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
```

It will require $n - 1$ additions, instead of n. Convince yourself that the order of growth of the number of operations remains $\Theta(n)$.

This all seems pretty straightforward; why did we say it was complicated? First, let's think about the operation $a[i]$. We need to think about how long it takes to execute, because it is also going to be done n times. If a is implemented, in the low level of the computer, as an *array*, then the computer knows how to find the ith element just using some constant-time arithmetic calculations, and that operation is $\Theta(1)$. But, if it is implemented as a *linked list*, in which the only way to find the 5th element is to start at the beginning of the list, and "follow" down a chain of pointers, from one element to the next, then $a[i]$ takes time $\Theta(i)$.[2]

For the sake of argument, let's presume that $a[i]$ takes $\Theta(i)$ time; then how long does the whole loop take? It looks like it's going to take that's about like $n^2/2$. In fact, we can show that this is $\Theta(n^2)$, by choosing $k_1 = 0.5$ and $k_2 = 1$. An algorithm with this time complexity is said to be *quadratic*, because the time it takes to run increases with the square of the size of the problem.

But wait! There's more. There's one more thing going on in the loop. It's `len(a)`. We need to think about how long it takes for this to be computed. Again, it depends on the underlying representation of `a`. If it's an array (or a list or vector that can't share structure with other lists), then the language usually computes and stores the length whenever it is changed, and so a call to `len(a)` takes $\Theta(1)$ time. In Lisp, though, where lists are represented as linked lists, it takes $\Theta(n)$ time. Just to make our example interesting, let's assume that `len(a)` takes $\Theta(n)$ time. Then how long does our algorithm take? It seems, now like it's about n times through a loop that takes about $n + (n/2)$ time to execute. You should be able to pick constants to show that, nonetheless it takes time $\Theta(n^2)$.

This illustrates an important idea: the details of how an algorithm is implemented can change the constants or make an algorithm faster or slower by some multiplicative amount. But if you give me algorithm 1, which is $\Theta(n)$ and algorithm 2 that is $\Theta(n^2)$, then no matter how slow your computer is, or how much extra stuff (as long as it doesn't increase the order of growth) you add in your implementation of algorithm 1, there is some n for which algorithm 1 will be faster.

Fibonacci Example

Imagine that we wanted to compute the Fibonacci numbers. It's sequence of numbers, starting with 1, and 1, in which each subsequent number is the sum of the previous two numbers in the series: $(1, 1, 2, 3, 5, 8, 13, 21, ...)$. This series has all kinds of cool properties. Cite some here.

Now, imagine that you wanted to compute the nth Fibonacci number. A straightforward recursive program would be:

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

[2] As it happens, in Python, list access is $\Theta(1)$ time; but it's $\Theta(n)$ for Lisp and Scheme lists, and $\Theta(1)$ for Lisp arrays. There's always a tradeoff, though! Adding an element to the front of a list in Lisp is $\Theta(1)$ and is $\Theta(n)$ in Python.

Try it. It works fine. Except see what happens when you try to compute fib(50). Let's try to see what's happening with this computation. In order to compute fib(50), it first computes fib(49). That's a lot of work! And then, it still has to compute fib(48). But wait! We've already computed fib(48) as a recursive call when we were doing fib(49). This repeated recalculation of the values is happening all over the place, and making this take forever.

Well...not exactly forever. Let's see if we can figure out roughly how long it takes to compute fib(5). Let's assume that each addition takes 1 computational unit. Interestingly enough, the time taken to compute fib(5) is not so different from the value of fib(5) itself. It will require 8 (well, actually, 7) additions. That's not so bad. But fib(50) = 20365011074, and that's a lot of additions! In fact, fib(n) is well approximated by

$$\hat{f}(n) = \frac{1}{\sqrt{5}}1.618^{n+1} \quad .$$

We can see that this function grows exponentially in n. So, we can show that, in fact, the order of growth of this algorithm for computing the Fibonacci numbers is $\Theta(e^n)$. (Remember that changing the base of an exponent only incurs a multiplicative factor, so for the purposes of Θ, if the base of the exponent is independent of n, it doesn't matter which one you use).

Here's another way to compute the Fibonacci numbers. (Remember: computing the numbers is our problem; there are many algorithms for doing it).

```
def ffib(n):
    fibs = [1, 1]
    for i in range(2, n+1):
        fibs.append(fibs[i-1] + fibs[i-2])
    return fibs[n]
```

This algorithm is not nearly as beautiful. It works by computing the sequence, starting from element 2, and working its way up to element n (remember that range only goes up to the integer less than the upper bound). But the good thing about this is that when it needs fib(n-1), it will already have been computed and stored.

So, how many additions are necessary to compute ffib(50) in this algorithm? Only 49, because each new value is just the sum of two previous values. But we have to be careful about more than just the additions. We should probably worry about how much we pay to append to fibs. Depending on the underlying implementation of the append operation, it might copy the whole list each time, requiring $\theta(i)$ time, or it might run in $\theta(1)$ time.

So far, we've been talking only about time requirements of an algoirthm. But space is an interesting question, as well. The ffib algorithm requires $\Theta(n)$ space, to store all the partial results. That's pretty easy to see. What might be harder to see is that the recursive fib algorithm requires $\Theta(n)$ space as well, because it often has n subroutine calls currently pending, and the computer has to remember the state at each level. (If that didn't make sense to you now, don't worry about it).

To avoid issues of the efficiency of the implementation of append and to use only $\Theta(1)$ space, we can implement fffib by looping forward as in ffib, but only remembering the last two values:

```
def fffib(n):
    oneStepAgo = 1
    twoStepsAgo = 1
    for i in range(2, n+1):
        value = oneStepAgo + twoStepsAgo
        twoStepsAgo = oneStepAgo
        oneStepAgo = value
    return value
```

In this algorithm, the time taken to compute `fffib(n)` clearly grows as $\Theta(n)$.

As you will find if you experiment with them, the difference between a linear-time and an exponential-time algorithm is huge.

Memoization

As it happens, the `ffib` procedure we wrote above explores an important structural regularity in the definition of the Fibonacci numbers. The problem has two important properties:

1. The solution to a big problem can be constructed from solutions to subproblems; and
2. There is a lot of overlap in the subproblems encountered when solving a big problem.

These things are both clearly true for the problem of computing Fibonacci numbers.

A problem with both of these properties can be effectively solved efficiently using the idea of `dynamic programming`: whenever you solve a subproblem, remember the answer, and use it when you re-encounter that same subproblem.

You can apply this principle by hand, in a special purpose way for each problem, as we did in `ffib`. But an even cooler strategy is to recognize this as a common pattern of interaction, and write some code that can make any function be computed in a way that takes advantage of shared substructure. The idea is called `memoization`: we arrange to remember the values of the function applied to particular arguments, so if we try to call the function on those arguments again, we can just look up and return the result, without recomputing it.

Here is a way to generically memoize any function, written in Python:

```
def memoize(f):
    storedValues={}
    def doit(arg):
        if not storedValues.has_key(arg):
            storedValues[arg] = f(arg)
        return storedValues[arg]
    return doit
```

This procedure takes another procedure `f` as an argument, and returns a new procedure `memoize(f)`. For concreteness, let's apply it to `fib` and then talk about how it works.

```
fib = memoize(fib)
>>> fib(10)
```

We pass fib into memoize. It starts by making a new dictionary, called storedValues. We're going to use this dictionary to store pairs of (arg, f(arg)); that is, inputs arg that have been given to the function to be computed and their associated values f(arg). Next, memoize makes a procedure, called doit. This doit procedure takes an argument, arg. It checks to see if we already know the value of f(arg) by looking arg up in the dictionary. If it doesn't find the value there, it calls the original f to compute the value, and then stores it in the dictionary. Now, it's guaranteed that the appropriate value is in the dictionary (either it was there before, or we just computed it), so we just look it up and return it. Finally, memoize returns this new function.

It's important to realize that the function that gets returned refers to storedValues. But *each time* memoize *is called, it makes a new* storedValues, and puts it in an environment that only the procedure doit has access to,[3] so that means that each memoized procedure has its own separate set of values. That way we can memoize all sorts of different functions, and they all have their on private dictionaries.

Finally, to actually use this procedure, we have to do evaluate the following line, which might seem slightly crazy:

```
fib = memoize(fib)
```

This is crucial. If you instead did

```
flub = memoize(fib)
flub(10)
```

you'd be in trouble. Let's see why. We'll start by examining the flub case. We have two procedures floating around: the original fib procedure, and the new flub procedure. When we call flub(10), everything starts out nicely, and it executes the procedure (which we called doit, when we were constructing it) that was made by memoize. But since we can't find the value for 10 in the dictionary, flub calls the f that was passed into memoize, which is our original fib procedure. All is good until fib decides it needs to know the values for 8 and 9. The problem is that the only definition of fib that is available is our original one, and so it calls itself recursively and is off to the exponential-time races.

Instead, if we do fib = memoize(fib), it all works right. But we have to be careful with our nomenclature. Let's call our original fib procedure, the one that was passed into memoize, theProcedudureFormerlyKnownAsFib, or TPFKAF, for short. And we'll call the procedure returned from memoize fib. Because of the assignment statement, if anyone comes along and looks up the name fib in the global environment, they'll get the memoized function. So, now, we call fib(10); 10 isn't in the dictionary, so it calls the function that was passed into memoize, TPFKAF. Now, TPFKAF realizes that it needs to figure out values for 8 and 9. But now, instead of calling itself, it looks up the name fib in the environment, and finds the memoized fib instead. This means that, every time anybody wants to evaluate the function at some value, it is first looked up in the dictionary, and then if it isn't there, the original function is called, but *only for one layer.*

Add a picture here.

[3] This is exactly what was going on in the object-oriented programming notes, where we made a bank account with a value stored in an environment that was shared by the deposit and balance procedures.

Example: exponentiation

Here's another example of two algorithms for the same problem, and another general trick that can be applied in some cases for considerable speedup. Here are two implementations of the same, very basic algorithm for raising a base `b` to a non-negative integer exponent `e`.

```
def exp(b, e):
    result = 1
    for i in range(e):
        result = result * b
    return result

def exp(b, e):
    if e == 0:
        return 1
    else:
        return b * exp(b, e-1)
```

They both have a time complexity of $\Theta(e)$. It doesn't really matter whether we implement it using recursion or iteration; in either case, we multiply `b` together `e` times.

But we can do better than that! To construct a more efficient algorithm, we can apply the principle of *divide and conquer*: a problem of size n can be divided into two problems of size $n/2$, and then those solutions can be combined to solve the problem of interest. If the time to solve a problem of size $n/2$ is less than half of the time to solve a problem of n, then this is a better way to go.

Here is an exponentiation procedure that wins by doing divide-and-conquer:

```
def fastExp(b, e):
    if e == 0:
        return 1
    elif odd(e):
        return b * fastExp(b, e-1)
    else:
        return square(fastExp(b, e/2))
```

It has to handle two cases slightly differently: if `e` is odd, then it does a single multiply of `b` with the result of a recursive call with exponent `e-1`. If it's even, then really good stuff happens: we can compute the result for exponent `e/2`, and square the result. So, we have to do at most half the work we expected.

What is the time complexity of this algorithm? Let's start by considering the case when `e` is a power of 2. Then we'll always hit the last case of the function, until we get down to `e` $= 1$. How many recursive calls will result? Just $\log_2$ `e`. (Sorry about the notation here: `e` is our variable, not the base of the natural log.) It could be worse than that: if we start with a number with a binary representation like `1111111`, then we'll always hit the odd case, then the even case, then the odd case, then the even case, and it will take $2\log_2$ `e` recursive calls. Each recursive call costs a constant amount. So, in the end, the algorithm has time complexity of $\Theta(\log e)$.