15-451 Algorithms, Spring 2016
Recitation #3 Worksheet

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: First, for simplicity suppose 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 $n/2$ length sequence of alternating pushes and pops. Each one of the latter $n/2$ operations would cost at least $n/2$. So 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? (Ideally, also give a proof using the banker's method.)

   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.

Potential Function: Note that the state of the algorithm is given by the pair $s := (k, L)$. Now consider the potential function

$$\Phi(s) := 2|k - L/2|.$$

Remember that $ac_i := c_i + (\Phi(s_i) - \Phi(s_{i-1})$. Show $ac_i \leq 3$ for each operation.

1. A push operation (without resize):

   Solution: The real cost is 1. The potential increases by 2. So $ac_i = 1 + 2 = 3$.

2. A pop operation (without resize):

   Solution: The real cost is 1. The potential *decreases* by 2. So $ac_i = 1 - 2 = -1 \leq 3$.

3. A push operation (that causes a resize):

   Solution: Say we change the size from L to $2L$. The potential at the beginning was $\Phi(L, L) = 2(L - L/2) = L$. The final potential is $\Phi(L+1, 2L) = 2(L+1-(2L/2)) = 2$. And the real cost is $c_i = L + 1$. So $ac_i = (L + 1) + (2 - L) = 3$.

4. A pop operation (with resize):

   Solution: Say we go from $(k = L/4, L)$ to $(L/4 - 1, L/2)$. So the initial potential is $2|L/4 - L/2| = L/2$. The final potential is $2|(L/4 - 1 - (L/2)/2)| = 2$.

   The real cost is $1 + (L/4)$, $L/4$ for the copying over and 1 for the pop. The potential *decreases* by $L/2 - 2$. And L better be at least 4, since we had $k = L/4$ elements to start off. So $ac_i = (1 + L/4) - (L/2 - 2) = 3 - (L/4) \leq 3$.

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 $= 1+(1+2)+(1+2+3)+ \ldots +(1+2+3+\ldots+k)$ where $k \approx \sqrt{2n}$ is the smallest integer such that $1+2+3+\ldots+k \geq n$ (i.e., $n \approx k^2/2$).