CMSC 202	**Asymptotic Analysis and**	Spring 1996
Sections 101, 102 & 103	**Big–Oh Notation**	Handout #25

The material presented herein was stolen wholesale from Tom Anastasio's handout of approximately the same name.

1 Complexity and Asymptotic Analysis

A programmer usually has a choice of data structures and algorithms to use. Choosing the best one for a particular job involves, among other factors, two important measures:

- *Time Complexity*: how much time will the program take?
- *Space Complexity*: how much storage will the program need?

A programmer will sometimes seek a tradeoff between space and time complexity. For example, a programmer might choose a data structure that requires a lot of storage to reduce the computation time of an algorithm. There is an element of art in making such tradeoffs, but the programmer must make the choice from an informed point of view. The programmer must have some verifiable basis on which to make the selection of a data structure or algorithm. Complexity analysis provides such a basis.

Complexity refers to the rate at which the storage or time grows as a function of the problem size. The absolute growth depends on the machine used to execute the program, the compiler used to construct the program, and many other factors. We would like to have a way of describing the inherent complexity of a program (or piece of a program), independent of machine/compiler considerations. This means that we must not try to describe the absolute time or storage needed. We must instead concentrate on a "proportionality" approach, expressing the complexity in terms of its relationship to some known function. This type of analysis is known as *asymptotic analysis*.

2 Big–Oh

Asymptotic analysis is based on the idea that as the problem size grows, the complexity can be described as a simple proportionality to some known function. This idea is incorporated in the "Big Oh" notation for asymptotic performance:

Definition: Suppose that $T(n)$ is some function whose complexity we wish to characterize. We say that $T(n) = O(f(n))$ if and only if there are constants c_0 and n_0 such that $T(n) \leq c_0 f(n)$ for all $n \geq n_0$.

The expression $T(n) = O(f(n))$ is read as "T of n is[1] Big–Oh of f of n." Big–Oh is sometimes said to describe an *upper–bound* on the complexity. Other forms of asymptotic analysis ("BigOmega", "LittleOh", "BigTheta") are similar in spirit to Big–Oh, but will not be discussed in this handout.

If a function $T(n) = O(f(n))$, then as n grows, eventually the value $cf(n)$ will exceed the value of $T(n)$. "Eventually" means "after n exceeds some value." Does this really mean anything useful? We might say (correctly) that $x^2 + 2x = O(x^{25})$, but we don't get a lot of information from that; x^{25} is simply too big. When we use Big–Oh analysis, we implicitly agree that the function $f(n)$ we choose is the smallest one we can find that still satisfies the definition. It is correct *and* meaningful to say $x^2 + 2x = O(x^2)$; this tells us something about the growth pattern of the function $x^2 + 2x$, namely that the x^2 term will dominate the growth as n increases.

The following functions are often encountered in computer science Big–Oh analysis (T stands for Time—all examples henceforth will implicitly refer to time complexity):

- $T(n) = O(1)$. This is called constant growth. $T(n)$ does not grow at all as a function of n; it is a constant. It is pronounced "Big–Oh of one." For example, array access has this characteristic. `A[i]` takes the same time no matter how big `A` is.
- $T(n) = O(\lg(n))$. This is called logarithmic growth. $T(n)$ grows as the base 2 logarithm of n (actually, the base doesn't matter, it's just traditional to use base–2 in computer science). It is pronounced "Big–Oh of log n." For example, binary search has this characteristic.
- $T(n) = O(n)$. This is called linear growth; $T(n)$ grows linearly with n. It is pronounced "Big–Oh of n." For example, looping over all the elements in a one–dimensional array would be an $O(n)$ operation.
- $T(n) = O(n \lg(n))$. This is called "n log n" growth. $T(n)$ grows as n times the base 2 logarithm of n. It is pronounced "Big–Oh of n log n." For example, heapsort and mergesort have this characteristic. In fact, no sorting algorithm that uses comparison between elements can be faster than $O(n \lg(n))$.
- $T(n) = O(n^k)$. This is called polynomial growth. $T(n)$ grows as the k–th power of n. In computer science, k is rarely greater than 3. For example, selection sort is an $O(n^2)$ algorithm. It is pronounced "Big–Oh of n squared."

[1] We do not say 'equals' for reasons that are explained on Page 3.

- $T(n) = O(2^n)$. This is called exponential growth. $T(n)$ grows exponentially. It is pronounced "Big–Oh of 2 to the n." Exponential growth is the most–feared growth pattern in computer science; algorithms that grow this way are basically useless for anything but very small problems.

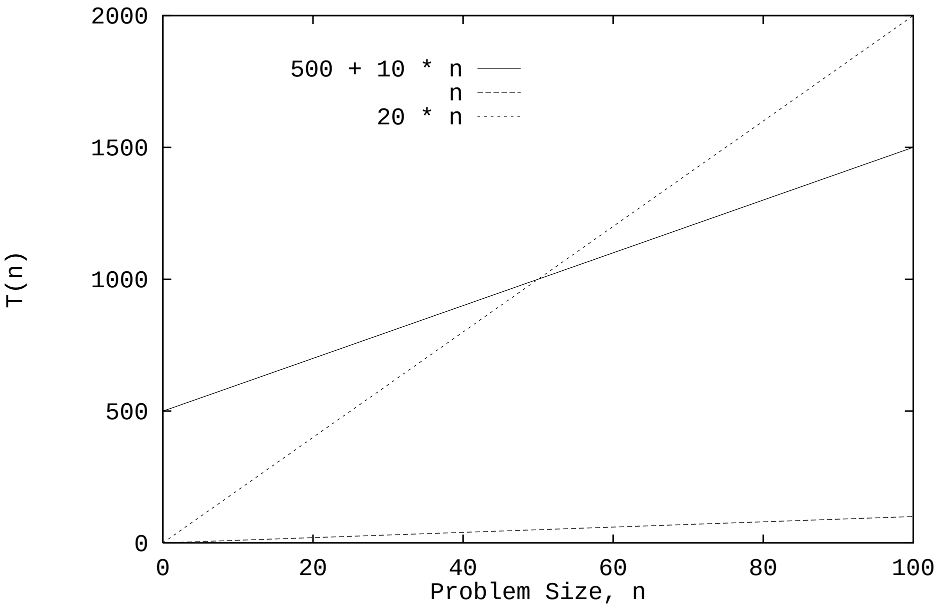
The growth patterns above have been listed in order of increasing "size." That is,

$$O(1) = O(\lg(n)) = O(n) = O(n\lg(n)) = O(n^2) = O(n^3) = O(2^n)$$

Note that it is not true that if $g(n) = O(f(n))$ then $f(n) = O(g(n))$. The "=" sign does not mean equality in the usual algebraic sense—that's why it is pronounced "$g(n)$ *is* Big–Oh of $f(n)$," and not "$g(n)$ *equals* Big–Oh of $f(n)$."

2.1 Example

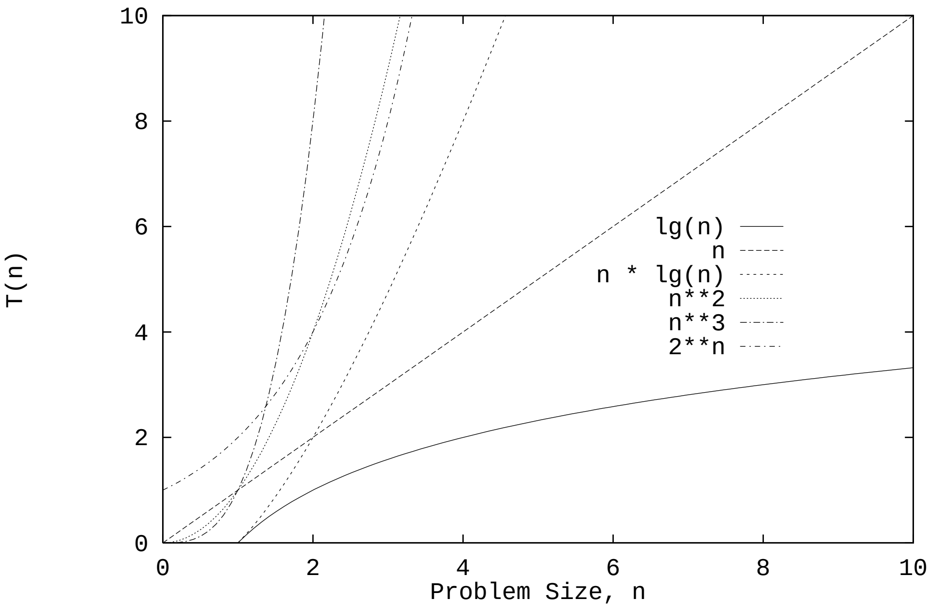
Suppose we have a program that takes some constant amount of time to set up, then grows linearly with the problem size n. The constant time might be used to prompt the user for a filename and open a file. Neither of these operations is dependent on the amount of data in the file. After these setup operations, we read the data from the file and do something with them (say print them). The amount of time required to read the file is certainly proportional to the amount of data in the file. We let n be the amount of data. This program has time complexity of $O(n)$. To see this, let's assume that the setup time is really long, say 500 time units. Let's also assume that the time taken to read the data is $10n$, 10 time units for each data point read. The following graph shows the function $500 + 10n$ plotted against n, the problem size. Also shown are the functions n and $20n$.

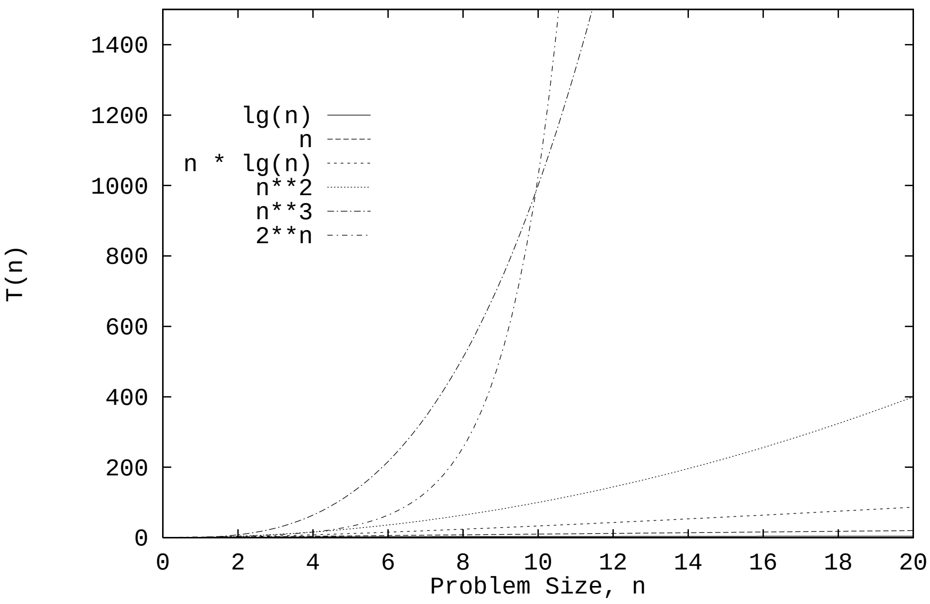


Note that the function n will *never* be larger than the function $500 + 10n$, no matter how large n gets. However, there are constants c_0 and n_0 such that $500 + 10n \leq c_0 n$ when $n \geq n_0$. One choice for these constants is $c_0 = 20$ and $n_0 = 50$. Therefore, $500 + 10n = O(n)$. (There are, of course, other choices for c_0; any value of $c_0 > 10$ will do.)

2.2 Example

Here we look at the functions $\lg(n)$, n, $n \lg(n)$, n^2, n^3, and 2^n to get some idea of their relative "size." In the first graph, it looks like n^2 and n^3 are larger than 2^n; they're not. The second graph shows the same data on an expanded scale. Clearly $2^n > n^2$ when $n > 4$ and $2^n > n^3$ when $n > 10$.


10
8
6
4
2
0
0
2
4
6
8
10
T(n)
Problem Size, n
lg(n)
n
n * lg(n)
n**2
n**3
2**n


1400
1200
1000
800
600
400
200
0
0
2
4
6
8
10
12
14
16
18
20
T(n)
Problem Size, n
lg(n)
n
n * lg(n)
n**2
n**3
2**n

2.3 Example

The following table shows how long it would take to perform $T(n)$ steps on a computer that does 1 billion steps/second.

n	$T(n) = n$	$T(n) = n\lg(n)$	$T(n) = n^2$	$T(n) = n^3$	$T(n) = 2^n$
5	.005 μs	.01 μs	.03 μs	.13 μs	.03 μs
10	.01 μs	.03 μs	.1 μs	1 μs	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	1 ms
50	.05 μs	.28 μs	2.5 μs	125 μs	13 days
100	.1 μs	.66 μs	10 μs	1 ms	4×10^{13} years

Notice that when $n \geq 50$, the compute time for $T(n) = 2^n$ has started to become too large to be practical. This is most certainly true when $n \geq 100$. Even if we were to increase the speed of the machine a million–fold, 2^n for $n = 100$ would be 40,000,000 years, a bit longer than you might want to wait for an answer.

2.4 Example

As a final example, we look at the `list` implementation of `queue`. In this implementation, each `enqueue` operation requires traversing the entire list to find the end, then "attaching" the new item at the end. What is the asymptotic time complexity of this operation as a function of n, the number of items in the `queue`? First of all, to find the end of the list, we must traverse the entire list. This takes n operations (one `rest` operation for each item on the list). Therefore, a single `enqueue` operation has asymptotic complexity $T(n) = O(n)$.

But, suppose we enqueue n items, one after another? The asymptotic complexity of such a series of insertions will be $T(n) = O(n^2)$. You may object that the list is not n long until all n items have been enqueued. Look at it this way; the first item takes one operation, the second takes two operations, the third takes three operations, etc. Therefore, the total number of operations to `enqueue` n items is

$$1 + 2 + 3 + \ldots + n$$

We can express this as

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2} = O(n^2)$$

The implementation that provides $O(1)$ performance for each `enqueue` operation allows n items to be `enqueued` in $O(n)$ time. This is significantly better than $O(n^2)$ (check the chart for $n = 100$, for instance).

3 Big–Oh Does Not Tell the Whole Story

Suppose you have a choice of two approaches to writing a program. Both approaches have the same asymptotic performance (for example, both are $O(n \lg(n))$). Why select one over the other, they're both the same, right? They may not be the same. There is this small matter of the constant of proportionality.

Suppose algorithms `A` and `B` have the same asymptotic performance, $T_A(n) = T_B(n) = O(g(n))$. Now suppose that `A` does ten operations for each data item, but algorithm `B` only does three. It is reasonable to expect `B` to be faster than `A` even though both have the same asymptotic performance. The reason is that asymptotic analysis ignores constants of proportionality.

As a specific example, let's say that algorithm `A` is

```
{
  set up the algorithm, taking 50 time units;
  read n elements into array A;  /* 3 units per element */
  for (i = 0; i < n; i++)
    {
      do operation1 on A[i];  /* takes 10 units */
      do operation2 on A[i];  /* takes  5 units */
      do operation3 on A[i];  /* takes 15 units */
    }
}
```

Let's now say that algorithm `B` is

```
{
  set up the algorithm, taking 200 time units;
  read n elements into array A;  /* 3 units per element */
  for (i = 0; i < n; i++)
    {
      do operation1 on A[i];  /* takes 10 units */
      do operation2 on A[i];  /* takes  5 units */
    }
}
```

Algorithm `A` sets up faster than `B`, but does more operations on the data. The execution time of `A` and `B` will be

$$T_A(n) = 50 + 3 * n + (10 + 5 + 15) * n = 50 + 33 * n$$

and

$$T_B(n) = 200 + 3 * n + (10 + 5) * n = 200 + 18 * n$$

respectively.

The following graph shows the execution time for the two algorithms as a function of n. Algorithm `A` is the better choice for small values of n. For values of $n > 10$, algorithm `B` is the better choice. Remember that both algorithms have time complexity $O(n)$.