CS 61B: Lecture 23
Monday, March 17, 2014

ASYMPTOTIC ANALYSIS (continued): More Formalism

```
|-------------------------------------------------------------------------|
| Omega(f(n)) is the set of all functions T(n) that satisfy:              |
|                                                                         |
|   There exist positive constants d and N such that, for all n >= N,     |
|                           T(n) >= d f(n)                                |
|-------------------------------------------------------------------------|
^^^^^^^^^^  Compare with the definition of Big-Oh:  T(n) <= c f(n). ^^^^^^^^^^^
```

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

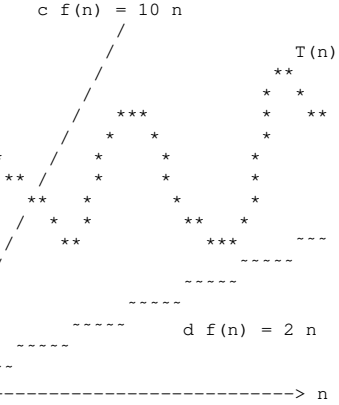
```
  2n   is in Omega(n)                BECAUSE    n               is in O(2n).
  n^2  is in Omega(n)                BECAUSE    n               is in O(n^2).
  n^2  is in Omega(3 n^2 + n log n)  BECAUSE    3 n^2 + n log n  is in O(n^2).
```

Big-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." Big-Omega says, "Your algorithm is at least this bad."

Recall that Big-Oh notation can be misleading because, for instance, n is in O(n^8). If we know both a lower bound and an upper bound for a function, and they're both the same bound asymptotically (i.e. they differ only by a constant factor), we can use Big-Theta notation to precisely specify the function's asymptotic behavior.

```
|-------------------------------------------------------------------------|
| Theta(f(n)) is the set of all functions that are in both of             |
|                                                                         |
|                        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 and d) 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 2n and 10n, 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 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(3 n^3 - n^2), and 3 n^3 - n^2 is in Theta(n^3). n^3 is not in Theta(n), and n is not in Theta(n^3).

Big-Theta notation isn't potentially misleading in the way Big-Oh notation can be: n is NOT in Omega(n^8). If your algorithm's running time is in Theta(n^8), it IS slow.

However, some functions are not in "Theta" of anything simple. For example, the function f(n) = n (1 + sin n) is in O(n) and Omega(0), but it's not in Theta(n) nor Theta(0). f(n) keeps oscillating back and forth between zero and ever-larger numbers. We could say that f(n) is in Theta(2n (1 + sin n)), but that's not a simplification.

Remember that the choice of O, Omega, or Theta is _independent_ of whether we're talking about worst-case running time, best-case running time, average-case running 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," and Omega is not a synonym for "best-case running time."

ALGORITHM ANALYSIS
==========
Problem #1: Given a set of p points, find the pair closest to each other.
Algorithm #1: 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).

Often, you can figure out the running time of an algorithm just by looking at the loops--their loop bounds and how they are nested. For example, in the closest pair code below, the outer loop iterates p times, and the inner loop iterates an average of roughly p / 2 times, which multiply to Theta(p^2) time.

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

/* Visit a pair (i, j) of points. */
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;
    }
  }
}
```

But doubly-nested loops don't always mean quadratic running time! The next example has the same loop structure, but runs in linear time.

Problem #2: Smooshing an array called "ints" to remove consecutive duplicates, from Homework 3.
Algorithm #2:

```
int i = 0, j = 0;

while (i < ints.length) {
  ints[j] = ints[i];
  do {
    i++;
  } while ((i < ints.length) && (ints[i] == ints[j]));
  j++;
}
// Code to fill in -1's at end of array omitted.
```

The outer loop can iterate up to ints.length times, and so can the inner loop. But the index "i" advances on _every_ iteration of the inner loop. It can't advance more than ints.length times before both loops end. So the worst-case running time of this algorithm is in Theta(ints.length). (So is the best-case time.)

Unfortunately, I can't give you a foolproof formula for determining the running time of any algorithm. You have to think! In fact, the problem of determining an algorithm's running time is, in general, as hard as proving _any_ mathematical theorem. For instance, I could give you an algorithm whose running time depends on whether the Riemann Hypothesis (one of the greatest unsolved questions in mathematics) is true or false.

Functions of Several Variables

Problem #3: Write a matchmaking program for w women and m men.
Algorithm #3: Compare each woman with each man. Decide if they're compatible.

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
d wm <= T(w, m) <= c wm for every w >= W and m >= 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 >> m or m >> 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. You cannot asymptotically compare the functions w^6 and wm.

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 one 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(log n + 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(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 <= n - k, start at the front of the list and walk forward k - 1 nodes.

Otherwise, start at the back of the list and walk backward n - k nodes.

If 1 <= k <= n, this algorithm takes Theta(min{k, 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.