Prof. Sergio A. Alvarez
Fulton Hall 410–B
Computer Science Department
Boston College
Chestnut Hill, MA 02467 USA

http://www.cs.bc.edu/∼alvarez/
alvarez@cs.bc.edu
voice: (617) 552-4333
fax: (617) 552-6790

CS383, Algorithms
Notes on Asymptotic Time Complexity

We discuss basic concepts related to measuring and comparing the time efficiency of algorithms.

1 Growth Rates; Asymptotic Notation

We introduce notation that expresses the rate of growth of a real-valued function $f : \mathrm{N} \to \mathrm{R}$ over the set of natural numbers. Such functions will later be used to describe the time required by an algorithm to complete its computation.

1.1 Big O

Let f and g be real-valued functions over the set of natural numbers, with $g(n) > 0$ for all n. We say that "f is big O of g" and write

$$f(n) = O(g(n))$$

if there exists a real constant C (independent of n) such that

$$f(n) \leq Cg(n) \text{ for all } n$$

Thus, f is big O of g if f is bounded above by a constant multiple of g. Intuitively, $f(n) = O(g(n))$ means that f grows no faster than g as the index n grows. For example, the function $f(n) = n^2$ is big O of the function $g(n) = n^3$, while it is *not* true that g is big O of f in this case. Every function is big O of itself.

The requirement that the inequality in the above definition of big O hold for all values of n is sometimes relaxed so that it only need apply for all $n \geq n_0$, for some number n_0. This requirement is superfluous if $g(n) > 0$ for all n, because if $f(n) \leq Cg(n)$ for all $n \geq n_0$, one can ensure that $f(n) \leq C'g(n)$ for the few remaining values $n = 0...n_0 - 1$ by enlarging the constant C to C' if need be: just pick C' to be as large as the largest of the n_0 values $f(0)/g(0)...f(n_0 - 1)/g(n_0 - 1)$ (and no smaller than the original value of C).

Examples.

1. $n^p = O(n^q)$ whenever $0 < p \leq q$. *Proof.* This follows from the fact that $n^p \leq n^q$ for all $n \geq 0$ if $0 < p \leq q$. Thus, the requirement in the definition of big O is satisfied with $C = 1$.

2. $n^p = O(b^n)$ for any $p > 0$ and $b > 0$. In words: polynomials grow slower than exponentials. *Proof.* This is a little trickier. We will show that

$$\lim_{n \to \infty} \frac{n^p}{b^n} = 0$$

Before we compute the limit, let's ask: would that be enough? Suppose for argument's sake that we knew the limit to be 0. That would imply in particular that there is some value n_0 such that

$$\frac{n^p}{b^n} \leq 1 \text{ for all } n \geq n_0$$

This would immediately satisfy the definition of big O as desired. With that guarantee in mind, we proceed to compute the limit. The limit is of the form ∞/∞, so we apply L'Hôpital's rule from calculus, which states that the limit in this situation is the same as the limit of the ratio of the derivatives of the numerator and denominator, where the argument, n, is interpreted as a continuous real variable. Rewriting b^n in terms of the natural logarithm base, e, we then find:

$$\lim_{n\to\infty} \frac{n^p}{b^n} = \lim_{n\to\infty} \frac{p\ n^{p-1}}{\ln b\ b^n}$$

Notice that the exponent in the numerator goes down by 1. Iterating this process of differentiation, if necessary, we eventually arrive at a quotient for which the numerator has a negative exponent and therefore approaches 0 as n approaches ∞. This proves that the limit is 0.

3. $\log n = O(n^p)$ for any $p > 0$. This is a good exercise. Follow the argument used in the preceding example. We'll describe an alternate technique to deal with this and similar situations below, at the end of this section.

1.2 Big Ω

Just as big O notation describes upper bounds on a function $f(n)$, there is a notation designed to describe lower bounds. We say that $f(n)$ is big Omega of $g(n)$, and write

$$f(n) = \Omega(g(n)),$$

when there exists a real constant $C > 0$ such that

$$f(n) \geq Cg(n) \text{ for all } n$$

Notice that C must be strictly positive; otherwise, one could pick $C = 0$ and have a trivial lower bound whenever f is non-negative, no matter what g is. Intuitively, $f(n) = \Omega(g(n))$ means that f grows asymptotically at least as fast as g does.

Exercise. Show that $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

1.3 Big Θ

Big Theta notation is used to describe pairs of functions that have equivalent asymptotic growth rates. We say that $f(n)$ is big Theta of $g(n)$, and write

$$f(n) = \Theta(g(n)),$$

when both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ simultaneously.

Exercise. Write a direct definition of big Θ notation in terms of constants and inequalities, in the spirit of the big O and big Ω definitions given above.

1.4 Finding asymptotic relationships through substitution

Consider the final example in the subsection on big O notation, above. We wish to show that

$$\log n = O(n^p)$$

It is possible to do this by using the L'Hôpital's rule argument that we used to show that

$$n^p = O(b^n)$$

There is, however, a direct way to show that the latter bound is actually already contained in the former as a special case. Start from the $\log n$ on the left hand side of the target bound. The function here is inconvenient because we may not have proven any bounds for it up to this point. So we simply declare the expression $\log n$ to be a new variable, u, thus eliminating the logarithm altogether:

$$u = \log n$$

We'll need to rewrite the target bound in terms of the new variable. This requires expressing n in terms of u, which, fortunately, is pretty easy:

$$n = 2^u,$$

assuming that the logarithm is in base 2 (if it isn't, just replace 2 with the appropriate base). The target bound now becomes:

$$u = O(2^{up})$$

This is an asymptotic growth statement about a function of u. But notice that the latter big O statement just compares a polynomial of u on the left (the simplest possible one) with an exponential of u on the right. This is exactly the case that we already addressed using L'Hôpital's rule before! Therefore, we see that the target statement is true, merely by referring back to that previous result.

Substitution Rule. Here is a more general statement of the substitution trick that we used above. Let $f, g : \mathrm{R} \to R$ be two functions. Suppose there is a strictly increasing function $\phi : \mathrm{R}^+ \to \mathrm{R}^+$ such that

$$f(\phi^{-1}(u)) \leq Cg(\phi^{-1}(u))$$

for some finite constant C and all values of u in the range of ϕ. Then

$$f(n) = O(g(n))$$

Proof. Assume that ϕ is as in the statement. Then ϕ is invertible on its range because it is one-to-one. Assume also that C is a finite constant such that

$$f(\phi^{-1}(u)) \leq Cg(\phi^{-1}(u))$$

for all values of u in the range of ϕ. If n is any non-negative integer, let $u = \phi(n)$. Then the preceding inequality becomes

$$f(n) \leq Cg(n)$$

which shows that

$$f(n) = O(g(n)),$$

as desired.

Exercise. Use the substitution trick to compare the asymptotic growth rates of the functions $\log n^{\log n}$ and n^2.

2 Time Complexity

Speed is a central issue in the study of algorithms. Actual time spent by a program as it performs a particular computation is a key concern. However, the time may depend on factors other than algorithm design, including the programming language used to implement the algorithm and the hardware on which the program is running. We discuss ways of measuring the intrinsic efficiency of the underlying algorithm itself while abstracting away such details.

2.1 Computational tasks, instances

Every algorithm is intended to address a particular *computational task*. A computational task is defined by its input-to-output specification: the collection of all pairs (x, y), where x is a possible input to the algorithm and y a desired output of the algorithm on input x. Each such pair (x, y) is an *instance* of the computational task. For example, if the computational task is sorting an array of integers, then each instance consists of a pair (a, a'), where a is an array of integers and a' is a sorted permutation (rearrangement) of a.

Size of an instance. Not all instances are created equal in terms of difficulty. For example, one would expect a random array to be harder to sort than one in which a single swap of two elements would complete the task. At the most basic level, such relative difficulty is determined by the size of the input objects (e.g., the length of the array to be sorted). We assume that a notion of size is available for the computational task of interest in each case. The size could be the number of elements of an array for a sorting task, or the number of digits of an integer in base 10 for a primality test, or the number of cities to be visited for a traveling salesman problem. A basic goal of complexity analysis will be to determine the computational cost of an algorithm as a function of the size of the input instance.

2.2 Notions of time complexity

Fix a computational task, as well as a measure of the size of an instance of this task. For a given algorithm that addresses the target task, we seek to measure the computational cost of the algorithm as a function of the size of the input instance.

Basic computational steps. In order to make the complexity measure independent of implementation language and hardware platform details, we count the number of basic computational steps required by a procedure. The notion of basic step is imprecise. However, in a given context it should be such that one basic step requires approximately the same amount of processing time independently of the size of the input instance. For example, evaluating a boolean expression is a reasonable candidate for a basic computational step, as is assignment of a memory address to a pointer variable, assuming that addresses are represented as memory words of a fixed size. Making a copy of an array (as opposed to passing the array's starting address) is a poor choice for a basic computational step because the time required for copying can reasonably be expected to be proportional to the length of the array, which is not independent of the size of the array.

Let A be an algorithm. For each input instance x, we denote by $A(x)$ the result of running algorithm A on input x; the pair $(x, A(x))$ should be one of the pairs that appear in the input-output specification of the target computational task for the algorithm. We denote by $t(x \to A(x))$ the number of basic computational steps taken by A in computing the result $A(x)$ on input x.

2.2.1 Worst-case time complexity

The worst-case time complexity of algorithm A is the function $\bar{t}_A : \mathrm{N} \to \mathrm{N}$ that counts the maximum possible number of basic computational steps used by A for an input of a given size:

$$\bar{t}_A(n) = \max_{\mathrm{size}(x)=n} t(x \to A(x))$$

2.2.2 Average time complexity

Worst-case complexity provides an upper bound on the cost of processing inputs of a given size. However, the number of steps required by "typical" input instances may be considerably less. An example is the task of searching for a value in an unordered list of length n. If the target value does not appear on the list, all n list elements will be examined during the search. On the other hand, if the target value *does* appear on the list, then the number of steps required will depend on the target value's position within the list. The number of elements examined will range from 1 if the target appears at the very beginning of the list, to n, if it appears at the very end. In the absence of additional information, it is reasonable to assume that all such situations are equally likely to occur. *On average*, the number of elements examined will be roughly $n/2$. In this case, this average is of order n, just like the worst-case complexity. We will see that there are algorithms for which the average complexity has a lower asymptotic growth rate than the worst case complexity.

The average time complexity of A is the function $\tilde{t}_A : \mathrm{N} \to \mathrm{N}$ that counts the average possible number of basic computational steps used by A for an input of a given size. Assume that a probability distribution on the set of input instances is specified, so that the relative likelihoods of different input instances is known. The running time of A on input x may then be seen as a random variable. The average time complexity of algorithm A (with respect to the input probability distribution) is the average value of that random variable:

$$\tilde{t}_A(n) = E_{\mathrm{size}(x)=n} t(x \to A(x))$$

The letter E here stands for *expectation*, which is the term that probabilists use for the average value of a random variable.

2.3 Examples of time complexity analysis

1. Algorithm 1 is an example of an algorithm for which the worst case and average case complexities have different asymptotic growth rates. We take the length of the input array to be the measure of instance size in terms of which the time complexity is to be expressed.

   In the worst case, algorithm 1 will perform n passes through the while loop before returning a value. We assume that the comparison $i \leq n$ takes time proportional to the number of digits of n, which is $\Theta(\log n)$, that the parity check on $a[i]$ takes time $\Theta(1)$, and that incrementing i by 1, as a special case of addition, takes time $\Theta(\log i)$. Summing over all values of i between 1 and n, this yields a total time $\Theta(n \log n)$ in the worst case.

Algorithm 1: First Odd Array Element
Input: An array of integers $a[1...n]$.
Output: The smallest integer index value i such that $a[i]$ is odd if a contains at least one odd value; the value $n+1$ otherwise.
INDEXOFFIRSTODD(a)
(1) $i = 1$
(2) **while** $i \leq n$ and $a[i]$ is even
(3) $i = i + 1$
(4) **return** i

On the other hand, assuming that odd and even elements are equally likely to occur, only two positions will need to be examined on average before the algorithm finds an odd element and returns a value. The comparisons $1 \leq a$ and $2 \leq n$ may reasonably be assumed to take time $\Theta(1)$ (even if n is very large, efficient comparison will proceed from the most significant digit downward, which will yield an answer immediately). Thus, the algorithm's average case time complexity is $\Theta(1)$.

2. The time complexity of a recursive algorithm can often be bounded by bounding the recursion depth and then bounding the number of steps spent at each level of the recursion. We illustrate this approach by analyzing the following algorithm.

Algorithm 2: Divide and Conquer Modular Exponentiation
Input: A real number b and integers $p \geq 0$ and $n > 1$.
Output: The result b^p of multiplying p copies of b together (mod n).
RECPOW(b, p, n)
(1) **if** p equals 0 **then return** 1
(2) $q =$ RECPOW$(b, \lfloor p/2 \rfloor, n)$
(3) **if** p is even **then return** $q * q \pmod{n}$
(4) **else return** $b * q * q \pmod{n}$

First, let's agree on a notion of size for the input instances. The inputs are numbers, so we'll use the number of digits as a measure of size. For real numbers, we assume that we are using finite precision approximations, and count the number of significant digits (or bits) in these approximations. We let d denote the largest number of digits among the three inputs.

We bound the recursion depth in terms of the number of digits, d. Since the value of p is divided by 2 with each recursive call (in integer arithmetic), and since the base case occurs when p equals 0, the recursion depth is the smallest value k^* of k that satisfies

$$\frac{p}{2^k} < 1$$

This value equals

$$k^* = \lfloor 1 + \log_2 p \rfloor$$

To see this, consider first the case in which p is a power of 2, $p = 2^m$. An induction argument in the exponent m proves the above expression in this case. In the general case, p is between two powers of 2, and the recursion depth is the same as for the lower of these two powers. This shows that the recursion depth is $\Theta(\log p)$, or, equivalently, $\Theta(d)$.

We now bound the number of steps per level of the recursion. Either one or two multiplications are carried out depending on whether the exponent is even or odd. This takes time $O(d^2)$, assuming that the "standard grade school algorithm" is used for multiplication. Likewise, computation of modular remainders may be completed in time $O(d^2)$ as discussed in class. We assume that division of p by 2 takes time $O(d)$ because of the underlying binary representation at the machine level. Finally, evaluation of the boolean expressions, assignments, return statements, and parameter passing can all be assumed to take time $O(d)$. Therefore, the total time at each level is $O(d^2)$.

Combining the above bounds on the recursion depth and the time complexity per level, we conclude that the total time complexity of the recursive exponentiation algorithm is $O(d^3)$. Both the worst case and average case complexities satisfy the $O(d^3)$ upper bound. The bound on the recursion depth in the preceding analysis a tight one: $\Theta(d)$. We loosened our grip only in analyzing the work per level. The dominant time is associated with multiplication. The reason for having used a big O bound for the multiplication time rather than a big Θ bound is that there are certain "easy" cases for multiplication, namely multiplication by powers of 2, which can be dealt with by linear time hardware operations if binary representation is used at the machine level.