CS 271 (Spring 2013) — Assignment 2
Due: 01/31/2013

This homework is to be done in groups of 3 students, assigned randomly by the program posted on the course web site. Each group can only turn in one solution, with all students' names listed on it. The group composition is posted at `http://www-bcf.usc.edu/~dkempe/CS271/team-assignments-2.html`.

(1) Read Chapter 3. (The material of Section 3.1 is not directly what we cover here, but the section sets the backdrop for our analysis; some of Section 3.3 goes beyond what we'll cover here, but you may find it interesting nonetheless. On the other hand, we'll discuss counting steps a little more precisely.)

(2) Attend a sherpa session this week. Preferably, that would be the one you're assigned to, but it doesn't have to be. This week, sherpas will be taking attendance. Your group will receive credit based on how many of you were at your sherpa sessions.

(3) Find examples in the real world (and outside of the analysis of the running time or memory requirements of algorithms) of the following.

(a) Quadratic growth: That is, find some quantity x that is naturally measured in (non-negative) real numbers or integers, and some other quantity that naturally is of the form $\Theta(x^2)$. Explain where the quadratic growth comes from.

(b) Exponential growth: This time, the first quantity t should be *time*. In other words, find some real-world phenomenon such that its value evolves over time (roughly) as $\Theta(a^t)$ for some $a > 1$. Explain what units you are measuring time in (seconds? years? microseconds?), and give a rough estimate of the value of a. Since exponential growth is very fast, use your form to estimate roughly when in the future (or in the past — you're welcome to consider something that grows exponentially into the past) this exponential growth must stop. If it works for your example, try to explain/guess what real-world constraints/facts would stop exponential growth at that point. (For instance, the explanation could be: "For reason XYZ, the simple mathematical model I used will not be correct any more at that point," or "At that point, the world will indeed have a serious problem," or "The laws of physics will mean that at a particular earlier point, the growth will stop".)

(c) Growth that is naturally neither linear, nor quadratic nor exponential. Give an expression for the growth, and explain where it comes from. (This one does not have to be a function of time.)

(4) Solve the following exercises from the textbook

(a) Section 3.2, Exercises 8, 14, 24, 26, 30, 42

(b) Section 3.3, Exercises 4, 8, 12, 16, 26, 46

(5) [0 points]
Chocolate Problem (2 chocolate bars): One of the things that one frequently needs to do with collections of sets is the following. You have a ground set (or universe U) of size $n = |U|$, and throughout the process, U is partitioned into a bunch of sets $S_1, S_2, \ldots$. (*Partitioned* means that the sets are pairwise disjoint, so $S_i \cap S_j = \emptyset$ whenever $i \neq j$, and $\bigcup_i S_i = U$. So each element of U appears in exactly one set.) You start out with each element of U in a set just by itself. Then, you repeatedly merge sets, until eventually, you are left with just one set $S_1 = U$, which contains all elements. More precisely, what happens is the following: you get a sequence of m requests, where each request k is of the form (u_k, v_k), giving you two elements $u_k, v_k \in U$. Then, if u_k and v_k are already in the same set, you do nothing. But if they are in different sets S_i, S_j, then you merge S_i and S_j into a new set $S_i \cup S_j$, and instead remove S_i and S_j from your collection of sets. You'll hopefully learn in CS271 (or

otherwise in CS303) why this kind of stuff — which probably sounds a little artificial right now — is actually important.

Here is a data structure that lets you implement this quite efficiently. For each element $u \in U$, you keep two things: a number $s[u]$ (which is initialized to 1), and a pointer $p[u]$, which is initialized as the `null` pointer. When you get a request (u_k, v_k), you start from u_k: if its pointer is `null`, you stop there; otherwise, you follow its pointer to a new node. If that node's pointer is `null`, you stop; otherwise, you follow its pointer, and so on, until you end up at a node u'_k whose pointer is `null`. You do the same starting from v_k, arriving eventually at a node v'_k whose pointer is `null`. If $u'_k = v'_k$, you do nothing (because u_k and v_k were already in the same set). Otherwise, you compare $s[u'_k]$ to $s[v'_k]$. If $s[u'_k] \leq s[v'_k]$, then you reassign $p[u'_k] = v'_k$; otherwise, you reassign $p[v'_k] = u'_k$. In either case, you reassign both $s[u'_k]$ and $s[v'_k]$ to $s[u'_k] + s[v'_k]$ (i.e., you sum the previous values).

Show that for any sequence of m merge requests, the total time of this algorithm will be $O(m \log n)$.[1] (Hint: Doing one or two example runs probably will help you. Think first about what the s_u capture, and then think about how long a chain of pointers the algorithm ever has to follow, and why?)

[1] In fact, with a slight optimization to the previous algorithm, this becomes $O(m \log^* n)$, where $\log^* n$ is the number of times you have to take the log starting at n until you hit 1. (That's a *really* slow-growing function.) But analyzing the modified algorithm would be more like a 10-chocolate bar problem.