme or FORTRAN, or a special-purpose circuit, or (my personal favorite) the consequence of a bunch of water-driven valves and fountains in your garden. There is some latitude in going from an algorithm to an implementation; we usually expect the implementation to have freedom to choose the names of variables, or whether to use `for` or `while` or recursion, as long as the overall structure and basic number and type of steps remains the same.

It is very important to maintain these distinctions. When you are approaching a problem that requires a computational solution, the first step is to state a problem (goal) clearly and accurately. When you're doing that, don't presuppose the solution: you might add some extra requirements that your real problem doesn't have, and thereby exclude some reasonable solutions.

[1]This distinction is nicely described in David Marr's book, *Vision*

Given a problem statement, you can consider different algorithms for solving the problem. Algorithms can be better or worse along different dimensions: they can be slower or faster to run, be easier or harder to implement, or require more or less memory. In the following, we'll consider the running-time requirements of some different algorithms.

It's also important to consider that, for some problems, there may not be any algorithm that is even reasonably efficient that can be guaranteed to get the exact solution to your problem. (Finding the minimum value of a function, for example, is generally arbitrarily difficult). In such cases, you might also have to consider trade-offs between the efficiency of an algorithm and the quality of the solutions it produces.

Once you have an algorithm, you can decide how to implement it. These choices are in the domain of *software engineering* (unless you plan to implement it using fountains or circuits). The goal will be to implement the algorithm as stated, with goals of maintaining efficiency as well as minimizing time to write and debug the code, and making it easy for other software engineers to read and modify.

Computational Complexity

Here's a program for adding the integers up to n:

```
def sumInts(n):
    count = 0
    while i < n:
        count = count + n
    return count
```

How long do we expect this program to run? Answering that question in detail requires a lot of information about the particular computer we're going to run it on, what other programs are running at the same time, who implemented the Python interpreter, etc. We'd like to be able to talk about the complexity of programs without getting into quite so much detail.

We'll do that by thinking about the *order of growth* of the running time. That is, as we increase something about the problem, how does the time to compute the solution increase? To be concrete, here, let's think about the order of growth of the computation time as we increase n, the number of integers to be added up. In actual fact, on some particular computer (with no other processes running), this program would take some time $R(n)$, to run on an input of size n. This is too specific to be useful as a general characterization; instead, we'll say that

Definition 1 *For a process that uses resources $R(n)$ for a problem of size n, $R(n)$ has an* order of growth $\Theta(f(n))$ *if there are positive constants k_1 and k_2 independent of n such that*

$$k_1 f(n) \leq R(n) \leq k_2 f(n) \ ,$$

for n sufficiently large.

To get an idea of what this means, let's explore the assertion that our `sumInts` program has order of growth $\Theta(n)$. This means that there are some constants, let's imagine 5 and 10, such that the time to run our program is always between $5n$ and $10n$ miliseconds (you're free to pick the units

along with the constants). This means that each new integer we propose to add in doesn't cost any more than any of the previous integers. Looking at the program above, that seems roughly right. We'd say, then, that this is a *linear time* algorithm.

There are at least a couple if things to think about, before we believe this. First of all, you might imagine that the time taken to do all this, for $n = 1$ is actually much more than the half the time to do it for $n = 2$ (you have to set up the loop, etc., etc.). This is why the definition says "for n sufficiently large." It allows there to be some "start-up" time at the beginning, where the basic pattern hasn't established iself yet.

We have assumed that, after the startup costs, adding each new number in costs the same amount as the previous one. But that might not be true. For instance, eventually integers get so big that they don't fit in one "word" (32 or 64 bits in most current computers), and therefore actually require a lot more work to add. Some computer languages just won't add numbers that get too big, and generate an "overflow" error. Python will add *really* big integers, but once they get too big to fit in a word, the time it takes to add them will actually depend on how big they are. So, in fact, as n gets really big, the algorithm is either broken, or not linear time. Still the idealization of it as being linear is useful for a lot of conversations.

What about this algorithm for summing integers (due to Gauss, in his schooldays)?

```
def sumInts(n):
    return n * (n-1) / 2
```

If we can assume that arithmetic on all numbers has the same cost, no matter how big the number, then we'd say that this algorithm has *constant* or $\Theta(1)$ order of growth. Although the answer gets bigger as n increases, the time to compute it stays roughly constant. (Of course, again, if the numbers get bigger than a computer word, this idealization no longer holds).

Let's consider another example, this time of a program that adds up the elements of a list:

```
def sumList(a):
    i = 0
    count = 0
    while i < len(a):
        count = count + a[i]
    return count
```

What is the order of growth of its running time as a function of the length, n, of `a`? This is going to turn out to be a somewhat complicated question. Let's start with an easier one: What is the order of growth of the number of `+` operations, as a function of n? That is pretty clearly $\Theta(n)$, because the loop is going to be executed n times.

We could save one addition operation by writing this program instead:

```
def sumList(a):
    count = a[0]
    i = 1
    while i < len(a):
        count = count + a[i]
    return count
```