Diego Escobedo February 26, 2020

6.046 Problem Set 3

Collaborators: *Ritaank Tiwari, Abhishek Mohan*

Problem 1

(A) Assuming m > k, then we can design an adversarial sequence of books that will make Cam's cost tend to infinity as n tends to infinity. The adversarial sequence looks like $[1, 2, ..., k-1, k, k+1, k, k-1, ..., 2, 1]$ and so on and so forth. Our claim is that this sequence of length n has a cost that is $O(n)$. The reason it is is that once we get all k books filled, and we are getting the k+1th book, we drop the kth book. So, if the very next book we get is the kth book, we have to go back to the library and drop the k-1th book. Further, if the next book is the k-1th book, we have to go to the library, and so on and so forth. Therefore, this sequence guarantees that we need to perform cost 1 operations at every step of the sequence. This means that for a sequence of length n, the worst-case runtime is O(n), which tends to infinity as n goes to infinity.

(B) Similar to last week, we will do a proof by cases. First, lets think of what we are trying to prove. We are trying to prove that for a sequence of inputs R, $\text{Cost}_{CAM}(R) \leq k*$ $\text{Cost}_{OPT}(R)$. If we use instead the amortized cost, we can instead state that we want to prove that $\hat{C}_{CAM}(R) \leq k * C_{OPT}(R)$, which we can re-write due to it being a telescoping series as $(\sum_i C_{CAM}(i)) + \Phi(i = |R|) - \Phi(i = 0) \leq k * C_{OPT}(R)$. Taking this to an individual level, we can try to prove that $\hat{C}_{CAM}(i) \leq k * C_{OPT}(i)$, which we can also express as $C_{CAM}(i) + \Phi(i) - \Phi(i-1) \leq k * C_{OPT}(i)$. So, let's look at the case and prove that this is true for all i. This is where the cases come in:

1. **Neither one goes to the library** In this case, the potential function does not change because Cam's set of books compared to Opal's does not change. Further, the cost of not going to the library is 0. Thus, $C_{CAM}(i) + \Delta\Phi \leq k * C_{OPT}(i)$ becomes $0 + 0 \leq k * 0 = 0$. Since this is true, this case holds.
2. **Cam has to go to the library, Opal does not** Within this case, there are two cases.
    - Opal does not have the book that Cam returns: In this case, since the book is at the bottom, the potential function decreases in the worst case by 1 (this means that all the other books in Cam's stack are books in common, and we only

decrease the potential through giving back a book at the bottom of the stack). This is because the new book that Cam picks up is already owned by Opal.

- Opal does have the book that Cam returns: In this case, since we are shifting the rest of the books down, the potential function decreases by 1*(number of books $\in$ S that shifted down). We know that this must be at least 1, because Cam just gave away a book that Opal had and the new book he picked up was already in Opal's possession. So, in the worst case, the potential function decreases by only 1.

We can see here that the change in the potential function is, in the worst adversarial case, -1. The cost of Cam going to the library is 1, and the cost for Opal is 0. So, lets plug this worst case scenario in. Thus, $C_{CAM}(i) + \Delta\Phi \leq k * C_{OPT}(i)$ becomes $1 + -1 \leq k * 0 = 0$. Since this is true, case this holds.

3. **Opal has to go to the library, Cam does not** The cost of not going to the library is 0 for Cam, whereas it is 1 for Opal. In this case, the potential function does change because Cam's set of books compared to Opal's does change. In the worst case, a new element was added to S (aka a book that both of them had was returned by Opal), and this element has a new weight that will change the potential function. However, the maximum weight it could have is k (because of the definition of our w(b) function). So, lets plug this worst case scenario in. Thus, $C_{CAM}(i) + \Delta\Phi \leq k * C_{OPT}(i)$ becomes $0 + k \leq k * 1 = k$. Since this is true, case this holds.

4. **Both have to go to the library** In this case, the worst-case scenario is that the potential function does not change because Cam's sequence of books compared to Opal's does not change (best case scenario is that the potential function would decrease). This is the worst case because since Cam has to give back a book, he cannot possibly increase the set of books that he has and Opal does not (because Opal is also getting the same book as him). Thus, $C_{CAM}(i) + \Delta\Phi \leq k * C_{OPT}(i)$ becomes $1 + 0 \leq k * 1 = k$. Since this is true, this case holds.

(C) What would change would be case 2, subcase 2. This is the case where Cam has to go to the library, and Opal has the book that Cam returns. Since we are no longer moving the books down, the potential function does not change. This means that the amortized cost is 1, and Opal's cost is 0, which means that the case does not hold here.

(D) The intro for these cases is the same as before. The only thing that was changed is that now our weight function is $w(b_{left}) = k$ and $w(b_{right}) = 1$ So, lets try to figure out this proof by cases.

1. **Neither one goes to the library** In this case, the potential function does not change because Amy's set of books compared to Opal's does not change. Further,

the cost of not going to the library is 0. Thus, $C_{AMY}(i) + \Delta\Phi \leq k * C_{OPT}(i)$ becomes $0 + 0 \leq k * 0 = 0$. Since this is true, this case holds.

2. **Amy has to go to the library, Opal does not** Within this case, there are two cases.

   - Opal does not have the book that Amy returns: In this case, since the book is all the way to the right, the potential function decreases in the worst case by only 1 (this means that all the other books in Amy's stack are books in common, and we only decrease the potential through giving back a book to the right of the stack). This is because the new book that Amy picks up is already owned by Opal.
   - Opal does have the book that Amy returns: In this case, since we are shifting the rest of the books to the right, the potential function decreases by 1*(number of books $\in$ S that shifted to the right). We know that this must be at least 1, because Amy just gave away a book that Opal had and the new book he picked up was already in Opal's possession. So, in the worst case, the potential function decreases by only 1.

   We can see here that the change in the potential function is, in the worst adversarial case, -1. The cost of Amy going to the library is 1, and the cost for Opal is 0. So, lets plug this worst case scenario in. Thus, $C_{CAM}(i) + \Delta\Phi \leq k * C_{OPT}(i)$ becomes $1 + -1 \leq k * 0 = 0$. Since this is true, case this holds.

3. **Opal has to go to the library, Amy does not** The cost of not going to the library is 0 for Amy, whereas it is 1 for Opal. In this case, the potential function does change because Amy's set of books compared to Opal's does change. In the worst case, a new element was added to S (aka a book that both of them had was returned by Opal), and this element has a new weight that will change the potential function. However, the maximum weight it could have is k (because of the definition of our w(b) function, which would imply the Opal returned was the book at the rightmost end of Amy's bookshelf). So, lets plug this worst case scenario in. Thus, $C_{CAM}(i) + \Delta\Phi \leq k * C_{OPT}(i)$ becomes $0 + k \leq k * 1 = k$. Since this is true, case this holds.

4. **Both have to go to the library** In this case, the worst-case scenario is that the potential function does not change because Amy's sequence of books compared to Opal's does not change (best case scenario is that the potential function would decrease). This is the worst case because since Amy has to give back a book, she cannot possibly increase the set of books that she has and Opal does not (because Opal is also getting the same book as her). So, the worst-case change in potential is 0. Thus, $C_{CAM}(i) + \Delta\Phi \leq k * C_{OPT}(i)$ becomes $1 + 0 \leq k * 1 = k$. Since this is true, this case holds.

b

Problem 2

For this problem, we are going to use a union-find data structure, augmented with a few things. First, all the student nodes are going to have two properties in addition to the self.parent property. Firstly, they are going to have a property called self.ratio_to_parent. This is going to be the ratio of food $\frac{f(node)}{f(parent)}$. Secondly, we are going to keep track of self.size, which is the number of nodes below this current node. For this to work, we are going to define this one operation, called FIND-SET-MODIFIED(s):

```
def find_set_modified(s):
    update_ratio(s)
    return find(s)

def find(node):
    if node.parent != node:
        node.parent = find(node.parent)
    return node.parent

def update_ratio(node):
    if node.parent != node:
        node.ratio_to_parent *= update_ratio(node.parent)
        #the ratio of initialized nodes aka representative nodes
        #should start and stay at 1, which makes this step work
    return node.ratio_to_parent
```

In any situation where we would normally use FIND-SET(s), we are now going to use FIND-SET-MODIFIED(s), because it allows us to update the ratios to the parents before moving the pointers to the representative. Using this method, we will guarantee that before we relink pointers we are keeping our ratios accurate. In terms of runtime, this runs in $O(\alpha(n))$; we know that the find operation within takes $O(\alpha(n))$. However, we have to factor in teh time for the update ratio operation. However, since this recursive function is performing the exact same sequence of steps, and just performing a multiplication at each step. Therefore, the total runtime would be $O(2 * \alpha(n))$ which is just $O(\alpha(n))$. We will use this proof of the runtime later.

Let us analyze the operations required individually, proving how to maintain our augmentations and keeping all operations within the required time. The format of this will be that I will explain the algorithm and wh its correct, and then analyze the runtime separately:

1. **Update(s_1,s_2,x)**: Our update function is going to be composed of two procedures.

(a) The first thing we have to do is perform FIND-SET-MODIFIED(s_1) and FIND-SET-MODIFIED(s_2). This is to check if they have the same representative element. If they do, we don't actually need to do anything, because they are in the same subset of nodes and the ratio between them is implied and can be extracted in amortized constant time. If they do not have the same representative element, then we can proceed to step 2.

(b) Now, we will perform the UNION(s_1, s_2). Remember that this is composed of FIND-SET-MODIFIED(s_1), FIND-SET-MODIFIED(s_2), and then LINKING($Rep[s_1], Rep[s_2]$). We have already covered the subtleties of the two find-sets, so lets focus on the linking. We are obviously using the smaller into larger improvement, so assume we are linking (w.l.o.g.) $Rep[s_2]$ into $Rep[s_1]$. To maintain our size augmentation, we must add $Rep[s_2]$.size to $Rep[s_1]$.size. Since any time a new element enters a subset it will update the representative element's size, we do not need to worry about all the non-representative's sizes. To maintain our ratio augmentation, we prove in the Get Ratio operation that we can easily determine the ratios $\frac{f(s_1)}{f(Rep[s_1])}$ and $\frac{f(s_2)}{f(Rep[s_2])}$. Since we are linking (w.l.o.g.) $Rep[s_2]$ into $Rep[s_1]$, if we multiply together the reciprocal of $\frac{f(s_2)}{f(Rep[s_2])}$ times the reciprocal of $\frac{f(s_1)}{f(s_2)}$ times $\frac{f(s_1)}{f(Rep[s_1])}$, we will get $\frac{f(Rep[s_2])}{f(Rep[s_1])}$, which is the ratio we want to store in $Rep[s_2]$.ratio_to_parent. This completes all we need to do to do the union between s_1 and s_2, maintain all our augmentations, and use the ratio given to compress paths in our tree further.

- Runtime: Let us analyze the cost of all these operations individually. In (a), we perform two FIND-SET-MODIFIED operations, which is a constant number of $O(\alpha(n))$ operations. In (b), we perform a UNION, which we proved in lecture is $O(\alpha(n))$ (even though we switched out the find function for a custom one it has the same asymptotic runtime, so it stays the same). Therefore, since we are performing a constant number of $O(\alpha(n))$ operations, the runtime for **Update(s_1,s_2,x)** is $O(\alpha(n))$.

2. **GetRatio(s_1,s_2)**: Our get ratio operation is going to be composed of two steps:

(a) First, we will perform FIND-SET-MODIFIED(s_1) and FIND-SET-MODIFIED(s_2). If the representative element for these two find-sets is *different*, then we can immediately terminate, since we do not have enough information to calculate the ratio between these two students.

(b) If the representative elements are the same, then we can indeed calculate the ratio. Assume the representative element for these two elements is x. So, we can calculate the ratio of $\frac{f(s_1)}{f(x)}$ by climbing up the parent nodes of s_1 and multiplying all the self.ratio_to_parent. This will work because the elements will all cancel out as we multiply them in sequence, except for the first numerator and the last denominator, which would leave us with $\frac{f(s_1)}{f(x)}$. We do the same thing to find $\frac{f(s_2)}{f(x)}$. Then, if we multiply $\frac{f(s_1)}{f(x)}$ by the reciprocal of $\frac{f(s_2)}{f(x)}$, we will get $\frac{f(s_1)}{f(s_2)}$.

- Runtime: Let us analyze the cost of all these operations individually. In (a), we perform two FIND-SET-MODIFIED operations, which is a constant number of $O(\alpha(n))$ operations. In (b), we 'climb up' to the representative nodes twice. However, we know that following parent pointers ot the representative element is the same sequence of steps as a FIND-SET-MODIFIED, even though it is now doing a different operation at each recursive step (multiplying instead of relinking). However, the cost is still $O(\alpha(n))$ because of the number of recursive steps we have to take. Therefore, the runtime for **GetRatio(**s_1**,**s_2**)** is $O(\alpha(n))$.

3. **GetRatio(**s**,**sf**)**: Our SetupFood function has a single step. First, we need to do a FIND-SET-MODIFIED(s) to get the representative of s (call it x). Then, we check if x.size is equal to the total number of students (which we should have access to). If so, then we return True. Otherwise, we return False.

   - Runtime: Let us analyze the cost of all these operations individually. We perform a single operation that is $O(\alpha(n))$. Therefore, the runtime for **GetRatio(**s**,**sf**)** is $O(\alpha(n))$.