CS3230: Design and Analysis of Algorithms
Semester 2, 2021-22, School of Computing, NUS

Solutions of Practice Problem Set on Amortized Analysis

April 18, 2022

Question 1: You are asked to maintain a list L while supporting the following two operations:

- Insert(x, L): Insert an integer x in the list L (cost is 1).
- ReplaceSum(L): Compute the sum of all the integers in L. Then remove all these integers, and finally just store the computed sum in L (cost is length of the list L).

(a) Use the accounting method to show that the amortized cost of both Insert and ReplaceSum operation is $O(1)$.

(b) Use the potential method to show that the amortized cost of both Insert and ReplaceSum operation is $O(1)$.

Answer: Consider the number of elements in the list as your potential function ϕ. At the beginning when the list is empty $\phi = 0$, and by definition at any point ϕ is the number of elements in the list which is non-negative.

After each insertion operation, the number of elements increases by 1, therefore the potential difference is also 1. Actual cost of Insert is $O(1)$. Hence the amortized cost (=actual cost + potential difference) is also $O(1)$.

After each ReplaceSum(L) number of elements decreases by $\ell - 1$ (let ℓ be the number of elements in the list before the operation). Therefore the potential difference is $1 - \ell$. Since actual cost is ℓ the amortized cost is again $O(1)$.

(Note, in the above analysis if somebody considers the actual cost of ReplaceSum(L) to be 2ℓ (which is more accurate) then he has to take potential function to be $2\times$ the number of elements in the list. The whole analysis remains the same.)

Question 2: Let us consider the following *Online List Problem*. In this problem we are given a doubly linked list L containing n items (you may consider them to be distinct), and a sequence of m search operations $s_1, s_2, \cdots, s_m$ with the following restrictions:

- Each search operation s_t will look like: Given an item e_t you have to correctly say whether e_t is in the list L or not.
- The sequence of search operations will arrive in online fashion, i.e., at time t you will get a request for the operation s_t and you have to answer correctly. (One important thing to note, you cannot see future search queries.)
- You can access the linked list only through the HEAD pointer. So if at time[1] t you are asked to search an item e_t which is the third element in the list at that moment, then you have to pay 3 unit cost to find that element.
- During answering a search query you may update the list. However, at any point you can only swap two neighboring elements in the list, and each such swap will cost you one unit. (So during any search operation if you will perform k swaps, you will have to pay additional k unit cost.)

[1]Do not get confused between time and cost. Here I use the term *time* to identify which operation you are going to serve at that particular moment, whereas the cost captures the complexity of each operation.

Now keeping all these restrictions in mind, ideally we would like to design an algorithm that achieves minimum total cost.

Ooh, that must be a very difficult task! So to make your life simpler, I am not asking you to design an algorithm with the minimum cost. Instead I ask you to consider the following simple heuristic algorithm: Whenever we search an item in the list, if we find it we bring that in front of the list (by performing a sequence of swaps).

You may wonder since this is an extremely simple heuristic strategy, its total cost will no way be close to the minimum. However, I have a different opinion. Suppose if all the m search operations were given in advance (instead of arriving in online fashion) then the minimum cost that one (optimum algorithm) could achieve is C_{opt}. I claim that the total cost of the above heuristic algorithm is just at most $4C_{opt}$. If you answer the following series of questions, you will also be able to prove this claim.

(a) During the search operation s_t suppose you are asked to search for the item e_t. Let r and r^* be the rank of e_t in the list maintained by the heuristic algorithm (denoted by L) and the optimum algorithm (denoted by L^*) respectively. Further suppose the optimum algorithm performs t^* swaps during the search operation s_t. What is the difference between the cost to perform s_t by the heuristic and the optimum algorithm?

Answer: The cost to search the item by the optimal algorithm is r^*, and so the overall actual cost is $r^* + t^*$. Similarly the overall actual cost for the heuristic algorithm is $r + (r-1) = 2r - 1$, where r is for the search and $r-1$ for those many swaps. So the difference is $2r - r^* - t^* - 1$.

(b) Let us define the *distance* between two lists L and L' containing exactly the same items (but in different order) as number of pairs (x, y) such that relative ordering of x and y in L and L' are different. For example, if $L = 10, 20, 30, 40$ and $L' = 10, 30, 40, 20$, then the distance between them is 2 (since $(20, 30), (20, 40)$ are the pairs with different relative ordering in L and L'). Show that $\phi(t)$, defined as the twice the distance between L and L^* at time t, is a valid potential function.

Answer: At the beginning both the lists (one maintained by the optimal algorithm and another maintained by the heuristic one) are same, and so $\phi = 0$. By the definition of the distance it can never be negative and hence at any point of time $\phi \geq 0$. So ϕ is a valid potential function.

(c) Show that during performing s_t by the heuristic algorithm, each swap changes the distance between L and L^* by at most one.

Answer: Suppose we are swapping items e_i and e_j where e_i appears before e_j. We know that such a swap can happen only between two consecutive elements in the list. So relative ordering of only these two elements will change. Either the distance will increase by 1 (if e_i appears before e_j in the optimum list) or decrease by 1 (if e_i appears after e_j in the optimum list).

(d) Next prove that the change in potential from t-th time to $(t+1)$-th time is at most $4r^* + 2t^* - 2r - 2$.

Answer: Note, during each operation s_t we will move exactly one element e_t in the heuristic algorithm to the front. Since rank of e_t in the list of optimum algorithm is r^*, its movement can increase the distance by at most $r^* - 1$. Since r is the rank of e_t in the list maintained by the heuristic algorithm, this movement also leads to a decrease in distance due to change in relative ordering with at least $r - r^*$ many elements (which appears before e_t in the heuristic algorithm list but after the optimum algorithm list). t^* swaps by the optimum algorithm can increase the distance by at most t^*.

So the overall change in potential is at most $2(r^* - 1 + t^* - (r - r^*)) = 4r^* + 2t^* - 2r - 2$.

(e) Now complete the amortize analysis using the above potential function to show that the total cost of the above mentioned heuristic algorithm is at most $4C_{opt}$.

Answer: The amortized cost of operation s_t is actual cost + potential difference, which is at most $(2r - 1) + (4r^* + 2t^* - 2r - 2) \leq 4(r^* + t^*) = 4 \times$ Actual cost of optimal algorithm for performing s_t.

Hence the total cost of all the operations by the heuristic algorithm is at most $4C_{opt}$.

Note, what you have just seen is called the *competitive analysis* in the domain of online algorithm. In the above question you have finally shown that your online algorithm is 4-competitive. We will see more online algorithms (and competitive analysis) in the last two lectures of this course.

Question 3: In the Set Union problem we have n elements, that each are initially in n singleton sets, and we want to support the following operations:

- Union(A, B): Merge the two sets A and B into one new set $C = A \cup B$ destroying the old sets.

- SameSet(x, y): Return true, if x and y are in the same set, and false otherwise.

We implement it in the following way. Initially, give each set a distinct color. When merging two sets, recolor the smaller (in size) one with the color of the larger one (break ties arbitrarily). Note, to recolor a set you have to recolor all the elements in that set. To answer SameSet queries, check if the two elements have the same color. (Assume that you can check the color of an element in $O(1)$ time, and to recolor an element you also need $O(1)$ time. Further assume that you can know the size of a set in $O(1)$ time.)

Use Aggregate method to show that the amortized cost is $O(\log n)$ for Union. That means, show that any sequence of m union operations takes $O(m \log n)$ time. (Note, we start with n singleton sets.)

Answer: Initially we have n singleton set. So any sequence of m union operations can involve at most $2m$ many elements. By our implementation, cost of a union operation is equal to the number of elements recolored during that operation. Hence total cost of m operations is at most twice the total number of recoloring that happens. Now count the total number of recoloring by element wise. Observe that each element can be recolored at most $\log n$ times. This is because since we are recoloring smaller set, an element is recolored k times means it was part of smaller set k times, and each time the size doubles. More specifically, if we consider all these k unions where that element is part of the smaller set, and let $s_1, s_2, \cdots, s_k$ be the sizes of those smaller sets. Clearly $s_i \geq 2s_{i-1}$, and so $k \leq \log n$. Thus total number of recoloring is bounded by $O(m \log n)$.

Question 4: Suppose Alice insists Bob to maintain a dynamic table (that supports both insertion and deletion) in such a way its size must always be a Fibonacci number. She insists on the following variant of the rebuilding strategy. Let F_k denote the k-th Fibonacci number. Suppose the current table size is F_k.

After an insertion, if the number of items in the table is F_{k-1}, allocate a new table of size F_{k+1}, move everything into the new table, and then free the old table. After a deletion, if the number of items in the table is F_{k-3}, we allocate a new hash table of size F_{k-1}, move everything into the new table, and then free the old table.

Use either Potential method or Accounting method to show that for any sequence of insertions and deletions, the amortized cost per operation is still $O(1)$. (If you use Potential method clearly state your potential function. If you use Accounting method clearly state your charging scheme.)

Answer: Suppose the current table size is F_k, then there must be at most F_{k-1} elements present in the table. Consider the following charging scheme for insertion: Charge \$4 for each insertion, use \$1 for insertion and remaining \$3 put in the bank. Between the creation of table of size F_{k+1} and F_{k+2} there must be at least $F_k - F_{k-1} = F_{k-2}$ elements being inserted. So the bank balance at the time of creating table of size F_{k+2} must be

$$3F_{k-2} > F_{k-3} + 2F_{k-2} = F_{k-1} + F_{k-2} = F_k.$$

So we can move all the F_k elements to the new table using the bank balance (free of cost). Hence the amortization cost of insertion is $O(1)$.

For the deletion consider the following charging scheme: Charge \$3 for each deletion, use \$1 for deletion and remaining \$2 put in the bank. Between the creation of table of size F_{k-1} and F_{k-2} there must be at least $F_{k-3} - F_{k-4} = F_{k-5}$ elements being deleted. So the bank balance at the time of creating table of size F_{k-2} must be $2F_{k-5} > F_{k-6} + F_{k-5} = F_{k-4}$. So we can move all the F_{k-4} elements to the new table using the bank balance (free of cost). Hence the amortization cost of deletion is $O(1)$.

Note, at any point the bank balance is some constant times the number of elements present in the table, and hence is non-negative.