"Cheat Sheet" - Week 3

CS50 — Fall 2012

Prepared by: Doug Lloyd '09

October 1, 2012

Complexity

We measure computational complexity (aka "Big O") by comparing how quickly the number of operations it takes to complete the function or program grows as we increase the size of the input to some arbitrarily large value. This lets us know how the well the algorithm scales with larger problems. When we talk about "input size", we mean whatever makes sense with respect to the algorithm. (For a string, probably the length of the string; for an array, probably the number of elements in the array...) There are many different "Big O" classes; here are a few:

- $O(1)$ - constant time
- $O(\log\ n)$ - logarithmic time
- $O(n)$ - linear time
- $O(n\ \log\ n)$ - supralinear time
- $O(n^c)$ - polynomial time
- $O(c^n)$ - exponential time

Sorting Algorithms

To use binary search, among many other useful things, it's helpful to be sure you have a sorted array. Thus, it's good to know some array sorting algorithms. While sample implementations will not be provided here in the interest of brevity (they can easily be found online), here are basic descriptions of two simple ones that run in $\mathrm{O}(n^2)$:

- **Selection Sort.** Starting from the first element, scan the array for the lowest value. Every time you find a new "lowest value", remember both that value and where in the array you found it. When you reach the end of the array, swap the first element with the "lowest value", putting the first element in the place where the "lowest value" once occupied. Start again at the second element (you know that the first element is now the lowest of any element in the array), and repeat the process until there is only one unsorted element left.
- **Bubble Sort.** Starting from the first and second elements, compare them. If the first element is larger than the second element, swap them. Then consider the second and third elements and repeat, swapping as necessary until you have finished comparing the $(n-1)^{th}$ and n^{th} elements. At this point, you know that the largest element in the array is in position `arr[n-1]`. Then, start over again, this time stopping after you compare the $(n-2)^{th}$ and $(n-1)^{th}$ elements. If you include a variable that keeps track of how many swaps have been made during each pass of the procedure, you can terminate the sort early if you ever notice that the swap counter is 0, as this means the array is sorted and further passes through would be a waste of time.

Linear Search

In linear search, you basically just step through an array, one element at a time from beginning to end, searching for the target element. If you find it, you can stop. If you reach the end of the array, you know–by negative implication–that the target is not part of the array. Linear search runs in $O(n)$.

Binary Search

The concept behind binary search is to eliminate half of the remaining portion of the array to be searched on each pass through of the algorithm. In order to guarantee success, however, your array MUST be sorted first, using one of the previously described sorting algorithms. The basic idea behind binary search is this:

```
Given an array with indexes ranging from A to B
   Calculate the midpoint (A+B/2), rounding up if necessary
   if(array[midpoint] > target)
      search again, setting midpoint+1 = B
   else if(array[midpoint] < target)
      search again, setting midpoint-1 = A
   else if(array[midpoint] == target)
      you have found the target
   else
      the target is not in the array
```

Because binary search cuts out half of the remaining possibilities on each pass, it runs in $O(\log_2 n)$, which we usually just call $O(\log n)$.

Recursion

A recursive procedure is one that calls upon itself in the process of its execution. The idea behind recursion is to turn a larger problem into a series of smaller and smaller problems until we reach a problem that we can easily solve, and that solution informs the solution to the bigger problem. Recursive procedures typically replace loops. Here's an example, using the summation formula (Σ), where `sum()` is the function that calculates the sum of all numbers less than or equal to the argument, but greater than 0.

```
sum(1) =             1
sum(2) =         2 + 1 =         2 + sum(1)
sum(3) =     3 + 2 + 1 =     3 + sum(2)
sum(4) = 4 + 3 + 2 + 1 = 4 + sum(3)
```

How do we code a recursive solution to the problem of calculating, e.g. sum(5727)? We need a base case first - here, that would be `sum(1)`...we know easily that the sum of all numbers in the range $[1, 1]$ is 1. For the recursive case, we'll try to take advantage of what was presented above - that `sum(n)` is equal to `n + sum(n-1)`.

```
int sum(int start) {
   if(start <= 1)
      return 1;
   else
      return start + sum(start-1);
}
```

Aha. Turns out `sum(5727)` is 16,402,128. And it only took four lines of code to get there!