CS61B Summer 2006
Instructor: Erin Korber
Lecture 12, 17 July

Asymptotic Analysis

Suppose an algorithm for processing a retail store's inventory takes: 10,000 milliseconds to read the initial inventory from disk, and then 10 milliseconds to process each transaction (items acquired or sold). Processing n transactions takes $(10,000 + 10n)$ ms. Even though 10,000 is much greater than 10, we can see that the $10n$ term will be more important if the number of transactions is very large.

We also know that these coefficients will change if we buy a faster computer or disk drive, or use a different language or compiler. We want a way to express the speed of an algorithm independently of a specific implementation on a specific machine - specifically, we want to ignore constant factors (which get smaller and smaller as technology improves).

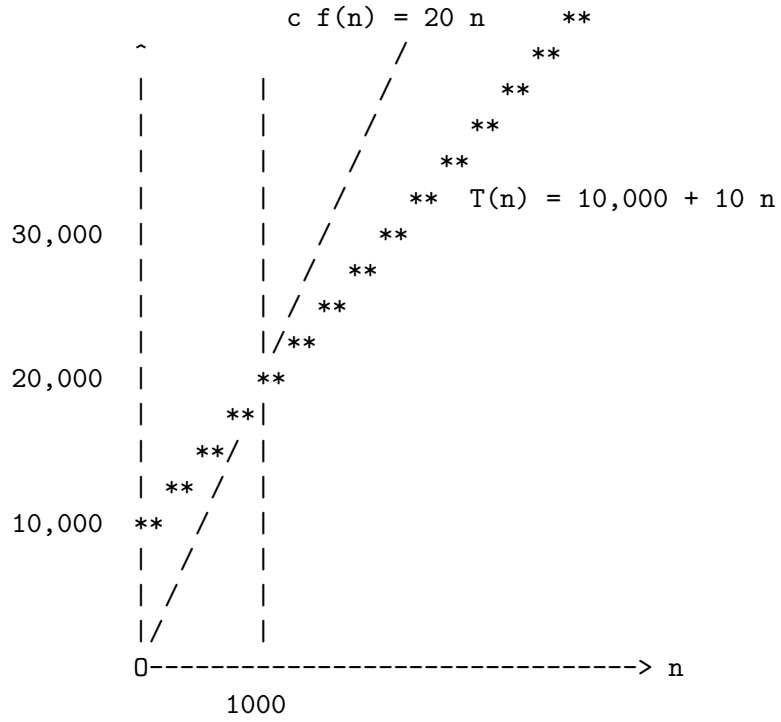
1 Big-Oh Notation (upper bounds on running time or memory)

Big-Oh notation tells us how slowly a program might run as its input grows.

- Let n be the size of a program's *input* (in bits or data words or whatever).
- Let $T(n)$ be a function. Here, $T(n)$ is precisely equal to the algorithm's running time, given an input of size n.
- Let $f(n)$ be another function – preferably a simple function like $f(n) = n$ or $f(n) = n^2$
- We say that $T(n)$ is in $O(f(n))$ IF AND ONLY IF $T(n) \leq cf(n)$ WHENEVER n IS BIG, FOR SOME CONSTANT c.
  - HOW BIG IS "BIG"? Big enough to make $T(n)$ fit under $c*f(n)$.
  - HOW LARGE IS c? Large enough to make $T(n)$ fit under $c*f(n)$.
- We'll be making this notion more formal in a bit, but first let's look at an example:

1.1 EXAMPLE: Inventory

Let's consider the function $T(n) = 10,000 + 10n$, from our example above. Let's try out $f(n) = n$, because it's simple. We can choose c as large as we want, and we're trying to make $T(n)$ fit underneath $c * f(n)$, so let's pick $c = 20$.



As these functions extend forever to the right, their asymptotes will never cross again. For large n – any n bigger than 1000, in fact - $T(n) \leq c * f(n)$. *THEREFORE,* $T(n)$ *is in* $O(f(n))$.

Now, we're going to express this idea rigorously. Here is the central lesson of today's lecture, which will bear on your entire career as a professional computer scientist, however abstruse it may seem now:

- $O(f(n))$ is the SET of ALL functions $T(n)$ that satisfy: There exist positive constants c and k such that, for all $n \geq k$, $T(n) \leq c * f(n)$

For instance, in the graph above, $c = 20$, and $k = 1000$.

Think of it this way: if you're trying to prove that one function is asymptotically bounded by another [$f(n)$ is in $O(g(n))$], you're allowed to multiply them by positive constants in an attempt to stuff one underneath the other. You're also allowed to move the vertical line (k) as far to the right as you like (to get all the crossings onto the left side). We're only interested in how the functions behave as they shoot off toward infinity.

1.2 Examples: Some Important Corollaries

1. $1,000,000n$ is in $O(n)$.

   Proof: set $c = 1,000,000$, $k = 0$.

   The point: Big-Oh notation doesn't care about constant factors. We generally leave constants out; it's unnecessary to write $O(2n)$.

2. n is in $O(n^3)$.

   Proof: set $c = 1$, $k = 1$.

   The point: Big-Oh notation only gives us an UPPER BOUND on the function. Just because an algorithm's running time is in $O(n^3)$ doesn't mean it's slow; it might also be in $O(n)$. Big-Oh notation only gives us an UPPER BOUND on the function.

3. $n^3 + n^2 + n$ is in $O(n^3)$.

   Proof: set $c = 3$, $k = 1$.

   The point: Big-Oh notation is usually used only to indicate the dominating (largest) term in the function. The other terms become insignificant when n is really big - we can always find a k large enough that they don't matter.

Here's a table to help you figure out the dominating term.

1.3 Table of Important Big-Oh Sets

Arranged from smallest to largest, in order of increasing domination:

```
                O(     1     )       ::  constant
is a subset of O(   log n    )       ::  logarithmic
is a subset of O(  sqrt(n)   )       ::  root-n
is a subset of O(     n      )       ::  linear
is a subset of O(  n log n   )       ::  n log n
is a subset of O(    n^2     )       ::  quadratic
is a subset of O(    n^3     )       ::  cubic
is a subset of O(    n^4     )       ::  quartic
is a subset of O(    2^n     )       ::  exponential
is a subset of O(    e^n     )       ::  exponential
```

Algorithms that are O(n log n) or faster are gemerally considered efficient. Algorithms that take n^7 time or more are usually considered so slow as to be useless on anything but very small input sizes. (However, if all possible ways to solve a problem are slower than that, and you need to solve the problem, such algorithms may still be quite useful!) In the regions between n log n and n^7, the usefulness of an algorithm generally depends on the typical input sizes and the associated constants hidden by the Big-Oh notation.

1.4 Common pitfalls

1. Here's a fallacious proof: n^2 is in $O(n)$, because if we choose $c = n$, we get $n^2 \leq n^2$. ** WRONG! c and k must be *constants*; they cannot depend on n.

2. The big-Oh notation expresses a relationship between functions. IT DOES NOT SAY WHAT THE FUNCTIONS ARE. In particular, the function on the left does not need to be the worst-case running time, though it often is.

   For example, in binary search on an array,

   - the worst-case running time is in O(log n),
   - the best-case running time is in O(1),
   - the memory use is in O(n), and
   - (47 + 18 log n - 3/n) is in O(the worst-case running time).

It's easy to fall into the trap of thinking that "big-O" always means "worst-case running time."

3. Big-Oh notation doesn't tell the whole story, because it leaves out the constants. If one algorithm runs in time $T(n) = nlogn$, and another algorithm runs in time $U(n) = 100n$, then Big-Oh notation suggests you should use $U(n)$, because $T(n)$ dominates $U(n)$ asymptotically - there is a k such that for input size $n \geq k$, $U(n) < T(n)$. However, this k is so large that $U(n)$ is only faster than $T(n)$ in practice if your input size is greater than current estimates of the number of subatomic particles in the universe. It might be helpful to know that $log_2 n < 50$ for any input size you are ever likely to encounter.

   Nevertheless, Big-Oh notation is still a good rule of thumb, because the hidden constants in real-world algorithms usually aren't that big.

2 Omega Notation (lower bounds on running time or memory)

- $\Omega(f(n))$ is the SET of ALL functions $T(n)$ that satisfy: There exist positive constants c and k such that, for all $n \geq k$, $T(n) \geq c * f(n)$

(Compare with the definition of Big-Oh: $T(n) \leq c * f(n)$.)

Omega is the reverse of Big-Oh. If $T(n)$ is in $O(f(n))$, then $f(n)$ is in $\Omega(T(n))$.

n is in $\Omega(1)$	BECAUSE	1 is in $O(n)$
n^2 is in $\Omega(n)$	BECAUSE	n is in $O(n^2)$
n^2 is in $\Omega(3n^2 + nlogn)$	BECAUSE	$3n^2 + nlognisinO(n^2)$

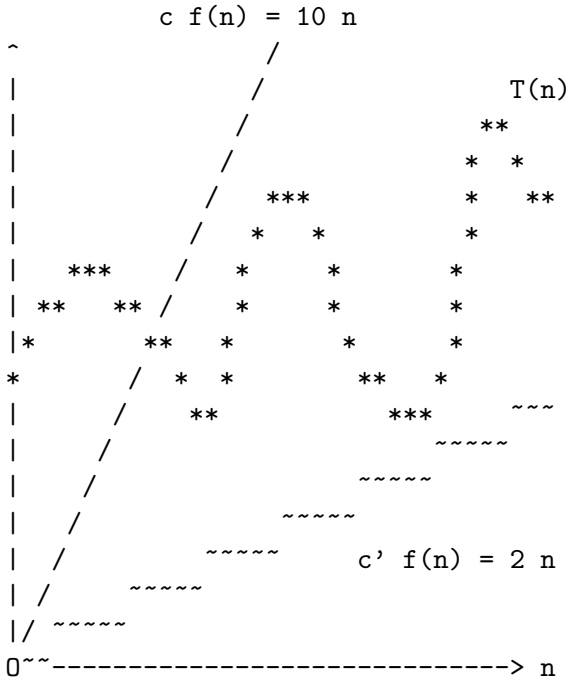
Omega gives us a LOWER BOUND on a function, just as Big-Oh gives us an UPPER BOUND. Big-Oh says, "Your algorithm is at least this good." Omega says, "Your algorithm is at least this bad."

If we have both–say, $T(n)$ is in $O(f(n))$ and is also in $\Omega(g(n))$–then $T(n)$ is effectively sandwiched between the two, below $f(n)$ and above $g(n)$.

If $f = g$, we say that $T(n)$ is in $\Theta(f(n))$:

- $\Theta(f(n))$ is the set of all functions $T(n)$ that are in both $O(f(n))$ and $\Omega(f(n))$

But how can a function be sandwiched between $f(n)$ and $f(n)$? Easy: we choose different constants c for the upper bound and lower bound. For instance, here is a function $T(n)$ in $\Theta(n)$:



If we extend this graph infinitely far to the right, and find that $T(n)$ remains always sandwiched between the two linear functions, then $T(n)$ is in $\Theta(n)$. If $T(n)$ is an algorithm's worst-case running time, the algorithm will never exhibit worse than linear performance, but it can't be counted on to ever exhibit better than linear performance, either.

Theta is symmetric: if $f(n)$ is in $\Theta(g(n))$, then $g(n)$ is in $\Theta(f(n))$. For instance, n^3 is in $\Theta(3n^3 - n^2)$, and $3n^3 - n^2$ is in $\Theta(n^3)$. n^3 is not in $\Theta(n)$, and n is not in $\Theta(n^3)$.

Recall that you have to interpret Big-Oh notation carefully – because n is in $O(n^8)$, for instance, so saying that an algorithm runs in $O(n^8)$ time doesn't mean it's slow. Theta notation is more specific: n is NOT in $\Theta(n^8)$. If your algorithm runs in $\Theta(n^8)$ time, it IS slow.

Finally, remember that the choice of O, Omega, Theta is *independent* of whether we're talking about worst-case running time, best-case time, average- case time, memory use, annual beer consumption as a function of population, or some other function. The function has to be specified. "Big-Oh" is NOT a synonym for "worst-case running time".

3 Designing programs

- Writing a faster algorithm is a much better way to speed up your program than trying to reduce the size of the constants.
- Programmer time is more valuable than computer time. This only becomes more true as time goes on and computers get faster and cheaper.
- If your typical input sizes are going to be small, almost any algorithm will be fast enough (especially if it's not exponential).

4 Analyzing Algorithms

Problem 1: Given an array of n integers, find the smallest.

Algorithm 1: Maintain a variable `min` that stores the smallest integer scanned so far.

Set `min` to the first element of the array.

Scan through the rest of the array, updating `min` as necessary.

The running time is in $\Omega(n)$, because we must scan the whole array once.

The running time is in $O(n)$, because all we do is scan the whole array once.

Therefore, the worst- and best-case running times are both in $\Theta(n)$.

Problem 2: Given a set of p points, find the pair closest to each other.

Algorithm 2: Calculate the distance between each pair; return the minimum.

There are $p(p-1)/2$ pairs, and each pair takes constant time to examine. Therefore, worst- and best-case running times are in $\Theta(p^2)$.

The code for this algorithm gives a strong hint, because it contains a doubly-nested loop, with both loops iterating through input points.

```
double minDistance = point[0].distance(point[1]);

for (int i = 0; i < numPoints; i++) {
/* We require that j > i so that each pair is visited only once. */
    for (int j = i + 1; j < numPoints; j++) {
       double thisDistance = point[i].distance(point[j]);
       if (thisDistance < minDistance) {
          minDistance = thisDistance;
        }
    }
```

4.1 Functions of Several Variables

Problem 3: Write a mixed doubles tennis-team forming program for w women and m men.

Algorithm 3: Compare each woman with each man. Decide if they can be on a team together.

If each comparison takes constant time then the running time, $T(w, m)$, is in $\Theta(wm)$.

This means that there exist constants c, d, W, and M, such that $dwm \leq T(w, m) \leq cwm$ for every $w \geq W$ and $m \geq M$.

T is not in $O(w^2)$, nor in $O(m^2)$, nor in $\Omega(w^2)$, nor in $\Omega(m^2)$. Every one of these possibilities is eliminated either by choosing w much greater than m or m much greater than w. Conversely, w^2 is in neither $O(wm)$ nor $\Omega(wm)$. You cannot asymptotically compare the functions wm, w^2, and m^2.

If we expand our service to help form women's volleyball teams as well, the running time is in $\Theta(w^6 + wm)$.

This expression cannot be simplified; neither term dominates the other.

Problem 4: Suppose you have an array containing n music albums, sorted by title. You request a list of all albums whose titles begin with "The Best of"; suppose there are k such albums.

Algorithm 4: Search for the first matching album with binary search. Walk (in both directions) to find the other matching albums.

Binary search takes at most log n steps to find a matching album (if one exists). Next, the complete list of k matching albums is found, each in constant time. Thus, the worst-case running time is in $\Theta(logn + k)$.

Because k can be as large as n, it is not dominated by the log n term.

Because k can be as small as zero, it does not dominate the log n term.

Hence, there is no simpler expression for the worst-case running time.

Algorithms like this are called "output-sensitive," because their performance depends partly on the size k of the output, which can vary greatly.

Because binary search sometimes gets lucky and finds a match right away, the BEST-case running time is in $\Theta(1 + k) = \Theta(k)$.

Problem 5: Find the k-th item in an n-node doubly-linked list.

Algorithm 5: If $k < 1$ or $k > n$, report an error and return. Otherwise, compare k with $n - k$. If k $\leq n - k$, start at the beginning of the list and walk forward $k - 1$ nodes. Otherwise, start at the end of the list and walk backward $n - k$ nodes.

If $1 \leq k \leq n$, this algorithm takes $\Theta(mink, n-k)$ time (in all cases) This expression cannot be simplified: without knowing k and n, we cannot say that k dominates n-k or that n-k dominates k.