Aaron Dardik

CS567 Problem Set 2

1) We have three functions: For given an input integer *x,* if *x* is odd, then . Otherwise  For with input *x,* if Now we will examine with input *n.* For every integer if then the total cost increases by *x.* If then the total cost increases by 1. To calculate the total cost, we see that there are numbers of the form and these numbers are the sequence Adding up these numbers we get For the remaining numbers in the sequence, of which there are the total cost increases by 1. So, the total cost equals As the amortized cost of running over a sequence is the same for any sequence, we see that the amortized cost of

2) According to Fred Hacker’s method of increasing the table size by 2 every time an insert is made we see that the cost per insert is 3: 1 “token” for inserting the item and 2 “tokens” for adding an extra 2 spaces to the array. Therefore we see that

3) First we will show the base case on By removing the root (and only node) on we get the empty set, By removing the root on we are left with one node and no edges remaining and thus have We will now move to the inductive step. Suppose that removing the root of a (*k-1)*th order binomial tree results in *k-1* binomial trees of smaller order for all integers 1, 2,…,*k-1.* Now, consider by definition where the “-“ means joined together with an edge. Let the two copies of be denoted by Assume without loss of generality that the root of the tree lies in By removing the root we are left with I) as it has now been separated from and II) everything that results from removing the root of which is a binomial tree of order *k-1.* This results in . By combining these with we have which shows by induction that for any binomial tree of order *k,* removing its root results in *k* binomial trees of smaller order.

4) Given the *n* ropes of lengths we begin by sorting them such that, without loss of generality, We start by joining rope 1 and rope 2, and, denoting the new rope, we now have *n-*1 ropes. Place the rope in place in the sequence of remaining ropes, and by relabeling we now have *n*-1 ropes in order of length, Repeat this process by adjoining the ropes that now have the smallest length and putting the adjoined rope in order in a new list of length *n*-2. Repeating this process until only one rope is left connects all the ropes with minimal cost.

5) Given M sorted lists, we will denote them as where and more generally that Order the lists such that We will now start by creating a “dummy list” of Now, starting with replace with where is the largest element of that is less than or equal to Repeat this process for all *I*, replacing with where is the largest element of its list that is less than or equal to This gives us the list with smallest range containing as its maximum. No list with smaller range can have in its “interior” as if there were a different last element in the list, the only way to decrease the range is to increase the minimum, but by construction of the list, this would move to the minimum position. Now we repeat the process starting with a new “dummy list” of Continue repeating for the following “dummy lists” of To find the biggest element in that is less than or equal to takes time by binary search. Doing this over each list and repeating the process for each dummy list gives a total time of

6) To construct the median heap we will begin by constructing a binary min-heap and a binary max-heap. Every element in the binary min-heap is greater than or equal to the median and every element in the binary max-heap is less than or equal to the median. Note that finding the median is as simple as taking the root of the larger binary heap. This can be done in O(1) time. To insert an element into the median heap we first check if the element is larger than the median. If so, it is inserted into the min-heap using the binary heap insert procedure, which is If it is smaller than the median we perform the same procedure, but on the binary max-heap. Extracting the median or deleting a node can be performed by running the binary tree extract algorithm on the appropriate heap and possibly adjust the heap sizes by at most 1 element. This takes )+ time.

7) The algorithm to place the lamps is as follows: proceed to the first tree, then walk an additional 4 meters and place the first light here. Then walk to the first tree that is more than 4 meters away, walk an additional 4 meters and place a light here. Continue this process. Once there is no tree that is more than 4 meters away the process terminates. We will prove the correctness of the algorithm by induction. If there is only one tree, clearly only one light is needed. If there are two trees, place the first light according to this process, and if the second tree is more than 4 meters from the first we place a second light. Suppose this algorithm works for a sequence of *k* trees. Add a (*k*+1)st plant to the sequence. If the plant lies within 4 meters of an already present light source and suppose a different placement of lights were optimal. This would imply the solution for *k* plants is non-optimal, which is a contradiction. If the plant is more than 4 meters away from any light source, and without loss of generality it is the (*k*+1)st plant in the sequence, if adding the (*k*+1)st light according to the algorithm were non-optimal, then this would also imply a non-optimality in the sequence of *k* plants, which is also a contradiction. Thus, by induction, the algorithm is optimal.

8) To determine whether each plant can receive greater than or equal to its water requirement we begin as follows. We first “sort” the plants so we have an order such that if We then sort the water bottles so that The plants and bottles can fill the requirement, if . This is equivalent to finding a function *f,* such that After sorting the plants we see that this function *f* must lie above the line *y=x.* Thus the plants and bottles fulfill their requirements if, after sorting, we have . To prove correctness we will use induction. If there is one plant with requirement then it gets its requirement if and only if Suppose that this algorithm works to determine if *n*-1 plants can meet the watering requirements. Now consider *n* plants and we wish to know if they can be watered. By the inductive hypothesis the first *n*-1 plants can be watered if and only if . Similarly, the last plant can be watered if If the first *n*-1 plants and the last plant can be watered then all *n* plants can be watered this way. If they can’t it would be because either the sequence of plants numbered 1,2,…,*n*-1 can’t be watered or the *n*th plant couldn’t be watered. But this would contradict the induction hypothesis. Therefore, this algorithm determines whether the plants can be watered.

9) The greedy algorithm that will store the water volume W in the fewest bottles proceeds as follows: First, pour from W into the bottle has volume then, if water remains in the filter, pour into the bottle which has volume . Repeat this process until no water remains in the filter. To prove this algorithm is correct, if there is only one bottle, then pouring the water into that bottle is the only, and therefore optimal, solution. Suppose this solution is optimal for *n* bottles. Given *n*+1 bottles, if the (*n*+1)st bottle has volume less than where the bottles are used to store the water from the filter, then we clearly have used the optimal solution by the inductive hypothesis. If this is not the case, then without loss of generality let have maximal volume amongst the bottles. By pouring from W a volume equal to into we have a remaining volume of to be distributed amongst the remaining bottles, and this can be done with a number of bottles less than or equal to the optimal solution when we had *n* bottles. Thus by induction in this case too is the greedy algorithm optimal.