15-451/651 Algorithms, Fall 2015
Recitation #3 Handout/Worksheet

Amortized Analysis and More Probability

Amortized non-space-hogging arrays: In lecture we discussed the "implementing a stack as an array" problem, and decided that doubling the array whenever it gets full does well in an amortized sense. Now, what if we don't want to be space hogs. If we do a lot of pops and the stack gets small, then we want to resize the array to make it smaller.

Say k = number of elements currently in the stack, and L = the current size of the array. Say the cost for performing a resize operation is equal to the number of elements moved. As before, if we perform a push on a stack with $k = L$ we will resize into an array of size $2L$ (paying a cost of L). Let's now consider two strategies for making these arrays non-space-hogging:

1. Suppose we cut the array size in half whenever we perform a pop on a stack with $k = L/2$. Is this a good strategy? How large could the amortized cost per operation be in a sequence of n operations?

   Solution: To make this cleaner, assume k is the number of elements in the stack *after* the pop. If n is a power of 2, a bad sequence would be $n/2$ pushes followed by a sequence of push/pop. This would have $n/2$ ops of cost $n/2$ for an amortized cost of at least $n/4$ per operation.

   If n is not a power of 2, let $m = 2^{i-1}$ where 2^i is the largest power of 2 that is $\leq n$. Consider m pushes followed by $m - n$ push/pop. This would have $\geq n/2$ ops of cost $\geq n/4$ for an amortized cost of at least $n/8$ per operation.

   If we define k to be the number of elements in the stack before the pop then the bad sequence has push/push/pop/pop, but still $\Omega(n)$ amortized cost per operation.

2. Suppose instead we cut the array size in half when we pop on a stack with $k = L/4$. Claim is that this now has an amortized cost of at most 3 per operation. Proof?

   Solution: Consider the interval between successive array resizes. The important point is that after each resize operation, the array is exactly half full. If it's a double, then there must have been at least $L/2$ pushes since the last resize. This can pay for the work. If it's a halving, then there must have been at least $L/4$ pops, and these can be used to pay for the resize.

   If you want to use the bank-account method, you can think of each push or pop as paying \$1 to perform the operation and putting \$2 in the bank to pay for a possible future resizing.

Slower resizing: Let's go back to the original case where we just increase the size of the array when we push into a full array. Suppose that instead of doubling, we do the following. We begin with an array of size 1. The first time we resize, we add 1 to the length. The second time, we add 2. The third time we add 3. And so on. What is (in Θ notation) the amortized cost per operation for n pushes?

Solution: $\Theta(\sqrt{n})$.

The total number of resizes is $O(\sqrt{n})$, so the total time spent doing resizing is upper bounded by $O(n\sqrt{n})$, giving an $O(\sqrt{n})$ upper bound on amortized cost. But we also have done $\Omega(\sqrt{n})$ resizes ever since the array had $n/2$ elements, so the total time is lower-bounded by $\Omega(n\sqrt{n})$, giving an $\Omega(\sqrt{n})$ lower bound on amortized cost.

Another way to argue, if we just sum up, is that the total resizing cost $= 1 + (1+2) + (1+2+3) + ... + (1+2+3+...+k)$ where $k \approx \sqrt{2n}$ is the smallest integer such that $1+2+3+\ldots+k \geq n$ (i.e., $n \approx k^2/2$).

Las Vegas and Monte Carlo:

A *Las Vegas* algorithm is a randomized algorithm that always produces the correct answer, but its running time $T(n)$ is a random variable. That is, sometimes it runs faster and sometimes slower, based on its random choices; for instance, quicksort when choosing a random pivot, or treaps. A *Monte Carlo* algorithm is an algorithm with a deterministic running time, but that sometimes doesn't produce the correct answer. For example, you may be familiar with randomized primality testing algorithms, that given a number N will output whether it is prime or not and be correct with probability at least 99/100, say.

Question: Show that if you have a Las Vegas algorithm with expected running time $\mathbf{E}[T(n)] \leq f(n)$, then you can get a Monte Carlo algorithm with (a) worst-case running time at most $4f(n)$ and (b) probability of success at least 3/4.

Solution: Consider the algorithm: run the LV algorithm for $4f(n)$ steps, and if it has not stopped yet, just abort it. Trivially, the running time is now upper bounded by $4f(n)$. What is the chance it was aborted?

This equals Pr[the algorithm did not stop by $4\times$ its expected run-time] $\leq 1/4$, by Markov's inequality.

Gory details: What's the random variable? X = run-time of the algo. Clearly a non-negative r.v. Also, we are given that $\mathbf{E}[X] \leq f(n)$. Therefore $\mathbf{Pr}[X > 4f(n)] \leq \mathbf{Pr}[X > 4\mathbf{E}[X]] \leq 1/4$.