Matthew Feng September 28, 2018

6.046 Problem Set 3

Collaborators: *Priscilla Wong, Katherine Xiao*

Problem 1

(a)

For any deck ordering D that Ben selects, the adversarial audience, having knowledge of D, can force Ben into using $\Omega(n^2)$ draws.

First we notice that having $D = (all\ Rs)(all\ Bs)$ (or vice versa) and an audience ordering of $BRBR...BR$ (i.e. alternating card color requests) forces Ben to use $\Omega(n^2)$ deck draws, because for every request, he has to move $n - \lceil (i-1)/2 \rceil$ cards from the top of the deck to the bottom of the deck for the ith request. With n requests, the sum of draws is $\Omega(n^2)$.

Now, the audience can force the ordering of the deck to be the above case while still ensuring that $\frac{n}{2}$ cards of both colors remain in the deck. To do this, the audience splits the deck into two halves — the front half X and the back half Y. We count the number of red and black cards in X, denoted r_X and b_X respectively, as well as the number of red and black cards in Y, denoted r_Y and b_Y, respectively.

WLOG assume we have $b_X > r_X$, which implies that $r_Y > b_Y$. We notice that if we order $r_X + 1$ red card requests at the front of our audience, then we will force b_X black cards to the back of Ben's deck. Y will now be at the top of the deck. Likewise, we can have b_Y black card requests follow the $r_X + 1$ red card requests, which will force r_Y red cards to the back of Ben's deck. Because $b_X > n/2$ and $r_Y > n/2$, and they are blocked together, we have constructed the worst case deck ordering for Ben, which we know is $\Omega(m^2)$, where m the number of each color remaining. Since $m = \Omega(n/2)$, we have $\Omega(\Omega(n/2)^2) = \Omega(n^2)$.

Because in the worst case the online algorithm will require $\Omega(n^2)$ operations, while the offline optimal algorithm will be able to run in $O(n)$ operations, the competitive ratio of Ben's deck ordering is $\Omega(n^2)/O(n)$, or never better than n-competitive.

(b)

Ben can achieve drawing at most $O(n)$ cards for any audience ordering if he is able to move cards from the bottom of the deck to the top of the deck and vice versa if he maintains the heuristic that, WLOG, he always has a red card at the top of the deck and a blue card at the bottom of the deck. If he maintains this property, moving a card from the top of the deck to the bottom of the deck if it isn't red, and moving a card from the bottom of the deck to the top if it isn't black, then Ben will always end up with a block of red cards at the top of the deck and a block of black cards at the bottom of the deck.

WLOG, let us define an inversion as the number of black cards B that are located in the top half of the deck. Every time we move a card from the front to the back, or the back to the front, we are fixing one inversion. There are at most n inversions at the start of the shuffle, and so Ben requires at most $O(n)$ draws from top to bottom or bottom to top in order to fix all the inversions.

Because Ben has an $O(n)$ strategy for any ordering of the audience, and the optimal offline strategy is $O(n)$, we know that Ben's strategy is α-competitive, for some constant $\alpha > 1$.

Problem 2

Our data structure D will have multiple fields. First, we will have a two dimensional matrix $D.coins$, which will hold the same values as G (i.e. 0 if no coin, 1 if coin), with the addition that if we have visited cell $D.coins[i, j]$, then $D.coins[i, j]$ is marked with $*$. D will also have a counter $D.c$ that keeps track of the number of coins we have yet to collect.

initialize(G) We can initialize D in $O(mn)$ time by copying G into $D.coins$, and while doing so, counting the number of coins on the grid G and saving that value to $D.c$. We look at every cell in our grid exactly once, so initialization takes $O(mn)$ time.

numCoins We can simply return the value of our counter $D.c$. This is a $O(1)$ operation.

removeCoin(i, j) We can remove the coin at (i, j) by setting $D.coins[i, j] = *$. We also must decrement $D.c$ by 1. This takes at most $O(1)$ time.

nextCoin(i, j) We notice that we never need to explore a cell more than once, because robot paths never intersect (which was a proof in lecture if I recall correctly). Thus, our $nextCoin(i, j)$ routine is very simple: starting from (i, j), scan down the column j to look for coins. Mark all scanned cells with a $*$. If we don't find any coins, increment j to scan down the next column starting at row i, and repeat until we find a coin. If we reach (m, n) without finding a coin, return "error". Furthermore, if we encounter a $*$ when scanning down a column, immediately increment j and start scanning the next column. Because we only explore each cell at most once, because we don't explore once they are marked as $*$, in aggregate this routine takes $O(mn)$.

However, we need to demonstrate that this algorithm is correct. We can see this because it scans all possible locations that the next coin may be located (to the right and to the bottom), in a order that finds the leftmost, topmost valid coin (scan from top to bottom, starting from the left), which is what we desired.

The runtime of our original algorithm was $O(rmn)$ because each call to $nextCoin$ took $O(mn)$ time; however, because with data structure D, these calls in aggregate cost $O(mn)$, the total run time of the PeelOff(G) algorithm is $O(mn)$.

Problem 3

(a)

We can implement the Update(i_1, i_2, x) and PieTime() operations using a Union-Find data structure represented as a **forest of trees**.

In particular, we note that if we know the ratio $r_{(a,b)}$ between ingredients a and b, and we know the ratio $r_{(b,c)}$ between ingredients b and c, then we know the ratio $r_{(a,c)}$ between a and c as $r_{(a,c)} = r_{(a,b)}r_{(b,c)}$. What this tells us is that we can think of the ingredients as nodes in an undirected graph $G = (V, E)$, and every time we learn the ratio between ingredient $i_1, i_2 \in V$, we add edge (i_1, i_2) to E.

More importantly, however, we want to keep track of the **connected components** in our graph G. This is because as long as there is a path between two nodes i_1, i_2, we can compute the ratio between the two ingredients through successive multiplications.

We learned in class how to create a Union-Find data structure that allows Union(x, y) and Find-Set(x) in amortized $O(\alpha(n))$. We augment this data structure with a counter C that counts the number of connected components in G, and hashset I that keeps track of the different ingredients we have in our recipe.

To implement UPDATE($i_1,\ i_2,\ x$), we first check to see if $i_1, i_2 \in I$, which we can do in expected $O(1)$ time. If an ingredient is not yet in our graph, we call MAKE-SET(i) for that ingredient, add i to I, and increment C for new ingredient we add, all of which are $O(1)$ operations. Next, we check that i_1 and i_2 are not part of the same connected component with by checking that FIND-SET(i_1) $\neq$ FIND-SET(i_2). If i_1 and i_2 are in separate connected components, then we decrement C by 1, because there will be one fewer connected component after the union. Calling the FIND-SET(i) operation twice costs $O(\alpha(n))$ amortized. Finally, we call UNION($i_1,\ i_2$), which will connect the connected components i_1 and i_2 belong to, together. This will also cost $O(\alpha(n))$ amortized.

To check if we are able to bake the pie via PIETIME(), we only need to check if the number of connected components we have is 1. In other words, return whether or not $C = 1$. This takes $O(1)$ time.

Note that this implementation does not store the actual ratio information; the data structure only tells us that we *have enough knowledge* to compute the ratio between two ingredients — it doesn't help us compute that ratio (which is left for part (b)).

(b)

In order to implement GETRATIO($i_1,\ i_2$) in amortized $O(\alpha(n))$ time, we augment every node into our forest of trees with a value $node.r$, which stores the ratio between the ingredients i and its parent $i.parent$, i.e. $i.r = q(i)/q(i.parent) = \rho(i, i.parent)$. Our goal is to use the traversal up the tree during the calls to FIND-SET(i_1) and FIND-SET(i_2) to compute $r_1 = \rho(i_1, ref[i_1])$ and $r_2 = \rho(i_2, ref[i_2])$, and, assuming $ref[i_1] = ref[i_2]$ (if not, just return that we don't know the ratio between the two), computing and returning back to the user $\rho(i_1, i_2) = r_1/r_2$.

In order for our data structure to function correctly, we need to ensure that for every ingredient, we are able to compute $i.r = \rho(i, i.parent)$. We will show that we can update this value throughout any call to FIND-SET and UNION.

When UNION($i_1,\ i_2,\ x$) is called, WLOG assume $i_2 = i_1.parent$, due to the heights of their corresponding subtrees. Then $i_1.r = q(i_1)/q(i_2) = x$, and $i_2.r$ is unchanged.

When we use FIND-SET(i) internally, we need to make sure that we update the ratios of the nodes along the path we traverse, because we reassign their parents to flatten the tree. This is easily doable, however. Suppose our path up to $ref[i]$ is $(i_1, i_2, ..., i_k)$, such that $i_n.parent = i_{n+1}, \forall n \in [1, k-1]$. As we traverse up the path, we keep track of $r_n = q(i_1)/q(i_n)$ for every node in the path, and compute $r_k = q(i_1)/q(i_k)$ at the end of the traversal. We are then able to calculate and update the new values for $i_n.r = q(i_n)/q(i_k) = r_k/r_n$ at the same

time we update the parent pointers. Since we are simply adding $O(1)$ operations to each step of the FIND-SET and UNION function calls, we don't change the asymptotic time complexity of those functions, and so we are able to implement GETRATIO(i_1, i_2) in amortized $O(\alpha(n))$.