CS 170 Algorithms Fall 2014 David Wagner HW 7

Due Oct. 24, 6:00pm

Instructions. This homework is due Friday, October 24th, at 6:00pm electronically. Same rules as for prior homeworks. See `http://www-inst.cs.berkeley.edu/~cs170/fa14/hws/instruct.pdf` for the required format for writing up algorithms.

1. (20 pts.) Amortized running time

We want to implement an append-only log. You can think of it as a list that allows you to append to the end. It supports one operation: Append(x), which adds the value x to the end of the list so far.

In memory, the data is stored in an array. If we append enough entries that the array becomes full, we allocate a new array that is twice as large (so we'll have space for the new element) and copy over the old values. In particular, our implementation of Append is as follows:

global variables:
an array $A[]$, integers s, t
initially $t = 0$, $s = 1$, and $A[]$ is 1 element long

Append(x):
1. If $t = s$:
2. Allocate a new array B with $2s$ elements.
3. For $i := 0, 1, \ldots, s-1$: set $B[i] := A[i]$.
4. Set $A := B$ (pointer assignment) and $s := 2s$.
5. Set $A[t] := x$ and then $t := t + 1$.

Line 4 does not copy the arrays element by element; instead, it simply swaps a pointer, so that the name A now refers to the new array just allocated on line 2. Notice that s always holds the size of the array A (the number of elements it can store), and $A[0..t-1]$ holds the elements that have been appended to the log so far. Also $0 \le t \le s$ is an invariant of the algorithm.

(a) Suppose we perform m calls to Append, starting from the initial state. Then, we perform one more call to Append(x). What is the worst-case running time of that last call, as a function of m?

(b) Suppose we perform $m = 2^k$ calls to Append, starting from the initial state. What are the values of t where calling Append causes lines 2–5 to be executed? Use this to argue that the total number of elements copied in line 3, summed across all m of these calls, is $1+2+4+8+\cdots+2^{k-1}$. Then, prove that the total running time of all m of these calls is $\Theta(m)$.

(c) Given your answer to part (b), what is the amortized running time of Append?

(d) Now let's see a different way to analyze the amortized running time of this data structure, using the accounting method. The running time of lines 2–4 (in a single call to Append) is $\Theta(s)$, so we will consider lines 2–4 to cost s dollars. The running time of lines 1 and 5 is $\Theta(1)$, so we will consider the execution of lines 1 and 5 to cost 1 dollar.

Suppose that each time Alice calls Append, she pays us ~~2~~ 3 dollars. If Alice calls Append at a time when $t < s$ (so lines 2–4 are not executed), how much profit do we have left over from this one call to Append?

(e) Suppose whenever we make a profit, we save our money for a rainy day. In particular, whenever Alice calls Append, we'll put our profits from that call next to the array element $A[t]$ that was just set in line 5.

Now suppose Alice calls Append at a time when $t = s$. Which elements of A have some money sitting next to them? How many dollars are there sitting next to the elements of A, in total? If we grab all of those dollars, does that provide enough to pay for the s dollars that it will cost to execute steps 2–4?

(f) For the accounting-based amortized analysis to be valid, we have to be sure that any time Append is called, there is enough money available to pay for its running time (i.e., our total balance will never go negative). Using the method outlined in parts (d)–(e), is this guaranteed?

(g) If Alice makes m calls to Append, she will have paid us a total of ~~$2m$~~ $3m$ dollars. Based on parts (d)–(f), what is an upper bound on the total running time needed to execute all m of those calls to Append?

(h) Based on part (g), what is the amortized running time of a call to Append?

Revised parts (d),(g) on 10/17: Alice pays us \$3 per Append, not \$2.

2. (15 pts.) Proof of correctness for greedy algorithms

A doctor's office has n customers, labeled $1, 2, \ldots, n$, waiting to be seen. They are all present right now and will wait until the doctor can see them. The doctor can see one customer at a time, and we can predict exactly how much time each customer will need with the doctor: customer i will take $t(i)$ minutes.

(a) We want to minimize the average waiting time (the average of the amount of time each customer waits before they are seen, not counting the time they spend with the doctor). What order should we use?

Hint: sort the customers by ____.

(b) Let $x_1, x_2, \ldots, x_n$ denote an ordering of the customers (so we see customer x_1 first, then customer x_2, and so on). Prove that the following modification, if applied to any order, will never increase the average waiting time:

- If $i < j$ and $t(x_i) \geq t(x_j)$, swap customer i with customer j.

(For example, if the order of customers is $3, 1, 4, 2$ and $t(3) \geq t(4)$, then applying this rule with $i = 1$ and $j = 3$ gives us the new order $4, 1, 3, 2$.)

(c) Let u be the ordering of customers you selected in part (a), and x be any other ordering. Prove that the average waiting time of u is no larger than the average waiting time of x—and therefore your answer in part (a) is optimal.

Hint: Let i be the smallest index such that $u_i \neq x_i$. Use what you learned in part (b). Then, use proof by induction (maybe backwards, in the order $i = n, n-1, n-2, \ldots, 1$, or in some other way).

3. (15 pts.) Job Scheduling

You are given a set of n jobs. Each takes one unit of time to complete. Job i has an integer-valued deadline time $d_i \geq 0$ and a real-valued penalty $p_i \geq 0$. Jobs may be scheduled to start at any non-negative integer time (0, 1, 2, etc), and only one job may run at a time. If job i completes at or before time d_i, then it incurs no penalty; otherwise, it is late and incurs penalty p_i. The goal is to schedule all jobs so as to minimize the total penalty incurred.

For each of the following greedy algorithms, either prove that it is correct, or give a simple counterexample (with at most three jobs) to show that it fails.

(a) Among unscheduled jobs that can be scheduled on time, consider the one whose deadline is the earliest (breaking ties by choosing the one with the highest penalty), and schedule it at the earliest available time. Repeat.

(b) Among unscheduled jobs that can be scheduled on time, consider the one whose penalty is the highest (breaking ties by choosing the one with the earliest deadline), and schedule it at the earliest available time. Repeat.

(c) Among unscheduled jobs that can be scheduled on time, consider the one whose penalty is the highest (breaking ties arbitrarily), and schedule it at the latest available time before its deadline. Repeat.

4. (15 pts.) Timesheets

Suppose we have N jobs labeled $1,\ldots,N$. For each job, there is a bonus $V_i \geq 0$ for completing the job, and a penalty $P_i \geq 0$ per day that accumulates for each day until the job is completed. It will take $R_i \geq 0$ days to successfully complete job i.

Each day, we choose one unfinished job to work on. A job i has been finished if we have spent R_i days working on it. This doesn't necessarily mean you have to spend R_i consecutive days working on job i. We start on day 1, and we want to complete all our jobs and finish with maximum reward. If we finish job i at the end of day t, we will get reward $V_i - t \cdot P_i$. Note, this value can be negative if you choose to delay a job for too long.

Given this information, what is the optimal job scheduling policy to complete all of the jobs? Prove your answer.

Revised 10/23 to clarify: please prove that your answer is optimal.

5. (15 pts.) A greedy algorithm—so to speak

The founder of LinkedIn, the professional networking site, decides to crawl LinkedIn's relationship graph to find all of the *super-schmoozers*. (He figures he can make more money from advertisers by charging a premium for ads displayed to super-schmoozers.) A *super-schmoozer* is a person on LinkedIn who has a link to at least 20 other super-schmoozers on LinkedIn.

We can formalize this as a graph problem. Let the undirected graph $G = (V,E)$ denote LinkedIn's relationship graph, where each vertex represents a person who has an account on LinkedIn. There is an edge $\{u,v\} \in E$ if u and v have listed a professional relationship with each other on LinkedIn (we will assume that relationships are symmetric). We are looking for a subset $S \subseteq V$ of vertices so that every vertex $s \in S$ has edges to at least 20 other vertices in S. And we want to make the set S as large as possible, subject to these constraints.

Design an efficient algorithm to find the set of super-schmoozers (the largest set S that is consistent with these constraints), given the graph G.

Hint: There are some vertices you can rule out immediately as not super-schmoozers.

6. (20 pts.) A funky kind of coloring

Let $G = (V,E)$ be an undirected graph where every vertex has degree ≤ 51. Let's find a way of coloring each vertex blue or gold, so that no vertex has more than 25 neighbors of its own color.

Consider the following algorithm, where we call a vertex "bad" if it has more than 25 neighbors of its own color:

1. Color each vertex arbitrarily.
2. Let $B := \{v \in V : v \text{ is bad}\}$.
3. While $B \neq \emptyset$:

4. Pick any bad vertex $v \in B$.
5. Reverse the color of v.
6. Update B to reflect this change, so that it again holds the set of bad vertices.

Notice that if this algorithm terminates, it is guaranteed to find a coloring with the desired property.

(a) Prove that this algorithm terminates in a finite number of steps. I suggest that you define a *potential function* that associates a non-negative integer (the potential) to each possible way of coloring the graph, in such a way that each iteration of the while-loop is guaranteed to strictly reduce the potential.

(b) Prove that the algorithm terminates after at most $|E|$ iterations of the loop.
Hint: You should figure out the largest value the potential could take on.

Optional: Think about how to implement the algorithm so that its total running time is $O(|V| + |E|)$ — this won't be graded.