UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Spring 1998

Lecture Notes #14*

1 Time Complexity

The obvious way to answer to the question "How fast does such-and-such a program run?" is to use something like the UNIX `time` command to find out directly. There are various possible objections to this easy answer. The time required by a program is a function of the input, so presumably we have to time several instances of the command and extrapolate the result. Some programs, however, behave fine for *most* inputs, but sometimes take a very long time; how do we report (indeed, how can we be sure to notice) such anomalies? What do we do about all the inputs for which we have no measurements? How do we validly apply results gathered on one machine to another machine?

The trouble with measuring raw time is that the information is precise, but limited: the time for *this* input on *this* configuration of *this* machine. On a different machine whose instructions take different absolute or relative times, the numbers don't necessarily apply. Indeed, suppose we compare two different programs for doing the same thing on the same inputs and the same machine. Program A may turn out faster than program B. This does *not* imply, however, that program A will be faster than B when they are run on some other input, or on the same input, but some other machine.

In mathematese, we might say that a raw time is the value of a function $C_r(I, P, M)$ for some particular input I, some program P, and some "platform" M (*platform* here is a catchall term for a combination of machine, operating system, compiler, and runtime library support). We can make the figure a little more informative by summarizing over *all* inputs of a particular size

$$C_w(N, P, M) = \max_{|I|=N} C_r(I, P, M),$$

where $|I|$ denotes the "size" of input I. How one defines the size depends on the problem: if I is an array to be sorted, for example, $|I|$ might denote `I.length`. We say that C_w measures *worst-case time* of a program. Of course, since the number of inputs of a given size could be very large (the number of arrays of 5 `int`s, for example, is $2^{160} > 10^{48}$), we can't directly measure C_w, but we can perhaps estimate it with the help of some analysis of P. By knowing worst-case times, we can make conservative statements about the running time of a program: if the worst-case time for input of size N is T, then we are guaranteed that P will consume no more than time T for *any* input of size N.

But of course, it always possible that our program will work fine on most inputs, but take a really long time on one or two (unlikely) inputs. In such cases, we might claim that C_w is too harsh a summary measure, and we should really look at an *average* time. Assuming all values of the input, I, are equally likely, that is

$$C_a(N, P, M) = \frac{\sum_{|I|=N} C_r(I, P, M)}{N}$$

Fair this may be, but it is often (usually) very hard to compute. In this course, therefore, I will say very little about average cases, leaving that to CS170 [plug].

We've summarized over inputs by considering worst-case times; now let's consider how we can summarize over machines. Just as summarizing over inputs required that we give up some information—namely, performance on particular inputs—so summarizing over machines requires that we give up information on precise performance on particular machines. Suppose that two different models of computer are running (different translations of) the same program, performing the same steps in the same order. Although they run at different speeds, and possibly execute different numbers of instructions, the speeds at which they perform any particular step tend to differ by some constant factor. By taking the largest and smallest of these constant factors, we can put bounds around the difference in their overall execution times. (The argument is not really this simple, but for our purposes here, it will suffice.) That is, the timings of the same program on any two platforms will tend to differ by no more than some constant factor over all possible inputs. If we can nail down the timing of a program on one platform, we can use it for all others, and our results will "only be off by a constant factor."

But of course, 1000 is a constant factor, and you would not normally be insensitive to the fact that Brand X program is 1000 times slower than Brand Y. There is, however, an important case in which this sort of characterization is useful: namely, when we are trying to determine or compare the performance of *algorithms*—idealized procedures for performing some task. The distinction between algorithm and program (a concrete, executable procedure) is somewhat vague. Most higher-level programming languages allow one to write programs that look very much like the algorithms they are supposed to implement. The distinction lies in the level of detail. A procedure that is cast in terms of operations on "sets," with no specific implementation given for these sets, probably qualifies as an algorithm. When talking about idealized procedures, it doesn't make a great deal of sense to talk about the number of seconds they take to execute. Rather, we are interested in what I might call the *shape* of an algorithm's behavior: such questions as "If we double the size of the input, what happens to the execution time?" Given that kind of question, the particular *units* of time (or space) used to measure the performance of an algorithm are unimportant—constant factors don't matter.

If we only care about characterizing the speed of an algorithm to within a constant factor, other simplifications are possible. We need no longer worry about the timing of each little statement in the algorithm, but can measure time using any convenient "marker step." For example, to do decimal multiplication in the standard way, you multiply each digit of the multiplicand by each digit of the multiplier and then add one or two digits for each of these multiplications. Counting just the one-digit multiplications, therefore, will give you the time within a constant factor, and these multiplications are very easy to count (the product of numbers of digits in the operands).

Another characteristic assumption in the study of *algorithmic complexity* (i.e., the time or memory consumption of an algorithm) is that we are interested in *typical* behavior of an idealized program over the entire set of possible inputs. Idealized programs, of course, being ideal, can operate on inputs of any possible size, and most "possible sizes" in the ideal world of mathematics

are extremely large. Therefore, in this kind of analysis, it is traditional not to be interested in the fact that a particular algorithm does very well for small inputs, but rather to consider its behavior "in the limit" as input gets very large. For example, suppose that one wanted to analyze algorithms for computing π to any number of decimal places. I can make *any* algorithm look good for inputs up to, say, 1,000,000 by simply storing the first 1,000,000 digits of π in an array and using that to supply the answer when 1,000,000 or fewer digits are requested. If you paid any attention to how my program performed for inputs up to 1,000,000, you could be seriously misled as to the cleverness of my algorithm. Therefore, when studying algorithms, we look at their *asymptotic behavior*—how they behave as they input size goes to infinity.

The result of all these considerations is that in considering the time complexity of algorithms, we may choose any particular machine and count any convenient marker step, and we try to find characterizations that are true asymptotically—out to infinity. This implies that our typical complexity measure for algorithms will have the form $C_w(N, A)$—meaning "the worst-case time over all inputs of size N of algorithm A (in some units)." Since the algorithm will be understood in any particular discussion, we will usually just write $C_w(N)$ or something similar. So what we need to describe algorithmic complexity is a way to characterize the asymptotic behavior of functions.

2 Asymptotic complexity analysis and order notation

As it happens, there is a convenient notational tool—known collectively as *order notation* for "order of growth"—for describing the asymptotic behavior of functions. It may be (and is) used for any kind of integer- or real-valued function—not just complexity functions.

We write

$$f(n) \in O(g(n))$$

(aloud, this is "$f(n)$ is in big-Oh of $g(n)$") to mean that the function f is eventually bounded by some multiple of $|g(n)|$. More precisely,

$$f(n) \in O(g(n)) \text{ iff } |f(n)| \leq K \cdot |g(n)|, \text{ for all } n > M,$$

for some constants $K > 0$ and M. That is, $O(g(n))$ is the *set* of functions that "grow no more quickly than" $|g(n)|$ does as n gets sufficiently large. Somewhat confusingly, $f(n)$ here does not mean "the result of applying f to n," as it usually does. Rather, it is to be interpreted as the *body of a function* whose parameter is n. Thus, we often write things like $O(n^2)$ to mean "the set of all functions that grow no more quickly than the square of their argument."

Saying that $f(n) \in O(g(n))$ gives us only an *upper bound* on the behavior of f. Accordingly, we define $f(n) \in \Omega(g(n))$ iff for all $n > M$, $|f(n)| \geq K|g(n)|$ for $n > M$, for some constants $K > 0$ and M. That is, $\Omega(g(n))$ is the set of all functions that "grow at least as fast as" g beyond some point. A little algebra suffices to show the relationship between $O(\cdot)$ and $\Omega(\cdot)$:

$$|f(n)| \geq K|g(n)| \equiv |g(n)| \leq (1/K) \cdot |f(n)|$$

so

$$f(n) \in \Omega(g(n)) \Longleftrightarrow g(n) \in O(f(n))$$

Because of our cavalier treatment of constant factors, it is possible for a function $f(n)$ to be bounded both above and below by another function $g(n)$: $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$. For brevity, we write $f(n) \in \Theta(g(n))$, so that $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Just because we know that $f(n) \in O(g(n))$, we don't necessarily know that $f(n)$ gets much smaller than $g(n)$, or even (as illustrated in Figure ??a) that it is ever smaller than $g(n)$. We

$f(n)$	*Is contained in*	*Is* not *contained in*
$1,\ 1+1/n$	$O(10000),\ O(\sqrt{n}),\ O(n),$ $O(n^2),\ O(\lg n),\ O(1-1/n)$ $\Omega(1),\ \Omega(1/n),\ \Omega(1-1/n)$ $\Theta(1),\ \Theta(1-1/n)$ $o(n),\ o(\sqrt{n}),\ o(n^2)$	$O(1/n),\ O(e^{-n})$ $\Omega(n),\ \Omega(\sqrt{n}),\ \Omega(\lg n),\ \Omega(n^2)$ $\Theta(n),\ \Theta(n^2),\ \Theta(\lg n),\ \Theta(\sqrt{n})$ $o(100+e^{-n}),\ o(1)$
$\log_k n,\ \lfloor\log_k n\rfloor,$ $\lceil\log_k n\rceil$	$O(n),\ O(n^\epsilon),\ O(\sqrt{n}),\ O(\log_{k'} n)$ $O(\lfloor\log_{k'} n\rfloor),\ O(n/\log_{k'} n)$ $\Omega(1), \Omega(\log_{k'} n), \Omega(\lfloor\log_{k'} n\rfloor)$ $\Theta(\log_{k'} n),\ \Theta(\lfloor\log_{k'} n\rfloor),$ $\Theta(\log_{k'} n+1000)$ $o(n),\ o(n^\epsilon)$	$O(1)$ $\Omega(n^\epsilon), \Omega(\sqrt{n})$ $\Theta(\log^2_{k'} n), \Theta(\log_{k'} n+n)$
$n,\ 100n+15$	$O(.0005n-1000),\ O(n^2),$ $O(n\lg n)$ $\Omega(50n+1000),\ \Omega(\sqrt{n}),$ $\Omega(n+\lg n),\ \Omega(1/n)$ $\Theta(50n+100),\ \Theta(n+\lg n)$ $o(n^3),\ o(n\lg n)$	$O(10000),\ O(\lg n),$ $O(n-n^2/10000),\ O(\sqrt{n})$ $\Omega(n^2),\ \Omega(n\lg n)$ $\Theta(n^2),\ \Theta(1)$ $o(1000n),\ o(n^2\sin n)$
$n^2,\ 10n^2+n$	$O(n^2+2n+12),\ O(n^3),$ $O(n^2+\sqrt{n})$ $\Omega(n^2+2n+12),\ \Omega(n),\ \Omega(1),$ $\Omega(n\lg n)$ $\Theta(n^2+2n+12),\ \Theta(n^2+\lg n)$	$O(n),\ O(n\lg n),\ O(1)$ $o(50n^2+1000)$ $\Omega(n^3),\ \Omega(n^2\lg n)$ $\Theta(n),\ \Theta(n\cdot\sin n)$
n^p	$O(p^n),\ O(n^p+1000n^{p-1})$ $\Omega(n^{p-\epsilon}),$ $\Theta(n^p+n^{p-\epsilon})$ $o(p^n),\ o(n!),\ o(n^{p+\epsilon})$	$O(n^{p-1}),\ O(1)$ $\Omega(n^{p+\epsilon}),\ \Omega(p^n)$ $\Theta(n^{p+\epsilon}),\ \Theta(1)$ $o(p^n+n^p)$
$2^n,\ 2^n+n^p$	$O(n!),\ O(2^n-n^p),\ O(3^n),\ O(2^{n+p})$ $\Omega(n^p),\ \Omega((2-\delta)^n),\ \Omega(n2^n)$ $\Theta(2^n+n^p)$ $o(n2^n),\ o(n!),\ o(2^{n+\epsilon}),\ o((2+\epsilon)^n)$	$O(n^p),\ O((2-\delta)^n)$ $\Omega((2+\epsilon)^n),\ \Omega(n!)$ $\Theta(2^{2n})$

Table 1: Some examples of order relations. In the above, $\epsilon > 0$, $0 \le \delta \le 1$, $p > 1$, and $k, k' > 1$.

occasionally do want to say something like "$h(n)$ *becomes negligible* compared to $g(n)$." You sometimes see the notation $h(n) \ll g(n)$, meaning "$h(n)$ is much smaller than $g(n)$," but this could apply to a situation where $h(n) = 0.001g(n)$. Not being interested in mere constant factors like this, we need something stronger. A traditional notation is "little-oh," defined as follows.

$$h(n) \in o(g(n)) \iff \lim_{n\to\infty} h(n)/g(n) = 0.$$

It's easy to see that if $h(n) \in o(g(n))$, then $h(n) \notin \Omega(g(n))$; no constant K can work in the definition of $\Omega(\cdot)$. It is not the case, however, that all functions that are not in $\Omega(g(n))$ must be in $o(g(n))$, as illustrated in Figure ??b.

Table 1 gives a few common examples of orders that we deal with and their containment relations.

3 Examples

3.1 Linear search

Let's apply all of this to a particular program. Here's a tail-recursive linear search for seeing if a particular value is in a sorted array:

```
/** True iff X is one of A[k]...A[A.length-1]. Assumes A is in
 *  increasing order, k>= 0. */
static boolean isIn(int[] A, int k, int X)
{
    if (k >= A.length)
        return false;
    else if (A[k] > X)
        return false;
    else if (A[k] == X)
        return true;
    else
        return isIn(A, k+1, X);
}
```

This is essentially a loop. As a measure of its complexity, let's define $C_{\texttt{isIn}}(N)$ as the maximum number of instructions it executes for a call with $k = 0$ and `A.length`$= N$. By inspection, you can see that such a call will execute the first `if` test up to $N + 1$ times, the second and third up to N times, and the tail-recursive call on `isIn` up to N times. With one compiler[1], each recursive call of `isIn` executes at most 14 instructions before returning or tail-recursively calling `isIn`. The initial call executes 18. That gives a total of at most $14N + 18$ instructions. If instead we count the number of comparisons `k>=A.length`, we get at most $N + 1$. If we count the number of comparisons against `X` or the number of fetches of `A[0]`, we get at most $2N$. We could therefore say that the function giving the largest amount of time required to process an input of size N is either in $O(14N+18)$, $O(N+1)$, or $O(2N)$. However, these are all the same set, and in fact all are equal to $O(N)$. Therefore, we may throw away all those messy integers and describe $C_{\texttt{isIn}}(N)$ as being in $O(N)$, thus illustrating the simplifying power of ignoring constant factors.

Again, this bound is a worst-case time. For all arguments in which `X<=A[0]`, the `isIn` function runs in constant time. That time bound—the *best-case* bound—is seldom very useful, especially when it applies to so atypical an input.

Giving an $O(\cdot)$ bound to $C_{\texttt{isIn}}(N)$ doesn't tell us that `isIn` *must* take time proportional to N even in the worst case, only that it takes no more. In this particular case, however, the argument used above shows that the worst case is, in fact, proportional to N, so that we may also say that $C_{\texttt{isIn}}(N) \in \Omega(N)$. Putting the two results together, $C_{\texttt{isIn}}(N) \in \Theta(N)$.

In general, then, asymptotic analysis of the space or time required for a given algorithm involves the following.

- Deciding on an appropriate measure for the *size* of an input (e.g., length of an array or a list).
- Choosing a representative quantity to measure—one that is proportional to the "real" space or time required.

[1] a version of gcc with the -O option, generating SPARC code for a Sun Sparcstation IPC workstation.

- Coming up with one or more functions that bound the quantity we've decided to measure, usually in the worst case.
- Possibly summarizing these functions by giving $O(\cdot)$, $\Omega(\cdot)$, or $\Theta(\cdot)$ characterizations of them.

3.2 Quadratic example

Here is a bit of code for sorting integers:

```
static void sort(int[] A) {
   for (int i = 1; i < A.length; i += 1) {
      int x = A[i];
      int j;
      for (j = i; j > 0 && x < A[j-1]; j -= 1)
         A[j] = A[j-1];
      A[j] = x;
   }
}
```

If we define $C_{\texttt{sort}}(N)$ as the worst-case number of times the comparison `x < A[j-1]` is executed for $N =$ `A.length`, we see that for each value of `i` from 1 to `A.length-1`, the program executes the comparison in the inner loop (on `j`) at most `i` times. Therefore,

$$\begin{aligned} C_{\texttt{sort}}(N) &= 1 + 2 + \ldots + N - 1 \\ &= N(N-1)/2 \\ &\in \Theta(N^2) \end{aligned}$$

This is a common pattern for nested loops.

3.3 Explosive example

Consider a function with the following form.

```
static int boom(int M, int X)
{
    if (M == 0)
        return H(X);
    return boom(M-1, Q(X)) + boom(M-1, R(X));
}
```

and suppose we want to compute $C_{\texttt{boom}}(M)$—the number of times `Q` is called for a given M in the worst case. If $M = 0$, this is 0. If $M > 0$, then `Q` gets executed once in computing the argument of the first recursive call, and then it gets executed however many times the two inner calls of `boom` with arguments of $M - 1$ execute it. In other words,

$$\begin{aligned} C_{\texttt{boom}}(0) &= 0 \\ C_{\texttt{boom}}(i) &= 2C_{\texttt{boom}}(i-1) + 1 \end{aligned}$$

A little mathematical massage:

$$C_{\texttt{boom}}(M) = 2C_{\texttt{boom}}(M-1) + 1, \text{ for} M \geq 1$$

$$
\begin{aligned}
&= 2(2C_{\texttt{boom}}(M-2)+1)+1, \text{ for} M \geq 2 \\
&\vdots \\
&= \underbrace{2(\cdots(2}_{M}0+\underbrace{1)+1)\cdots+1}_{M} \\
&= \sum_{0 \leq j \leq M-1} 2^j \\
&= 2^M - 1
\end{aligned}
$$

and so $C_{\texttt{boom}}(M) \in \Theta(2^M)$.

3.4 Divide and conquer

Things become more interesting when the recursive calls decrease the size of parameters by a multiplicative rather than an additive factor. Consider, for example, binary search.

```
/** Returns true iff X is one of A[L]...A[U].  Assumes A is in
  * increasing order, L>=0, U-L < A.length. */
static boolean isInB(int[] A, int L, int U, int X);
{
    if (L > U)
        return false;
    else {
        int m = (L+U)/2;
        if (A[m] == X)
            return true;
        else if (A[m] > X)
            return isInB(A, L, m-1, X);
        else
            return isInB(A, m+1, U, X);
    }
}
```

The worst-case time here depends on the number of elements of `A` under consideration, $U - L + 1$, which we'll call N. Let's use the number of times the first line is executed as the cost, since if the rest of the body is executed, the first line also had to have been executed[2]. If $N > 1$, the cost of executing `isInB` is 1 comparison of `L` and `U` followed by the cost of executing `isInB` either with $\lfloor (N-1)/2 \rfloor$ or with $\lceil (N-1)/2 \rceil$ as the new value of N[3]. Either quantity is no more than $\lceil (N-1)/2 \rceil$.

If $N \leq 1$, then in the worst case, there are two comparisons against N.

Therefore, the following recurrence describes the cost, $C_{\texttt{isInB}}(i)$, of executing this function when $U - L + 1 = i$.

$$
\begin{aligned}
C_{\texttt{isInB}}(1) &= 2 \\
C_{\texttt{isInB}}(i) &= 1 + C_{\texttt{isInB}}(\lceil (i-1)/2 \rceil), \; i > 1.
\end{aligned}
$$

[2] For those of you seeking arcane knowledge, we say that the test `L>U` *dominates* all other statements.

[3] The notation $\lfloor x \rfloor$ means the result of rounding x down (toward $-\infty$) to an integer, and $\lceil x \rceil$ means the result of rounding x up to an integer.

This is a bit hard to deal with, so let's again make the reasonable assumption that the value of the cost function, whatever it is, must increase as N increases. Then we can compute a cost function, $C'_{\texttt{isInB}}$ that is slightly larger than $C_{\texttt{isInB}}$, but easier to compute.

$$\begin{aligned} C'_{\texttt{isInB}}(1) &= 2 \\ C'_{\texttt{isInB}}(i) &= 1 + C'_{\texttt{isInB}}(i/2),\ i > 1 \text{ a power of 2.} \end{aligned}$$

Again, this is a slight over-estimate of $C_{\texttt{isInB}}$, but that still allows us to compute upper bounds. Furthermore, $C'_{\texttt{isInB}}$ is defined only on powers of two, but since `isInB`'s cost increases as N increases, we can still bound $C_{\texttt{isInB}}(N)$ conservatively by computing $C'_{\texttt{isInB}}$ of the next higher power of 2. Again with the massage:

$$\begin{aligned} C'_{\texttt{isInB}}(i) &= 1 + C'_{\texttt{isInB}}(i/2),\ i > 1 \text{ a power of 2.} \\ &= 1 + 1 + C'_{\texttt{isInB}}(i/4),\ i > 2 \text{ a power of 2.} \\ &\vdots \\ &\underbrace{1 + \cdots + 1}_{\lg N} + 2 \end{aligned}$$

The quantity $\lg N$ is the logarithm of N base 2, or informally "the number of times one can divide N by 2 before reaching 1." In summary, we can say $C_{\texttt{isIn}}(N) \in O(\lg N)$. Similarly, one can in fact derive that $C_{\texttt{isIn}}(N) \in \Theta(\lg N)$.

4 Divide and fight to a standstill

Consider now a subprogram that contains *two* recursive calls.

```
static int mung(int[] A, L, U);
{
    if (L >= U)
        return false;
    else {
        int m = (L+U)/2;
        mung(A, L, m);
        mung(A, m+1, U);
    }
}
```

We can approximate the arguments of both of the internal calls by $N/2$ as before, ending up with the following approximation, $C_{\texttt{mung}}(N)$ to the cost of calling `mung` with argument $N = U - L + 1$ (we are counting the number of times the test in the first line executes).

$$\begin{aligned} C_{\texttt{mung}}(1) &= 3 \\ C_{\texttt{mung}}(i) &= 1 + 2C_{\texttt{mung}}(i/2),\ i > 1 \text{ a power of 2.} \end{aligned}$$

So,

$$\begin{aligned} C_{\texttt{mung}}(N) &= 1 + 2(1 + 2C_{\texttt{mung}}(N/4)),\ N > 2 \text{ a power of 2.} \\ &\vdots \\ &= 1 + 2 + 4 + \ldots + N/2 + N \cdot 3 \end{aligned}$$

This is a sum of a geometric series ($1+r+r^2+\cdots+r^m$), with a little extra added on. The general rule for geometric series is

$$\sum_{0 \le k \le m} r^k = (r^{m+1}-1)/(r-1) = (r \cdot r^m - 1)/(r-1)$$

so, taking $r = 2$,

$$C_{\texttt{mung}}(N) = 4N - 1$$

or $C_{\texttt{mung}}(N) \in \Theta(N)$.

5 Complexity of Problems

So far, I have discussed only the analysis of an algorithm's complexity. An algorithm, however, is just a particular way of solving some problem. We might therefore consider asking for complexity bounds on the *problem's* complexity. That is, can we bound the complexity of the *best possible* algorithm? Obviously, if we have a particular algorithm and its time complexity is $O(f(n))$, where n is the size of the input, then the complexity of the best possible algorithm must also be $O(f(n))$. We call $f(n)$, therefore, an *upper bound* on the (unknown) complexity of the best-possible algorithm. But this tells us nothing about whether the best-possible algorithm is any *faster* than this—it puts no *lower bound* on the time required for the best algorithm. For example, the worst-case time for `isIn` is $\Theta(N)$. However, `isInB` is much faster. Indeed, one can show that if the only knowledge the algorithm can have is the result of comparisons between `X` and elements of the array, then `isInB` has the best possible bound (it is *optimal*), so that the entire *problem* of finding an element in an ordered array has worst-case time $\Theta(\lg N)$.

Putting an upper bound on the time required to perform some problem simply involves finding an algorithm for the problem. By contrast, putting a good lower bound on the required time is much harder. We essentially have to prove that no algorithm can have a better execution time than our bound, regardless of how much smarter the algorithm designer is than we are. Trivial lower bounds, of course, are easy: every problem's worst-case time is $\Omega(1)$, and the worst-case time of any problem whose answer depends on all the data is $\Omega(N)$, assuming that one's idealized machine is at all realistic. Better lower bounds than those, however, require quite a bit of work. All the better to keep our theoretical computer scientists employed.