Chapter 2

Weiss Chapter 2 Notes

This course is about algorithms. Before we actually begin our study of any specific algorithms, however, we'll start by studying some of the mathematics necessary to analyze the resources (such as processing time, memory, or disk storage) consumed by the execution of an algorithm.

Two algorithms that solve the same problem may consume wildly different quantities of resources– and this usage may depend upon the inputs to the algorithms, which are not known until the algorithm is actually executed. Depending on the requirements of the system that the algorithm is part of, the quantities of resources used may have a profound effect on the usability of the system, or even whether the system is successful or not.

2.1 Definitions

The *order* of a function $T(N)$ is a description of its rate of growth as N becomes very large.

When analyzing the order of algorithms, the following definitions are commonly used:

- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

  As an example, if $T(N) = 2N + 1$, then $T(N) = O(N)$. To prove this, all we need to do is find c and n_0 such that $2N + 1 \leq cN$ when $N \geq n_O$. If we try $c = 3$, then we can subtract $2N$ from both sides of the inequality and get $1 \leq N$. This is clearly true when $N \geq 1$, so n_0 is 1.

  It may seem strange that in this case $f(N)$ is clearly less than $T(N)$ for all positive values of N, and yet $T(N) = O(f(N))$. The actual values of $T(N)$ and $f(N)$ are not the issue– the important issue is whether a c and n_0 can be found that satisfy this rule.

- $T(N) = \Omega(f(N))$ if there are positive constants c and n_0 such that $T(N) \geq cf(N)$ when $N \geq n_0$.

  This is similar to the big-O notation, but from the other direction– the order of $f(N)$ is typically as large as possible without making it larger than the order of $T(N)$.

  As an example, if $T(N) = 2N + 1$, then $T(N) = O(N)$. To prove this, once again all we need to do is find c and n_0 such that $2N + 1 \geq cN$ when $N \geq n_O$. If we try $c = 2$, then we can subtract $2N$ from both sides of the inequality and get $1 \geq 0$. This is clearly true for any N.

- $T(N) = \Theta(f(N))$ if and only if $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$.

- $T(N) = o(f(N))$ if and only if $T(N) = O(f(N))$ and $T(N) \neq \Theta(f(N))$.

Remember that in this notation, our goal is to find $f(N)$ (an order), given $T(N)$ (a function). For example, imagine that we have a function $g(N) = 3n^2 + 12$, and we would like to say things about it using big-O notation. It is correct to say "$g(N)$ is $O(n^2)$" but it is usually not correct to say "$O(g(N))$ is n^2".

2.1.1 Big-O Conventions

It may seem strange that the order of $f(N)$ might actually be *greater* than the order of $T(N)$. For example, if $T(N) = O(n^2)$, then it is also true that $T(N) = O(n^3)$, and so on.

By general convention, $f(N)$ is chosen to be the smallest order that is known to be greater or equal to the order of $T(N)$. In this example, you would never say that $T(N) = O(N^3)$ if you knew that $T(N) = O(N^2)$. It would be correct, at least in a mathematical sense, but potentially very misleading to the reader. In fact, in many contexts, big-O is used synonymously with big-theta.

As another convention, any constant factors are dropped from the order. For example, according to the definition given above, it is true that $3n^2 = O(3n^2)$. However, it is also true that $3n^2 = O(n^2)$. The latter form, where the constant factor has been omitted, is the standard way of expressing the big-O of a function.

Similarly, it is also a convention to drop all but the "largest" term of the order. The largest term is the *dominant* term (see section 2.2.1 for more information). For example, $n^2 + 100n + 12 = O(n^2)$, since as n becomes large, the quadratic term will grow more quickly than the others.

2.1.2 Why Not Use Θ All The Time?

At first glance, it would appear that Θ is all we really want or need– after all, it's the "right" answer, where the order of $T(N)$ is exactly $f(N)$. If we know the real order of $T(N)$, then we know $\Theta(f(N))$.

Unfortunately, life is not so simple. Consider the following code snippet:

```
void    strange1 (int A[N])
{
        int     i;

        for (i = 0; i < N; i++) {
                if (A[N] == 0) {
                        break;
                }
                printf ("A [%d] = %d\n", i, A[i]);
        }
}
```

What is the order of `strange1`? It is impossible to know without knowing what the contents of array `A` are: if there is always a zero element in the array, and the first zero element is always at a fixed location in this array (for example, the first location), then this function is $\Theta(1)$– it always takes the same amount of time to run, regardless of N. If we know that there *isn't* a zero element anywhere in the array, then this algorithm is $\Theta(N)$, since it visits every location in the array, and the array is of length N. If we don't know anything about the contents of the array, however, then we can't be sure what the order is, so we don't know what the true order of `strange1`(N) is. We know that the order is *no less* than order 1, so `strange1`$(N) = \Omega(1)$. Similarly, we know that the order is *no larger* than N, so `strange1`$(N) = O(N)$.

2.2 Finding the Big-O of a Function

Although the definitions given above completely define what the big-O of a function is, it is not always immediately obvious how to use them to actually discover the big-O of a function. Finding the order of many useful functions can be a challenging task, and there are even functions that have defied all attempts at a complete analysis! Luckily, since there has been so much work done in the area already, finding the big-O of many functions is simply a matter of finding a known result for a similar function and using it.

The following rules provide a jumping off point:

1. If $T_1(N) = O(f_1(N))$ and $T_2(N) = O(f_2(N))$, then:

   (a) $T_1(N) + T_2(N) = \mathbf{max}(O(f_1(N)), O(f_2(N)))$

   (b) $T_1(N) \times T_2(N) = O(f_1(N) \times f_2(N))$

2. If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(N^k)$.

   The fact that $T(N) = O(N^k)$ follows from the previous rule. Showing that $T(N) = \Theta(N^k)$ is slightly more complicated, but can be proven by showing that there is always a choice of c and n_0 that satisfies the definition of Θ.

3. $\log^k N = O(N)$, for any constant k.

   This is true, but of relatively limited use. Since $O(N)$ is an upper bound, it is OK to say that anything with a lower order is $O(N)$. This is handy as a last resort, when trying to find the big-O of a nasty function that includes logarithms, but whenever possible it is useful to draw a distinction between $O(N)$ and $O(\log^k N)$. For example, an algorithm that has a big-O of $O(N \log N)$ is (usually) much better than an algorithm that is $O(N^2)$.

2.2.1 The Dominance of Functions

Another method of finding the big-O of a function is to find the dominant term of the function, and find its order. The order of the dominant term will also be the order of the function.

The dominant term is the term that grows most quickly as n becomes large. Some rules of dominance include the following:

- Any exponential function of n dominates any polynomial function of n.
- Any polynomial function of n dominates any logarithmic function of n.
- Any logarithmic function of n dominates a constant term.
- A polynomial of degree k dominates a polynomial of degree l if and only if $k > l$.

There are additional rules that you can discover for yourself. The key is that term $x(n)$ dominates function $y(n)$ if and only if $x(n)/y(n)$ grows as n grows large.

Using these rules can sometimes make finding the order of a complicated function easy. For example, the function $f(n) = n^4 + 2^n + \log n$, is clumsy to manipulate algebraically. However, thanks to the rules of dominance, we know that we the 2^n term will dominate the order of this function, so we can simply find the order of 2^n, which is itself. Therefore, we can immediately say that $f(n) = O(2^n)$.

There are other methods that can be used find the order of functions. but we will not introduce them yet. Later in the semester, when we analyze several recursive algorithms, methods for analyzing the order of recursive functions will be introduced.

2.2.2 Properties of Orders

Since the orders of functions look like ordinary functions, it is tempting to treat them as such, and manipulate them as though they were ordinary functions. **This is usually a huge mistake.**

For example, let $f(n) = n^2$ and $g(n) = 2n^2$. It should be clear that the both $f(n)$ and $g(n)$ are $O(n^2)$. What is the big-O of $g(n) - f(n)$? If we compute the order of the difference as the difference of the orders, then we would mistakenly conclude that $(g(n) - f(n)) =$ the order of $f(n)$ less the order of $g(n)$, or $(n^2 - n^2) = 0$. This answer is obviously wrong. If we actually compute $g(n) - f(n)$, however, we arrive at the correct conclusion: $(g(n) - f(n)) = (2n^2 - n^2) = n^2 = O(n^2)$.

2.3 Big-O Notation and Algorithm Analysis

Now that we have seen the basics of big-O notation, it is time to relate this to the analysis of algorithms.

In our study of algorithms, nearly every function whose order we are interested in finding is a function that defines the quantity of some resource consumed by a particular algorithm in relationship to the parameters of the algorithm. (This function is often referred to as a *complexity* of the algorithm, or less frequently as the *cost function* of the algorithm.) This function is usually not the same as the algorithm itself, but is a property of the algorithm. For example, when we are analyzing an algorithm that multiplies two numbers, the functions we might be interested in are the relationships between the number of digits in each number and the length of time or amount of memory required by the algorithm.

Although big-O notation is a way of describing the order of a function, it is also often meant to represent the time complexity of an algorithm. This is sloppy use of the mathematics, but unfortunately not uncommon. (For example, at the bottom of page 22 in Weiss, a factorial function is described as being $O(N)$. Clearly, the factorial function itself is not $O(N)$, it is $O(N!)$ according to its definition. What is meant is that the particular algorithm given for computing the factorial of N requires $O(N)$ time to execute.) Usually it is clear from the surrounding context what the big-O refers to, but when it is not clear, please ask.

Note that this notation, like the orders themselves, doesn't tell us how quickly or slowly the algorithms actually execute for a given input. This information can be extremely important in practice– during the semester, we will study several algorithms that address the same problems and have the same order running time, but take substantially different amounts of time to execute.

Similarly, the order doesn't tell us how fast an algorithm will run for a small N. This may also be quite important in practice– during the semester, we will study at least one group of algorithms where the choice of the best algorithm depends on N– when N is small, one algorithm is best, but when N is large, a different algorithm is much better.

2.3.1 The Size of a Problem

Big-O notation is used to express an ordering property among functions. In the context of our study of algorithms, the functions are the amount of resources consumed by an algorithm when the algorithm is executed. This function is usually denoted $T(N)$. $T(N)$ is the amount of the resource (usually time or the count of some specific operation) consumed when the input to the algorithm is of size N.

Sometimes it is not obvious what the "size" of the input is, so here are few examples:

- **Multiplication**

  If we multiply two numbers together using the algorithm we learned in grade school, the number of single-digit multiplications and additions that we do depends on the number of digits in the two numbers being multiplied. (If we multiply two numbers together using a calculator, then the number of keystrokes we need to type also depends on the number of digits in the two numbers.) Therefore, for an algorithm that performs multiplication (or any other arithmetic operation) the "size" of the input is usually the number of digits in the input numbers.

  (As an aside, remember that most modern computers can perform arithmetic operations on reasonably-sized numbers (i.e. numbers that are 32 or 64 bits in length) in a single instruction, regardless of the number of non-zero digits in the numbers. Therefore, for most purposes, we will assume that arithmetic operations take a constant amount of time. There are many important applications that deal with numbers large enough so that this is not the case, however.)

- **Searching**

  If we have a set of N unknown items, arranged in an unknown order, and we want to do something involving all of them, then the size of the problem is N. For example, if we want to find the smallest element in an array of N numbers, then the size of the problem is N.

2.3.2 What Does It All Mean?

The big-O complexity of an algorithm is important, but it does not tell the entire story. Any honest and complete analysis of an algorithm should attempt to address the question of what the real cost of executing the algorithm is for real or typical problems.

For example, imagine that (through some some wonderful twist of fate), you must choose between job offers from two companies. The first company offers a contract that will double your salary every five years. The second company offers you a contract that gives you a raise of \$1000 per year. Your salary at the first company increases at $O(2^n)$, while your salary at the second company increases at a rate of $O(n)$. Which position would you choose?

From this information, it is impossible to say. If your starting salaries are the same at the two companies, then what the starting salary is will make all the difference (and if the first company gives you a starting salary of \$0, then you are really in trouble!). If the starting salary at the second company is much higher than the first company, then even the wonderful raises the first company offers might not actually make any difference before you retire. In this case, n isn't going to get too large (assuming an ordinary career), so the behavior of your salary as you become infinitely old is not an issue.