CSCI 570 Homework 2
Spring 2023

Due Date: Feb. 09, 2023 at 11:59 P.M.

1. Consider the following functions:

```
bool function0(int x){
    if (x == 1) return true;
    if (x == 0 || (x % 2 != 0)) return false;
    if (x > 1) return function0(x/2);
}
void function1(int x){
    if (function0(x)){
        for (int i = 0; i < x ; i++)
            print i;
    } else
        print i;
}
void function2(int n){
    for (int i = 1; i <= n; i++)
        function1(i);
}
```

Compute the amortized time complexity of `function2` in terms of n. Explain your answer. Provide the tightest bound using the big-O notation.

solution:

$O(n)$. function2 has a loop that runs n times. Each iteration of the loop, it calls function1 once, with inputs 1...n.

When function1 is called, it calls function0 once. If the input is odd, function0 runs in $O(1)$ time. If the input is even, but not a power of 2, function0 will recurse until it reaches an odd number. The number of recursions is the number of times the original input can be divided by 2, which we can model as a constant C. Thus the time complexity of function0 in this case is $O(log(n)) - O(log(n/C))$, which can be written as $O(log(n)) - O(log(n)) + C = O(1)$. If the input is a power of 2,

then function0 recurses $log(n)$ times, and thus the time complexity is $O(log(n))$. So when function1 is called n times over inputs 1...n, the cost of every non-diadic input of function0 will be $O(1)$, and there will be $n - log(n)$ of such inputs, for a total cost of $O(n)$. The cost of any diadic input n will be $O(log(n))$ for the call to function0, and $O(n)$ for the printing loop, and thus the total cost of the input is $O(n)$. Over all diadic inputs, we have the cost $\sum_{n=1}^{log(n)}(n)$, which is bounded above by $O(n)$. Thus the total cost over the inputs 1...n of function1 is $O(n)$, and thus the amortized cost is $O(1)$.

Therefore, function2 called m times on inputs of size n (where n is a large power of 2 roughly equal to m), has a total cost of $O(m * n)$ which we can model as $O(m^2)$, and thus has an amortized cost of $O(m)$.

Alternate solution: First, we will calculate the amortized cost for function0, then using that we will calculate the amortized cost for function1 and finally the amortized cost for function2 would be n*amortized cost for function1.

To calculate the amortized cost for function0, we can take the following three cases:
case 1: n is even and of form 2^p.
case 2: n is even and of the form: $m * 2^p$, where m is an odd number.
case 3: n is odd.

$$T(n) = \sum(\text{case} - 1) + \sum(\text{case} - 2) + \sum(\text{case} - 3)$$

$$\sum(\text{case} - 3) = \sum_{i = 3,5,7,9\ldots.(\text{up to } \frac{n}{2} \text{ terms})} 1 = n/2$$

$$\sum(\text{case} - 1) = \sum_{i = 2,4,8,16,32\ldots.(\text{up to } \log(n) \text{ terms})} \log(i)$$
$$= \log 2 + \log 4 + \log 8 + \log 16 + \log 32 + \cdots \text{ (up to } \log n \text{ terms)}$$
$$= 1 + 2 + 3 + 4 + \cdots \text{ (up to } \log n \text{ terms)} = \frac{(\log n)\big((\log n) + 1\big)}{2}$$

$$\sum(\text{case} - 2) = \sum_{i = 6,10,12,14,18,20,22\ldots.(\text{up to } \frac{n}{2} - \log n \text{ terms})} p, \qquad (\textit{where } i = m * 2^p) \cong n$$

$$\frac{T(n)}{n} = \frac{\left(\frac{n}{2} + \frac{(\log n)\big((\log n) + 1\big)}{2} + n\right)}{n} = O(1) \qquad \textit{amortized}$$

Therefore, the function0 is of $O(1)$ amortized.

To calculate the amortized cost for function1, we have the following two cases.
Case-1: function0 returns True to function1 (happens when n is a power of two)
Case-2: function0 returns False to function1 (happens when is not a power of two)

$$T(n) = \sum(\text{case} - 1) + \sum(\text{case} - 2)$$

$$\sum(\text{case} - 1) = \sum_{i = 2,4,8,16,32\ldots.(\text{up to } \log(n)\ terms)} i$$
$$= 2 + 4 + 8 + 16 + \cdots (\text{up to } \log(n)\ terms) = 2 * (n - 1)$$

$$\sum(\text{case} - 2) = 1 * (\textit{remaining terms that are not power of } 2) = n - \log(n)$$

$$\frac{T(n)}{n} = \frac{(2 * (n - 1) + n - \log(n))}{n} = O(1) \qquad \textit{amortized}$$

Therefore, the function1 is of O(1) amortized.

Now, the amortized cost of function2 would be n*O(1) => O(n) amortized.

2. We have argued in the lecture that if the table size is doubled when it's full, then the amortized cost per insert is acceptable. Fred Hacker claims that this consumes too much space. He wants to try to increase the size with every insert by just two over the previous size. What is the amortized cost per insertion in Fred's table?

   solution:
   $O(n)$. With every insert into an already full array, you will copy all of the elements from the old array into the new array of increased size. This means an insert into an full array with n elements will be $O(n)$. Since we are only increasing by 2 each resizing, we will have to resize every other insert, or $n/2$ times. This gives us a total cost of $O(n^2)$ over n n inserts, and thus an amortized cost of $O(n)$.

3. Prove by induction that if we remove the root of a k-th order binomial tree, it results in k binomial trees of the smaller orders. You can only use the definition of B_k. Per the definition, B_k is formed by joining two B_{k-1} trees.

solution.
Base Case: B_1. If we remove the root node we get 1 empty binomial tree B_0.

Induction Hypothesis: Assume that B_{k-1} without its root node is made up of $k-1$ trees of smaller orders.

Proving k-th case: Per our definition, B_k is formed by joining two B_{k-1} trees. The joining happens at the root node of one of the two trees, making the root node of the resulting tree connected to the root node of a B_{k-1} tree as well as being part of the other tree. Therefore, when we remove the root node, we get

- One B_{k-1} tree, and
- One B_{k-1} tree without the root node.

It follows from Induction Hypothesis that the B_{k-1} tree without the root node consists of $k-1$ binomial trees. This plus the one B_{k-1} tree adds up to k binomial trees of smaller orders.

4. There are n ropes of lengths $L_1, L_2, \ldots, L_n$. Your task is to connect them into one rope. The cost to connect two ropes is equal to sum of their lengths. Design a greedy algorithm such that it minimizes the cost of connecting all the ropes. No proof is required.

   solution.
   Algorithm: Make a min-heap containing the lengths of all the ropes. While there is more than one element in the min-heap, extract the two smallest elements from the min-heap, add their length and put the added length back into the min-heap.

5. Given M sorted lists of different length, each of them contains positive integers. Let N be the total number of integers in M lists. Design an algorithm to create a list of the smallest range that includes at least one element from each list. Explain its worst-case runtime complexity. We define the list range as a difference between its maximum and minimum elements. You are not required to prove the correctness of your algorithm.

   solution.
   1. Start with a min-heap of size M and insert the first element of each list

into it. Then delete the root element (minimum element) from the heap and insert the next element from the "same" list as the deleted element. Repeat this process until any list is exhausted. To find the minimum range, maintain a variable that stores the maximum element in a heap at any point. Since the minimum element is present in the min-heap at its root, compute the range (high element – root element) and return the minimum range found at every delete operation. Because the heap is of size M , each pop/push operation takes $O(\log M)$ time. Since we pop and push at most N times, the worst case total time complexity is $O(N \log(M))$.

2. Once the minimum range is found, each list can be traversed to get the minimum element falling in that range in $O(N)$ to construct our required final list.

6. Design a data structure that has the following properties:

   - Find median takes $O(1)$ time
   - Extract-Median takes $O(\log n)$ time
   - Insert takes $O(\log n)$ time
   - Delete takes $O(\log n)$ time

   where n is the number of elements in your data structure. Describe how your data structure will work and provide algorithms for all aforementioned operations. You are not required to prove the correctness of your algorithm.

   solution:
   We use the $\lceil n/2 \rceil$ smallest elements to build a max-heap and use the remaining $\lfloor n/2 \rfloor$ elements to build a min-heap. The median will be at the root of the max-heap and hence accessible in time $O(1)$ (we assume the case of even n, median is $(n/2)^{\text{th}}$ element when elements are sorted in increasing order).

   Insert() algorithm: For a new element x,

   (a) Compare x to the current median (root of max-heap).

   (b) If $x <$ median, we insert x into the max-heap. Otherwise, we insert x into the min-heap. This takes $O(\log n)$ time in the worst case.

(c) If size(maxHeap) > size(minHeap)+1, then we call Extract-Max() on max-heap and insert the extracted value into the min-heap. This takes $O(\log n)$ time in the worst case.

(d) Also, if size(minHeap) >size(maxHeap), we call Extract-Min() on min-heap and insert the extracted value in to the max-heap. This takes $O(\log n)$ time in the worst case.

Extract-Median() algorithm: Run ExtractMax() on max-heap. If after extraction, size(maxHeap) < size(minHeap) then execute Extract-Min() on the min-heap and insert the extracted value into the max-heap. Again worst case time is $O(\log n)$.

Delete(): We will store the memory locations of all the elements while constructing the heaps and inserting new elements in a hash table/map. Since we will have direct access to the location of the elements, we can go to the element to be deleted, and delete it in O(logn) time. We will have similar condition as mentioned in the insert opeartin to maintain the size of each heap.

7. In a greenhouse, several plants are planted in a row. You can imagine this row as a long line segment. Your task is to install lamps at several places in this row so that each plant receives 'sufficient' light from at least one lamp. A plant receives 'sufficient' if it is within 4 meters of one of the lamps. Design an algorithm that achieve this goal and uses as few numbers of lamps as possible. Prove the correctness of your algorithm.

solution.
Algorithm: Find the first plant from the left and go four meters to its right and place a lamp there. Eliminate all plants covered by that lamp and repeat the process until all plants are covered.

Proof: Use induction to show that our lamps are always to the right of (or not to the left of) the corresponding lamps in any optimal solution.Using this fact, we can then easily show that our solution is optimal (using proof by contradiction). 1- Our lamps are always to the right of

(or not to the left of) the corresponding lamps in any optimal solution:

Base case: we place our first lamp as far to the right of the leftmost plant as possible. If the lamp in the optimal solution is to its right then the first plant on the left will not receive enough light.

Inductive step: Assume our kth lamp is to the right of the kth lamp in the optimal solution. We can now show that our $k+1$st lamp is also to the right of the $k+1$st lamp in the optimal solution. To prove this we look at the leftmost plant to the right of our kth lamp which does not receive light by our kth lamp . We call this plant P. If P is not covered by our kth lamp, then it cannot be covered by the kth lamp in the optimal solution since our lamp is to the right of it. Our $k+1$st lamp is placed 4 meters to the right of P. If the $k+1$st lamp in the optimal solution is further to the right of our $k+1$st lamp, then P is not going to receive light by neither the Kth lamp nor the $k+1$st lamp in the optimal solution so the $k+1$st lamp in our solution must also be to the right of the $k+1$st lamp in the optimal solution.

2- Now assume that our solution required n lamps and the optimal solution requires fewer lamps. We now look at our last lamp. The reason we needed this lamp in our solution was that there was a plant to the right of our $n-1$st lamp that did not receive light by it. We call this plant P. If P is not covered by our $n-1$st lamp, then P will not be covered by the $n-1$st lamp in the optimal solution either (since our lamps are always to the right of the corresponding lamps in the optimal solution). So, the optimal solution will also need another lamp to cover P.

8. We are back again at the same greenhouse from problem-1, but this time our task is to water the plants. Each plant in the greenhouse has some minimum amount of water (in L) per day to stay alive. Suppose there are n plants in the greenhouse and let the minimum amount of water (in L) they need per day be l1, l2, l3,. . . ln. You generally order n water bottles (1 bottle for each plant) of max(l1, l2, l3,. . . .ln) capacity per day to water the plants, but due to some logistics issue on a bad day, you received n bottles of different capacities (in L). Suppose c1, c2,c3. . . cn be the capacities of the water bottles, and you are required to use one bottle completely to water one plant. In other words, you will allocate one bottle per plant, and use the entire water present (even if it is more than the minimum amount of water required for that plant) in that bottle to water a particular plant. You cannot use more than one bottle (or partial amount of water) to water a single plant (You need to use exactly one bottle per plant). Suggest an algorithm to determine whether it is possible to come up with an arrangement such that every plant receives more than or equal to its minimum water requirement. Prove the correctness of your algorithm.

solution.

Algorithm:

1) Sort the capacities of the bottles in ascending order
2) Sort the minimum water requirements in ascending order
3) Possible if no li $>$ ci, otherwise not possible.

Proof:

Base case: There is only one plant in the greenhouse having minimum water requirement of l and we have only one bottle of c capacity. Clearly, our algorithm will compare c with l, it will return True if c$>$l, and false otherwise, which is trivially simple and always correct.

Inductive Hypothesis: We assume that our algorithm produces correct judgement for k plants and k bottles:

Our algorithm has ordered water requirements and bottle capacities in

non-decreasing order
Then we assume:
If ci > li for i = 1 to k => possible, otherwise not possible.

Induction step: Using our inductive hypothesis, We will prove for $k+1$ plants and $k+1$ bottles:

Case-1: Let's say that our algorithm finds that there in fact exists an arrangement where all the fist k plants are satisfied (Again, our algorithm arranges the plants in ascending order of water requirements), which we assumed that our algorithm will always make a correct judgment in the induction hypothesis. Now, we have $k+1$th plant w/ requirement l(k+1) and capacity c(k+1). Here our algorithm would compare c(k+1) w/ l(k+1). It will decide: c(k+1) > l(k+1), then arrangement possible, otherwise not possible. We can argue that this is in fact true because if c(k+1) cannot water the plant (k+1) completely, then none of the c(i), i = 1 to k bottles would be able to water the (k+1)th plant, because by the nature of our algorithm, all of them have lesser water capacities compared to c(k+1). On the other hand, if c(k+1) is able to water the (k+1)th plant, then the algorithm will return true, which is also correct because all the first k plants (our hypothesis) have already been watered and now the (k+1)th plant also got watered.

Case-2: Let's say that our algorithm finds that the arrangement is not possible because kth plant's water requirements were not satisfied, and we have k+1 plants and k+1 bottles . Now, if bottle k cannot water the kth plant, then none of the 1 to k-1 bottles would be able to water the kth plant because they have lesser water capacity compared to bottle k. If we use (k+1)th bottle to water the plant k, then (k+1)th plant would be left out, since kth bottle cannot bottle kth plant, it will also be not able to water the (k+1)th plant as water requirement for (k+1)th is higher than kth plant or C(k+1) > C(k). Therefor, one plant will be left out (w/ out getting satisfied), and hence our algorithm will return the correct judgement.

Alternate Proof:
Suppose there is an optimal solution where all plants receive their minimum water requirement. We will order plants in that solution in ascending order of their minimum water requirements (i.e. lj > li if j>i.) Inversion can be defined as follows: plant i has a larger water requirement than plant j but appears before plant j. It is obvious that by removing this inversion the solution will still remain optimal (because ci > cj so plant j will be fulfilled, and since lj > li it must be that cj > li so giving plant j's bottle to plant i will also fulfil plant i.) If there is an optimal solution that has inversions in it, we can remove all inversions one by one resulting in an optimal solution with no inversions. This solution is the same as our solution since our solution has no inversions. So, our solution is also optimal.

9. Let's assume that you are worker at a bottled water company named Trojan Waters which supplied water bottles to the greenhouse in problem-2. At the facility, there is a water filter (completely filled with water) which has a capacity of W (in L), and there are n different empty bottles of capacities p1, p2,. . . .pn. Your job as a loyal worker is to design a greedy algorithm, which, given W and p1,p2,. . . ,pn, determine the fewest number of bottles needed to store the entire water W . Prove that your algorithm is correct.

solution.
Algorithm: Sort the n bottles in decreasing order of their capacity. Pick the 1st bottle, which has largest capacity, and fill it with water. If there is still water left in the water filter, fill the next bottle with water. Continue filling bottles until there are no more water left in the water filter.

Proof:

1.) The prove is trivially simple. We will use our same old greedy approach, that the greedy stays ahead of a corresponding optimal solution. Here staying ahead would mean that, at every step, our greedy algorithm would add more water in the selected bottles compared to an optimal so-

lution.

Base Case: The very first bottle selected by our greedy solution would be the bottle of the largest capacity, which can store the highest amount of water. The optimal solution can not select any bottle of higher capacity because there isn't any.

Inductive hypothesis: Let us assume that the first k bottles selected by our greedy solution would store more water in total compared to the first k bottles selected by an optimal solution.

Induction step: Here we will argue that first $k+1$ bottles selected by our greedy solution will contain more water compared to $k+1$ bottles selected by the corresponding optimal solution. We will denote the capacity of $k+1$st bottle selected by the greedy solution as $G(k+1)$, and the capacity of $k+1$st bottle selected by the optimal solution as $O(k+1)$.

Case-1: $G(k+1) >= O(k+1)$.
Since sum of capacities of first k bottles selected by our greedy solution is greater than sum of capacities of first k bottles selected by our optimal solution, the same will hold true for k+1 bottles:

sum-greedy(k) >= sum-optimal(k)
Since $G(k+1) >= O(k+1)$
sum-greedy(k) + $G(k+1)$ >= sum-optimal(k) + $O(k+1)$
sum-greedy(k+1) >= sum-optimal(k+1)

Case-2: $G(k+1) < O(k+1)$.

When case-2 happens, we make the following observations:
1.) By the design of our greedy algorithm, at every step we are picking the bottle of highest capacity available, therefore our greedy algorithm would have already picked up the bottle of capacity O(k+1) in some previous jth step, where j = 1 to k.

2.) Also, G(k+1) is the smallest bottle amongst all the bottles picked till now by our greedy solution.

3.) For case-2 to ever occur, the optimal solution must pick at least one bottle of equal or lesser capacity compared to G(k+1) in some previous step.

From 1,2 and 3, We can say that when at k+1st step, both optimal and greedy solution would have at least one bottle in common of capacity O(k+1). Also, there is at least one bottle in the optimal solution that has a lesser capacity compared to G(k+1). Note: G(k+1) is the bottle of minimum capacity amongst k+1 bottles in the greedy solution.

Since, the greedy algorithm contains all the bottles of larger capacity compared to G(k+1) (w/ G(k+1) included amongst k+1 bottles), and the optimal solution contains at least one bottle of equal or lesser capacity compared to G(k+1), we can conclude that sum of k+1 bottle capacities of the greedy solution will be greater than or equal to the sum of k+1 capacities of the optimal solution.

Now, using the above hypothesis that we just proved (i.e, our greedy algorithm would add more water in the selected bottles compared to an optimal solution), we can prove that our greedy solution will have same number of bottles (if not less) compared the optimal solution.

Let's assume that optimal solution would need fewer number of bottles compared to our greedy solution. Let's say that at some (n-1)st step, our greedy algorithm adds n-1 bottles and some W' amount of water is still left in the tank because of which the greedy solution had to add 1 more bottle (n bottles in total). But since our greedy algorithm adds more or equal amount of water at every step compared to a corresponding optimal solution, it would imply that at n-1st step of the optimal solution, the water left in the tank will be more or equal to W', therefore, the optimal

solution can not get away with fewer number of bottles. It has to also add n (or more in total). Therefore, we arrive at a contradiction and our assumption was wrong. So, using prove by contradiction, we showed that our greedy solution will have same number of bottles (if not less) compared the optimal solution .

Alternate apporach/proof: we will show that there is an optimal solution that includes the 1st greedy choice made by our algorithm. Let k be the fewest number of bottles needed to store the water in the water filter and let S be some optimal solution. Denote the 1st bottle chosen by our algorithm by p'. If S contains p', then we have shown our bottle is in some optimal solution. Otherwise, suppose p' is not contained in S. All bottles in S are smaller in capacity than p' (since p' is the largest bottle) and we can remove any of the bottles in S and empty its water into p', creating a new set of bottles S' = S – p Union p' that also contains k bottles (the same number of bottles as in S'). Thus S' is an optimal solution that includes p' and we have shown there is always an optimal solution that includes the 1st greedy choice. Because we have shown there always exists an optimal solution that contains p', the problem is reduced to finding an optimal solution to the subproblem of finding k-1 bottles in (p1,p2,. . . ,pn)- (p')to hold W-p' amount of water. The subproblem is of the same form as the original problem, so that by induction on k we can show that making the greedy choice at every step produces an optimal solution.

10. **Online Questions.** Please go to DEN (https://courses.uscden.net/) and take the online portion of your assignment.