Recitation 8: Big-O and Sorting

Content by: Josh Zimmerman and Elizabeth Davis
Compiled by: Jonathan Clark

September 21, 2012

Overview

Today, we'll review recent concepts from lecture: Big-O and sorting. Here's the agenda:

- Hand back homework 1
- Big O is a set of functions. What's in it?
- How do we find average case versus worst case using Big O?
- A bit of practice putting Big O bounds in order
- Analyze binary search using Big O
- An overview of sorting
- Analyze selection sort using Big O
- Another look at quick sort (as time permits)

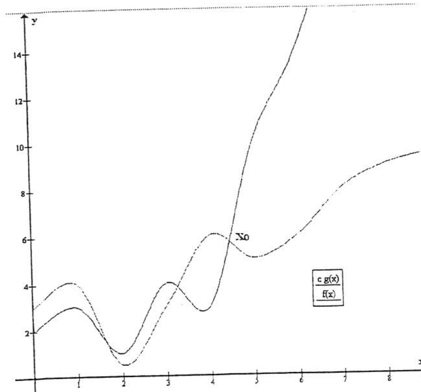
Big-O

I used the $\in$ symbol above for big-O, which might be confusing if you've never dealt with big-O formally before. So, let's familiarize ourselves with the formal definitions.

First of all, $O(g(n))$ is a *set* of functions. The formal definition is:

$f(n) \in O(g(n))$ if and only if there is some $c \in \mathbb{R}^+$ and some $n_0 \in \mathbb{R}$ such that for all $n > n_0$, $f(n) \le c * g(n)$. (Note that there should be some absolute values in there, but as we don't generally deal with negative runtime complexities it's fine to neglect them for this class.)

That's a bit formal and possibly confusing, so let's look at the intuition. Here's a diagram, courtesy of Wikipedia. (Note that it uses x and x_0 instead of n and n_0.)



Below n_0, the functions can do anything. Above it, we know that $c * g(n)$ is always greater than $f(n)$.

In problems 1 and 2, we play with the definition of big-O.

Something that's very important to note about this diagram is that there are infinitely many functions that are in $O(g(n))$: If $f(n) \in O(g(n))$, then $\frac{1}{2}f(n) \in O(g(n))$ and $\frac{1}{4}f(n) \in O(g(n))$ and $2f(n) \in O(g(n))$. In general, for any constant k, $k * f(n) \in O(g(n))$. In problem 3, we prove this.

Something that will come up often with big-O is the idea of a tight lower bound.

It's technically correct to say that binary search, which takes around $\log(n)$ steps on an array of length n, is $O(n!)$, since $n! > \log(n)$ for most n, but it's not very useful. If we ask for a *tight* bound, we want the closest bound you can give. For binary search, $O(\log(n))$ is a tight bound because no function that grows more slowly than $\log(n)$ provides a correct upper bound for binary search.

Practice!

1. Rank these big-O sets from left to right such that every big-O is a subset of everything to the right of it. (For instance, $O(n)$ goes farther to the left than $O(n!)$ because $O(n) \subset O(n!)$.) If two sets are the same, put them on top of each other.

$O(n!)$ $O(n)$ $O(4)$ $O(n\log(n))$ $O(4n+3)$ $O(n^2+2n+3)$ $O(1)$ $O(n^2)$ $O(2^n)$
$O(n^3+300n^2)$ $O(\log(n))$ $O(\log^2(n))$ $O(\log(\log(n)))$ $O(n^3)$

2. Using the formal definition of big-O, prove that $n^3 + 300n^2 \in O(n^3)$.

3. Using the formal definition of big-O, prove that if $f(n) \in O(g(n))$, then $k * f(n) \in O(g(n))$ for $k \ge 0$.

One interesting consequence of this is that $O(\log_i(n)) = O(\log_j(n))$ for all i and j (as long as they're both greater than 0), because of the change of base formula:

$$\log_i(n) = \frac{\log(n)}{\log(i)}$$

But $\frac{1}{\log(i)}$ is just a constant! So, it doesn't matter what base we use for logarithms in big-O notation.

Determining the runtime of an algorithm

Here are a couple of functions. Can we figure out a tight big-O bound just by looking at the code?

```
int binsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A,n);
/*@ensures (-1 == \result && !is_in(x, A, n))
      || ((0 <= \result && \result < n) && A[\result] == x); @*/
{
   int lower = 0;
   int upper = n;
   while (lower < upper)
     //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
     //@loop_invariant (lower == 0 || A[lower-1] < x);
     //@loop_invariant (upper == n || A[upper] > x);
     {
        int mid = lower + (upper-lower)/2;
        //@assert lower <= mid && mid < upper;
        if (A[mid] == x) return mid;
        else if (A[mid] < x) lower = mid+1;
        else /*@assert(A[mid] > x);@*/ upper = mid;
     }
     return -1;
}
```

```
void time_waster(int n) {
  for (int i = 0; i < n; i++) {
    int[] arr = alloc_array(int, i);
    for (int j = 0; j < i; j++)
      arr[j] = j;
  }
  return;
}
```

Sorting

Sorting is a really useful task that's central to many programs (for instance, spreadsheet programs) as well as many algorithms (for instance, Google PageRank needs to sort pages before presenting them).

One such algorithm is selection sort, which we discussed in lecture. The basic idea behind selection sort is that we find the smallest number in the region of the array that we're considering, switch that with the first element of the region of the array that we're switching, shrink the size of the array we're considering by one, and repeat until we're considering no elements of our array.

Here's a good visualization of selection sort on a series of poles: `http://www.youtube.com/watch?v=LuANFAXgQEw`. In that video, the light blue square points to the smallest element we've found so far, the purple square points to the element we're comparing with the minimum, and the pink rectangle is the region of the array we still need to consider.

This is not a particularly efficient algorithm: if we have i elements in the region of the array we're considering, we need to look at all i of them, every time.

So, the overall amount of work we'll have to do is $n + (n-1) + (n-2) + \cdots + 2 + 1$, or:

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

Now that you know big-O notation and have started to learn about time complexity, it should be possible for you to figure out what the time complexity of this algorithm is in big-O notation.

```
1  int min_index(int[] A, int lower, int upper)
2  //@requires 0 <= lower && lower < upper && upper <= \length(A);
3  //@ensures lower <= \result && \result < upper;
4  //@ensures le_seg(A[\result], A, lower, upper);
5  {
6      int m = lower;
7      int min = A[lower];
8      for (int i = lower+1; i < upper; i++)
9      //@loop_invariant lower < i && i <= upper;
10     //@loop_invariant le_seg(min, A, lower, i);
11     //@loop_invariant lower <= m && m < upper;
12     //@loop_invariant A[m] == min;
13     {
14         if (A[i] < min) {
15             m = i;
16             min = A[i];
17         }
18     }
19     return m;
20 }
21
22 void sort(int[] A, int n)
23 //@requires 0 <= n && n <= \length(A);
24 //@ensures is_sorted(A, 0, n);
25 {
26     for (int i = 0; i < n; i++)
27     //@loop_invariant 0 <= i && i <= n;
28     //@loop_invariant is_sorted(A, 0, i);
29     //@loop_invariant le_segs(A, 0, i, i, n);
30     {
31         int m = min_index(A, i, n);
32         //@assert le_seg(A[m], A, i, n);
33         swap(A, i, m);
34     }
35     return;
36 }
```

Quicksort

Quicksort is a fast ($O(n \log n)$ on average, $O(n^2)$ worst case) sorting algorithm. The general idea behind quicksort is that we start with an array we need to sort (it may be sorted or unsorted), and then select a *pivot* index. (There are many different methods of picking a pivot index: some strategies always pick index 0 or the largest possible index for the array (i.e. `n - 1` where `n` is the length of the subarray we're considering), some choose a randomly generated index, and some choose the median of three randomly generated indices. The different strategies affect the performance of quicksort in different ways.)

After we pick `pivot_index`, we rearrange the array so that everything to the on the left of the array is less than or equal to `A[pivot_index]` and everything on the right of the array is greater than `A[pivot_index]`. Then, we call quicksort recursively on the portion of the array that is less than or equal to the pivot and on the portion that is greater than the pivot. If the list is empty or has one element, we just return the list since it's already sorted.

Quicksort can be a hard to understand at first, so let's look at an example.

Start with the array `[4, 2, 1, 0]`. We randomly select 1 as our pivot index. We switch `A[1]` and `A[n - 1]`, which gives us `[4, 0, 1, 2]`.

We split the array so everything on the left is less than or equal to the pivot. After we've done that, we have `[1, 0, 4, 2]`. Then, we recursively sort the two portions of the array: `[1, 0]` and `[4]`.

First, we sort `[1, 0]`. We must choose 0 as our pivot index, since otherwise the switching won't work correctly — we switch the element at the pivot index with the last element of the array, and if the pivot index is the index of the last element in the array we can't do that switch.

Our switch then results in the array `[0, 1]`. This already has all of the smallest elements on the left, so we can continue.

Then, we recursively sort the array `[0]`. It has only one element, so it's sorted already.

Here's the code for `partition`. We're going to prove that it's correct. (Note: since `qsort` is a recursive function, it requires a slightly different proof format. Instead of loop invariants, we'd have to base our proof entirely on preconditions and postconditions of `qsort` and `partition`. Recursive programs generally have relatively clean proofs by induction that they're correct. If you don't know what induction is yet, that's fine. [If you're currently in 15-151, you'll learn about it in lecture today.])

```
1  int partition(int[] A, int lower, int pivot_index, int upper)
2  //@requires 0 <= lower && lower <= pivot_index;
3  //@requires pivot_index < upper && upper <= \length(A);
4  //@ensures lower <= \result && \result < upper;
5  //@ensures ge_seg(A[\result], A, lower, \result);
6  //@ensures le_seg(A[\result], A, \result + 1, upper);
7  {
8      int pivot = A[pivot_index];
9      swap(A, pivot_index, upper - 1);
10
11     int left = lower;
12     int right = upper - 2;
13     while (left <= right)
14     //@loop_invariant lower <= left && left <= right + 1 && right + 1 < upper;
15     //@loop_invariant ge_seg(pivot, A, lower, left);
16     //@loop_invariant le_seg(pivot, A, right + 1, upper - 1);
17     {
18         if (A[left] <= pivot) {
19             left++;
20         }
21         else {
22             //@assert A[left] > pivot;
23             swap(A, left, right);
24             right--;
25         }
26     }
27     //@assert left == right + 1;
28     //@assert A[upper - 1] == pivot;
29     swap(A, left, upper - 1);
30     return left;
```