CS125 : Introduction to Computer Science

Lecture Notes #31 and #32
Introduction to Algorithm Analysis

Lectures 31 and 32 : Introduction to Algorithm Analysis

In many situations, there are a number of algorithms we could use to solve a problem, and so we need to choose which one to use. This choice could be based on a number of factors – for example, the speed of the algorithm, how much memory it uses, or how much data we need to process. For any given situation, some of those factors might be more important to us than others. Therefore, we need a way to describe the time, speed, etc. requirements of an algorithm, so that we can easily compare algorithms and choose the one that is best for our situation.

For the moment, let's consider comparing the speed of two algorithms. If we have two algorithms, A and B, for solving the same problem, and if algorithm A is faster than algorithm B, then that is one possible reason for choosing algorithm A over algorithm B. The problem comes when we try to define what "faster" means. One thing we could do is to take two different machines, and run algorithm A on one machine, and run algorithm B on the second machine on the same data, and time the algorithms with a stopwatch. If we do that, perhaps the following are the results we get:

```
 Algorithm A         Algorithm B
 on machine 1        on machine 2
-------------        ------------
 92.7 seconds        20.8 seconds
```

Now, at first glance, it might seem like Algorithm B is much faster. However, if we are going to compare algorithms in this manner, we want to compare them in identical environments. For example, maybe machine 1 is a seven-year-old machine, and once we run *both* algorithms on machine 2, we get the following results:

```
 Algorithm A         Algorithm B
-------------        ------------
 10.4 seconds        20.8 seconds
```

Now, the comparison is a bit more appropriate, since the two algorithms are running on the same machine, the same operating system, etc.

However, things like processor speed and compiler optimizations (i.e. how good the translation from high-level code to machine code was) can affect this result, and those things really don't have anything to do with the abstract description of an algorithm. So, we'd prefer a more mathematical way of comparing two algorithms, so that we can focus on the key details of the algorithm without considering what platform things are running on, what compiler was used, or other such implementation details that have nothing to do with the inherent properties of the actual solution description itself. Even if the machine, operating system, compiler, etc. were the same for the tests of algorithm A and B above, we could still get different results if we used a different compiler, different machine, and different operating system. Above, algorithm B appears to take twice as long to run as does algorithm A. Perhaps if a different machine were used, the gap would not be so large. Or perhaps it would be larger.

We also have to take into account data size. Knowing how fast we can sort an array of 10 elements isn't particularly useful information, because we aren't always going to be sorting arrays as small as 10 elements. On the other hand, if we can describe the running time as a function of a general number of elements, `n`, then that is more useful. If we increase our data size from `n` to `2n`, then our running time function will give us the new running time if we substitute `2n` for `n` in the function.

So, perhaps the data above was for a set of 100 data items:

data size	Algorithm A	Algorithm B
100 elements	10.4 seconds	20.8 seconds

Perhaps if we run the same two algorithms (again, using the same machine, same operating system, same compiler, etc.) but this time we have 200 data items instead of just 100, perhaps we would get the following results:

data size	Algorithm A	Algorithm B
100 elements	10.4 seconds	20.8 seconds
200 elements	41.6 seconds	41.6 seconds

And with a few more data sizes, we might get:

data size	Algorithm A	Algorithm B
50 elements	2.6 seconds	10.4 seconds
100 elements	10.4 seconds	20.8 seconds
200 elements	41.6 seconds	41.6 seconds
400 elements	166.4 seconds	83.2 seconds

Now, this is not unrealistic data; we certainly *could* have two algorithms that exhibit this performance for the given data sizes. So the question is, why do we get this strange behavior? It seems that for some data sizes, algorithm A takes longer, but for some other sizes algorithm B takes longer. But actually, this behavior is not strange at all, and it illustrates the important concept we want to discuss today. Look closely at the numbers in the column for algorithm B. As the data size doubles each time – from 50, to 100, to 200, to 400 – the time the algorithm needs to run also doubles – from 10.4, to 20.8, to 41.6, to 83.2. Every time the data set doubles, the running time of the algorithm also doubles.

Now, look at the numbers for algorithm A. Every time the data size is doubled, the running time of algorithm A increases by a factor of 4.

If you were to graph the performance of these two algorithms as a function of the number of data elements, you would get a picture with a parabola (for Algorithm A) and a straight line (for algorithm B). That is, we would get a quadratic function and a linear function.

That is the quality we are after. What we are discussing here is the notion of *order of growth* – we want to know not how fast an algorithm is on one data set, but rather, how quickly the performance degrades as we increase the size of the data set. (We can make a similar analysis for memory usage.) If your time vs. data size graph is a quadratic function, as with Algorithm A, then it means every time you increase the data size by a factor of k, you increase the running time by a factor of k^2. But if your time vs. data size graph is a linear function, as with algorithm B, then it means every time you increase the data size by a factor of k, the running time also increases by a factor of k. This would suggest that, as our data size gets larger and larger, the running time of the algorithm with a quadratic order of growth, would get larger much faster than the running time of the algorithm with a linear order of growth – and indeed, we can see that exact affect in our data above. Eventually, for large enough data sets, every algorithm with a running time whose order of growth is quadratic, will start to take longer than an algorithm with a running time whose order of growth is linear.

So, we will start by considering operations that do *not* depend on `n`. Meaning, some parts of a function take exactly the same amount of time whether `n` is 10 or 1000 or 1 million. For example:

- Declaring a variable would be such an operation. Now, if we declare 100 variables, certainly that will take longer than if we declare 1, but if (for example) we are about to search an array of elements using a `for`-loop, whether we are going to search 100 elements or 1 million elements, declaring a single variable `i` to use to run the `for`-loop iteration will take the same amount of time either way.

- A basic operation such as comparison, addition or subtraction, a logical operation, or assignment would be such an operation For example, the time it takes to write *one* object location to a reference variable (i.e. assignment) does not vary based on how many assignments you eventually do. The time to do *one* addition doesn't suddenly increase because you plan on doing 10000 additions instead of 100.

- Array access is such an operation as well. At first, you might not think so. It would seem that to get to cell (for example) 10, the machine first needs to move through cells 0 through 9. Therefore, it would seem that accessing a later cell takes longer than accessing an earlier one.

  But, actually, this is not the case. Arrays can given you (nearly) instant access to any cell because in memory the cells are arranged one after the other. So, it is possible to hold the starting address inside the array reference, and then use the calculation

  ```
  cell address = starting address + index * typesize
  ```

  to determine the starting address of the cell you want. For example, we said that variables of type `int` take up 32 bits. So, if we allocate an array of 10 `ints`, those ten cells take up ten consecutive 32-bit cells in memory. The first cell starts at some address in memory – the starting address of the array – and then the other cells are located immediately after it, one by one. But, if you use the calculation above to get the address of the first cell, you would get:

  ```
  address of A[0] = (starting address of A) + 0 * 32 bits
                  = (starting address of A) + 0
                  = starting address of A
  ```

  and we just said above that the first cell was located at the start of the array. Likewise,

  ```
  address of A[2] = (starting address of A) + 2 * 32 bits
                  = (starting address of A) + 64
  ```

  And, if `A[0]` is located at the starting address of the array and takes up 32 bits, it would follow that at bit 33 of the array, we would see the start of `A[1]`. And since that takes up another 32 bits, it would follow that at bit 65 of array, we would see the start of `A[2]`. And that is exactly what our calculation above tells us – that 64 bits after the starting address, `A[2]` begins.

  In other words, we can jump immediately to cell 2 by doing one multiplication and one addition. And in general, we can get to *any* cell in the array via one multiplication and one addition. This is because array cells appear consecutively in memory, and so if you want `A[i]`, that means (because we start indexing at 0) that you have to skip over `i` cells at (in this case) 32 bits each, and so one multiplication tells you how many total bits to skip over, and one addition of that bit total to the starting address tells you where the start of `A[i]` must therefore be in memory. You do not need to traverse down the cells one by one – a single multiplication and addition will give you your array cell each time, no matter how large the array is and no matter which particular index you are looking up. So, array cell access time does not depend on the size of the array in any way.

Operations like those above are said to take *constant time*, because their running time is the same constant value regardless of the data size. Similarly, we can say that algorithms take *constant memory* if the amount of memory the algorithm uses, does not increase as the data size increases. In general, if the amount of a resource (time, memory, whatever) that is used, is unaffected by the growth of the data size, then we say that the order of growth of that resource usage is *constant*.

Similarly:

- If the increase in the usage of a resource grows proportionally to the increase in the size of the data set, then we say that the usage of that resource by the algorithm has an order of growth that is *linear*.

  A linear-time algorithm is one that grows linearly as the amount of data to process grows. For example, a loop that counts down from n to 0 in steps of 1 would be linear – each loop pass subtracts 1 from n, so you'll need n of those loop passes to reduce the number to 0. And the substraction is constant-time, so you are performing constant-time work n times. The math function c*n would be a linear function (i.e. a straight line).

  That's one way to recognize a linear function – if you are doing a constant amount of work for each step, and the number of steps you do is proportional to n. Or in other words, you are eliminating a constant amount of data with each constant step. Printing an array is another example – each step, do you a constant-time amount of work to print one cell, so if you have n cells, you need to do that constant-time amount of work n times.

  So, another way to recognize a linear algorithm is to double the amount of data, or triple it, or quadruple it. What happens to the running time? If it likewise doubles, or triples, or quadruples, respectively, then you have a linear algorithm. This is because since you perform constant-time work to process a constant amount of the information, if you double the amount of information, you will be doubling the amount of times you have to run that constant-time work.

- The order of growth of an algorithm's usage of a particular resource is said to be *quadratic* if, when the data size grows by some factor, the resource usage grows by the square of that factor, rather than growing by that same factor as a linear algorithm would do. An example is printing an n row, n column array to the screen. That is, if n is the number of rows of the array and # rows == # cols, then if n doubles, we have four times as many cells and thus four times as much printing work to do. If n triples, we'll have 9 times as much printing work to do.

  Another way to view this is that you do a linear amount of work (such as printing an entire row of an array) to eliminate a constant amount of our count (one row of n).

  If a quadratic-time algorithm has it's running time graphed as a function of the data size, the graph will be a parabola – i.e. a quadratic function.

- The order of growth of an algorithm's usage of a particular resource is said to be *logarithmic* if an increase in the data size by some factor, increases the resource usage by the log of that factor. For example, in the case of logarithmic running time, each constant amount of work eliminates an entire fraction of the data, so if you double the amount of work, you are adding only one more operation total – i.e. adding constant time to the running time (because log-base-2 of 2, is 1). An example is binary search – you compare A[mid] to the key, first for equality and then to see if the key is less than A[mid] – and those two comparisons together eliminate at least half the array from having to be examined. So the next step is on at most half the original array. And the next step is on at most half of *that* subarray, which is a fourth of the original array. And so on. The number of steps needed to reduce the subarray are dealing with to size 1 is going to be a logarithmic number of steps, and if you graph a logarithmic-time algorithm's running time as the size of the data varies, you get the graph of a logarithmic function.

We aren't very concerned with the lower order terms, and constant factors, and such, here. 4.332n and 1.232n+2 are still both linear functions. 0.5 (n squared) and 213.234 (n squared) + 3n - 2 are still both quadratic functions. In fact, you can even take a quadratic function and see how little the lower order terms start to matter as `n` increases:

n	n^2 + 2n + 1	n^2 / (n^2 + 2n + 1)
10	121	.826446
100	10,201	.980296
1000	1,002,001	.998003
10000	100,020,001	.999800
100000	10,000,200,001	.999980

The column at the far right measures what percentage the quadratic term is, as a total of the entire expression. As you can see, even when `n` is `100`, the quadratic term accounts for almost the entire value of the expression. This result just gets more prominent the larger `n` gets. So, we don't have to concern ourselves too much with the particular details of the function. All we really care about is whether the function is linear or quadratic or such. (You'll learn some formal notation for ignoring the lower-order terms in CS173.)

This information is useful because we can use it to learn what kind of effect increasing our data set is likely to have. If it is taking me about an hour to search an array for an element (it's a big array), then I know trying to search an array about twice the size should take me about two hours. Trying to search an array four times the size should take me about four hours. But this has nothing to do with the operating system I am using, it has nothing to do with the processor in my machine, and it has nothing to do with the compiler I used. The fact that the running time is linear in the size of the array is a *fundamental mathematical property of the algorithm!!*.

That means our analysis is completely platform-and-software-tool independent. That is the best way to compare algorithms, because it helps us choose between algorithms when we are still in the process of designing our program, and because the decisions made based on such a comparison don't change when a new processor comes along because the result of the comparison of two algorithms using mathematical analysis is completely independent of the processor you used. The algorithm that is mathematically faster today will still be mathematically faster next year when processors have doubled in speed.

Up to this point, the analysis we have done is what is known as *worst-case* analysis. In this kind of analysis, we are concerned with finding out how long the algorithm takes to run in the worst possible situation. For example, you are designing software to pipe extra oxygen into the space shuttle living area for our astronauts in the event that their existing oxygen all leaks out somehow, you'd want to make sure that the algorithm you chose to make this oxygen delivery system work was very very fast, even in the worst case. It would do you no good if it "usually" worked quickly, but "hit a snag" in certain situations and spent an extra hour processing before piping in oxygen. The astronauts would die waiting for your algorithm to finish!! So, in such a case it would be important to know the absolute longest that your algorithm could take, no matter *what* input the algorithm is given. That is the idea of worst-case.

On the other hand, often what we are more concerned with is what *usually* occurs. We might not like using a computer system that took 10 minutes to send a file to the printer each time we wanted to print. But, if it usually printed right away, and only took 10 minutes every once in a long while due to some quirk in certain types of files, then we might find that acceptable. In fact, we might prefer that to a situation where the worst case was only 5 minutes, but that was what "usually" happened as well. Sometimes we will prefer to focus on what "usually" happens, and pick an algorithm that "usually" runs very fast, even if occasionally we end up with a slower case as a result.

This notion of "usual" is known as the *average case*, and attempts to analyze an algorithm with the average case in mind are known as *average case analysis* (in contrast to *worst case analysis*).

Sometimes – though not usually – we also discuss the "best case". "What happens in the most ideal situation?" is what we are asking here. We usually don't care about this case because it's not interesting. We don't *care* if things work out better than we expect, but we *do* care if they end up worse than we expect...so while we care about what the worst we can expect to see would be, we don't care too much about what the possible best we can expect to see it. We care about the worst possible situation, and also what is most likely.