15-122: Principles of Imperative Computation

Recitation 5 — Josh Zimmerman

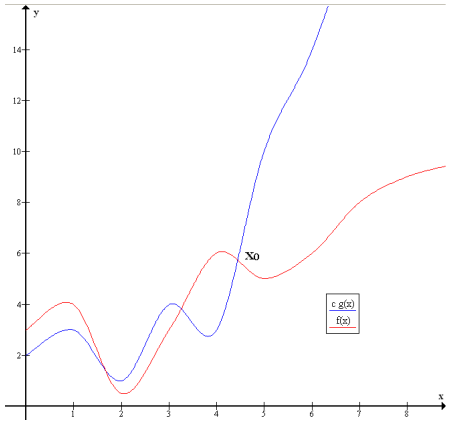
Big-O

The definition of Big-O has a lot of mathematical symbols in it, and so can be very confusing at first. Let's familiarize ourselves with the formal definition and get an intuition behind what it's saying.

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 $n > n_0$, $f(n) \leq c * g(n)$.

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.)



To the left of n_0, the functions can do anything. To its right, $c * g(n)$ is always greater than or equal to $f(n)$.

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* bound on the runtime of a function.

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 all $n > 0$ 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.

Unless we specify otherwise, we want a tight bound.

Selection sort

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 and switch that into

the section of the array that's sorted. Next we shrink the size of the region of the array we're considering by one and repeat until the region we're considering is empty.

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 a tight (big-O) upper bound for the complexity of selection sort is.

```
1 #use "sortutil.c0"
2
3 int min_index(int[] A, int lower, int upper)
4 //@requires 0 <= lower && lower < upper && upper <= \length(A);
5 //@ensures lower <= \result && \result < upper;
6 //@ensures le_seg(A[\result], A, lower, upper);
7 {
8    int m = lower;
9    int min = A[lower];
10   for (int i = lower+1; i < upper; i++)
11   //@loop_invariant lower < i && i <= upper;
12   //@loop_invariant le_seg(min, A, lower, i);
13   //@loop_invariant A[m] == min;
14   {
15      if (A[i] < min) {
16         m = i;
17         min = A[i];
18      }
19   }
20   return m;
21 }
22
23 void sort(int[] A, int lower, int upper)
24 //@requires 0 <= lower && lower <= upper && upper <= \length(A);
25 //@ensures is_sorted(A, lower, upper);
26 {
27   for (int i = lower; i < upper; i++)
28   //@loop_invariant lower <= i && i <= upper;
29   //@loop_invariant is_sorted(A, lower, i);
30   //@loop_invariant le_segs(A, lower, i, i, upper);
31   {
32      int m = min_index(A, i, upper);
33      //@assert le_seg(A[m], A, i, upper);
34      swap(A, i, m);
35   }
36   return;
37 }
```

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, 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 in the array, some choose a randomly generated index, and some randomly generate three indices, look at the values at those indices, and pick the index of the median value. The different strategies affect the performance of quicksort in different ways.)

After we pick a pivot index, we rearrange the array so that everything to the to the left of the pivot index in the array is less than or equal to `A[pivot_index]` and everything on the right of the pivot index in 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.

Example of Quicksort

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

We split the array so everything on the left is less than or equal to the pivot. After we've done that, we have `[2, 0, 1, 4]`. We then swap the pivot, 2, back to the place where we now know it belongs in the final array – index 2. Therefore we have `[1, 0, 2, 4]` 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, switching index 0 with index 0 keeps the array exactly the same: Our switch then results in the array `[1, 0]`. Then, we can swap the pivot, 1, to the place where it belongs in the final array – index 1.

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

Next, we recursively sort the array `[]` (everything in `[0, 1]` that's larger than `1`). The empty array is already sorted.

Now, we know that `[0, 1]` is sorted, so we can sort `[4]`.

`[4]` is already sorted, so we know our whole array, `[0, 1, 2, 4]` is now sorted.

Now, let's look at the code for quicksort and see what we can say about its correctness:

```
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, upper);
7  {
8   // Hold the pivot element off to the left at "lower"
9   int pivot = A[pivot_index];
10  swap(A, lower, pivot_index);
11
12  int left = lower+1; // Inclusive lower bound (lower+1, pivot's at lower)
13  int right = upper; // Exclusive upper bound
14
15  while (left < right)
16   //@loop_invariant lower+1 <= left && left <= right && right <= upper;
17   //@loop_invariant ge_seg(pivot, A, lower+1, left); // Not lower!
18   //@loop_invariant le_seg(pivot, A, right, upper);
19   {
20    if (A[left] <= pivot) {
21     left++;
22    } else {
23     //@assert A[left] > pivot;
24     swap(A, left, right-1); // right-1 because of exclusive upper bound
25     right--;
26    }
27   }
28  //@assert left == right;
29
30  swap(A, lower, left-1);
31  return left-1;
32 }
33
34 void sort(int[] A, int lower, int upper)
35 //@requires 0 <= lower && lower <= upper && upper <= \length(A);
36 //@ensures is_sorted(A, lower, upper);
37 {
38  if (upper - lower <= 1) return;
39  int pivot_index = lower + (upper - lower)/2; // Pivot at midpoint
40
41  int new_pivot_index = partition(A, lower, pivot_index, upper);
42  sort(A, lower, new_pivot_index);
43  //@assert is_sorted(A, lower, new_pivot_index + 1);
44  sort(A, new_pivot_index + 1, upper);
45 }
```

Checkpoint 0

What would go wrong if the `partition` function ignored `pivot_index` and picked a new pivot?

Checkpoint 1

Why is swapping the pivot with `left-1` the right thing? Why is `left` wrong? Why is `left-1` safe?

Checkpoint 2

Prove that the that `partition` function is correct.

Loop invariants hold initially

Loop invariants are preserved

Negation of loop guard and loop invariants imply postcondition

Checkpoint 3

Could you change `partition` (code and/or loop invariants) in order to justify one of the postconditions on line `5` or `6` being `gt_seg` or `lt_seg`?

Checkpoint 4

Using the `rand` library in C0, modify this code to select a random pivot.

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.

$O(n!)$ $O(n)$ $O(4)$ $O(n\log(n))$ $O(4n+3)$ $O(n^2+20000n+3)$ $O(1)$ $O(n^2)$ $O(2^n)$ $O(\log(n))$ $O(\log^2(n))$ $O(\log(\log(n)))$

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 $k > 0$.

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 1), because of the change of base formula:

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

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