dnabar@usc.edu

Deven Parag Nabar
USC ID: 7229446568

CSCI 570 Homework 2

1. SOLUTION:

- Here, observing the given code we can see that function2 is calling function1 n times and function1 is calling function0 every single time.
- **Function0** will either return the value true or false depending on the fact of whether variable x is a power of 2. In order to do this, it will call itself after dividing the variable x by 2
- Let us assume that the cost of comparison is 1 and cost of calling funtion0 is also 1. For example, in the case of input 4, function0 is called 2 times and comparison is done once so the total cost is 3.
- We can make a table for the cost of function0

x in function0(x)	**Cost**	**Returns (T/F)**
0	1	F
1	1	T
2	2	T
3	1	F
4	3	T
5	1	F
6	2	F
7	1	F
8	4	T
9	1	F

- We can split these costs as follows for the sake of simplicity
  - $1 \to 1$
  - $2 \to 1 + 1$
  - $3 \to 1$
  - $4 \to 1 + 1 + 1$
  - $5 \to 1$
  - $6 \to 1 + 1$
  - $7 \to 1$
  - $8 \to 1 + 1 + 1 + 1$
  - $9 \to 1$
- We can place these costs in different columns.
- We can observe that when we can split the costs for N inputs of n natural numbers 1 cost comes N times, the next 1 cost comes N/2 times, the next 1 cost comes N/4 and the next cost comes N/8 times and so on.
- For function0:

$$T(n) \le N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \cdots + 1$$

dnabar@usc.edu

Deven Parag Nabar
USC ID: 7229446568

$$T(n) \le N\left(1+\frac{1}{2}+\frac{1}{4}+\cdots+\frac{1}{N}\right)$$

$$T(n) \le N\left(\frac{1}{1-\frac{1}{2}}\right)$$

$$T(n) = 2N$$

- Now we can observe that **function1** checks if function0 returns T or F
- If function0 returns T, we print all the numbers from 0 to x
- If function0 returns F, we print x only once
- Function0 returns F, for all values other than powers of 2
    - 1 → 1(function0 returns T) +1(print)
    - 2→ 2(function0 returns T) + 2(print)
    - 3→ 1(fuction0 returns F) + 1(print)
    - …
- $T(x) = Cost\ of\ powers\ of\ 2 + Cost\ of\ nonpowers\ of\ 2$

$$= \left(1+2+4+\cdots+2^{\log x}\right) + 2*(x-\log x)$$

$$= 2^{\log x+1} - 1 + 2x - 2\log x$$

- Now adding the function0 cost (2x) we got in the beginning we get,

$$T(x) = 2^{\log x+1} - 1 + 2x - 2\log x + 2x$$

$$T(x) = 2^{\log x+1} - 1 + 4x - 2\log x = 6x - 2\log x - 1$$

- Now **in case of function2**, function1 is run from 1 to n

$$T(n) = \sum_{x=0}^{n} 6x - 2\log x - 1 = \frac{6n(n+1)}{2} - 2\log n! - n$$

$$\le 3n^2 + 3n - 2nlog\ n - n$$

$$T(n) \le 3n^2 + 2n - 2nlog\ n$$

- Amortized cost = $\lim_{n\to\infty} \frac{3n^2+2n-2nlog\ n}{n}$ = $O(n)$
- **Amortized time complexity of function2 in terms of n is in linear time which is $\boldsymbol{O(n)}$**

dnabar@usc.edu

Deven Parag Nabar
USC ID: 7229446568

2. SOLUTION:

- By following Fred Hacker's method, we increase the size by two just over the previous size.

Insert i^{th} element	**Old size**	**New size**	**No. of Copies**
1	1	-	-
2	1	3	1
3	3	-	-
4	3	5	3
5	5	-	-
6	5	7	5
7	7	-	-
8	7	9	7
9	9	-	-
10	9	11	9

- We can see that according to the table, for 10 insert operations we need 1+3+5+7+9 =25 copy operations and 10 inserts operations.
- We can generalize this for n insert operations and say that total cost of n operations is total cost of insert operations and total cost of copy operations.
- $T(n) = n + (1+3+5+7+\cdots+n) = n + \left(\frac{n+1}{4}\right).(n+1)$

  $\ldots Using\ Summation\ of Arithmetic\ progression\ formula\ for\ series\ of\ odd\ numbers$

$$= n + \frac{(n+1)^2}{4} = \frac{n^2+6n+1}{4}$$

- Now the amortized cost is

$$\lim_{n\to\infty} \frac{T(n)}{n} = \frac{\frac{n^2+6n+1}{4}}{n} = n$$

- **Here we can see that amortized cost is in linear time i.e., $\boldsymbol{O(n)}$**

.

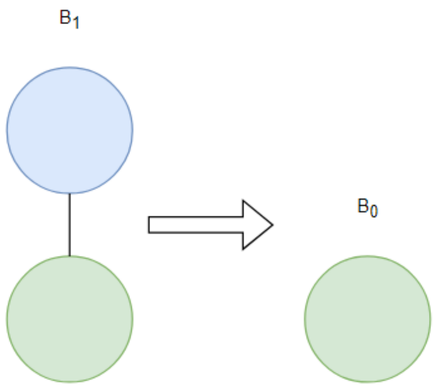
dnabar@usc.edu

Deven Parag Nabar
USC ID: 7229446568

3. **PROOF: by induction**

Base Case:

- Consider a binomial tree B_1 of order 1. This tree is made of 2 nodes
- If we remove the root node of this binomial tree, we are left with only one node and hence a binomial tree of order 0.



Induction Hypothesis:

- Assume that the given statement holds for a binomial tree B_k for $k > 0$

Induction Step:

- We need to prove that the statement still holds true for B_{k+1} as well
- Now we remove the root node of this binomial tree B_{k+1} and end up with one binomial tree of order k and another binomial tree B_k with the root node removed for this tree
- Since we have assumed in the Induction Hypothesis that the given statement holds for binomial tree of order k, this results in formation of k binomial trees of smaller order and one binomial tree of order k.
- Thus, we are left with $(k + 1)$ binomial trees of smaller order
- **Hence, proved by induction**

dnabar@usc.edu

Deven Parag Nabar
USC ID: 7229446568

4. SOLUTION:

- We are given n ropes of lengths $L_1, L_2, L_3, \ldots\ L_n$. The cost of connecting two ropes is equal to the sum of their lengths. We can connect them into a single rope using greedy algorithm by following these steps:
    a. First, we initialize a variable totalCost = 0
    b. Now we perform the heapify operation on all the lengths of the rope to build a min-heap.
    c. We then perform deleteMin operation twice on the min-heap and add the values of these elements in the totalCost variable.
    d. We add these values together and add their sum in the min-heap again by using the insert operation.
    e. We repeat steps c and d until we are left with a single item in the end.
- We know that the time complexity for heapify is $O(n)$, time complexity for insert operation is $O(\log n)$ and time complexity for deleteMin operation is $O(\log n)$
- Hence the **time complexity for this algorithm is $\boldsymbol{O(n \log n)}$** because we have $O\big(n + n(2\log n + \log n)\big)$ as heapify takes place once, deleteMin takes place $2n$ times and insert operation takes place n times.

dnabar@usc.edu
Deven Parag Nabar
USC ID: 7229446568

5. SOLUTION:

- We are given M sorted lists of different lengths, each containing positive integers. N being the total number of integers in M lists.

a. We can now create a min heap of size M.

b. We now insert first elements from all the M sorted lists in the min heap

c. We now delete the root node of the min heap and add the next element from the same list which contained the minimum element

d. We repeat the above $step\ c$ until one of the M lists is empty and we stop here.

- As we are deleting and adding number in the min heap, we have to keep a track of min and max elements in order to calculate the range. We update this range whenever we come across a smaller value for it.
- We can maintain a variable called Max to store the maximum value encountered which gets updated every time we add an element in the heap.
- **Time Complexity:**
  The size of the heap in our case is M and any operations like deleteMin and insert taking place will cost $O(\log M)$ complexity for the heap. Since in the worst case we have to perform these operations N times (the total number of elements is N), the worst-case runtime complexity for the algorithm comes out to be $\boldsymbol{O(N\log M)}$**.**

dnabar@usc.edu

Deven Parag Nabar
USC ID: 7229446568

6. **SOLUTION:**
   - We essentially create an empty min-heap for the upper half of the elements and an empty max-heap for the lower half of the elements.
   - We start adding our first element in any one of the heaps.
   - Let us say that the first element was placed in the max heap, if the next element is smaller than this element it is placed in the max heap or else it goes in the min heap (and vice versa).
   - We also make sure that the difference between these two heaps is never greater than 1. If the difference in the number of elements in the two heaps is greater than 1, we delete the root node from the bigger heap and insert it into the smaller size heap to rebalance it.

   a. **Find median:**
      - If the size of one of the heaps is larger than the other heap after inserting n numbers in the data structure described above, we peek at the root node of the larger heap which would give us the median.
      - If the size of the min heap and max heap are the same, we peek at the roots of both the heaps and find the average value of the two.
      - This will be done in $O(1)$ time as peeking takes constant time.

   b. **Extract-median:**
      - After finding the median we can run deleteMin or deleteMax operations on the min heap or the max heap depending on the size difference.
      - We will need to run deleteMin once if min heap is larger in size than max heap and vice versa.
      - If their sizes are equal, we need to run deleteMin on min heap and deleteMax on max heap and find the average.
      - Even in the worst-case scenario where we run both of these functions the time complexity is still $O(2\log n) = O(\log n)$

   c. **Insert:**
      - We compare the element to be inserted with the root nodes of the min heap and max heap.
      - If the element is greater than the root node of the max heap we insert it into the min heap and if the element is smaller than the root node of min heap we place it in the max heap.
      - If after insertion, the difference in the size of these heaps is greater than 1, we delete the root node of the bigger size heap using deleteMin or deleteMax operations and insert it in the smaller size heap.
      - Since insert operation for a heap has the time complexity of $O(\log n)$ and time complexity for the worst case where delete operation is also taking place is also $O(\log n)$, we can say that time complexity for insert operation for this data structure is also $O(\log n)$

dnabar@usc.edu

Deven Parag Nabar
USC ID: 7229446568

d. Delete:

- For deleting an element, we can first keep track of all the indices of the elements in this data structure (min heap + max heap) using a hash table. This will let us find element in $O(1)$ which is constant time.
- We could then delete the element in question and percolate the child nodes of the deleted element in $O(\log n\ )$ time.

dnabar@usc.edu

Deven Parag Nabar
USC ID: 7229446568

7. **SOLUTION:**
    - We can solve this problem using greedy algorithm by performing the following operations:
        a. Sort the plants in the order from left to right as they appear in the greenhouse and store them in an array as plants $p_{1,}p_{2,}p_{3,}p_{4,} \dots p_n$
        b. Go to the first plant and then put a lamp 4 metres on the right of the first plant.
        c. Then go to the next plant in the sorted order that is not covered by the lamp and place a lamp 4 metres to the right of this plant
        d. Repeat these steps until all plants are covered.

Proof of correctness:

- Let the Algorithm we used produce the lamps in order $l_1, l_2, l_3, l_4, \dots l_k$
- And let the Optimal solution produce the lamps in order $t_1, t_2, t_3, t_4, \dots t_m$
- We now perform induction on lamps:
    - **Base Case:**
      Let there be one plant, in which case we can put a lamp 4 metres to the right of the plant and it would be optimal.
    - **Inductive Hypothesis:**
      Assume that this is true for $(c-1)$ plants.
    - **Inductive Step:**
      We need to now prove it for c plants.
      We know that by Inductive hypothesis,
      $$ALG: l_1, l_2, l_3, l_4, \dots l_{c-1}$$
      Case 1: We place our lamp to the left of the optimal position for the lamp. Since optimal must be at most 4 metres away from the plant, the difference in between lamp placed by algorithm and the plant would be smaller than 4 metres. But according to our definition we are placing the lamp at 4 metres from the plant and hence this case is not valid. This wouldn't happen in greedy approach.
      Case 2: We place our lamp to the right of the optimal position for the lamp. In this case (optimal solution) lamp placement is less than 4 metres to the right of $(c-1)^{th}$ plant. Our solution yields a bigger distance than the optimal solution and so it wouldn't be optimal.
      Hence, our solution does not perform worse than optimal solution in both the cases and is as good as the optimal solution.
      Hence proved.

dnabar@usc.edu

Deven Parag Nabar
USC ID: 7229446568

8. **SOLUTION:**

- We can determine whether it is possible to come up with an arrangement to water every single plant so that it receives more than or equal to its minimum water requirement by following these steps:
  a. Sort all the water bottles c1, c2, c3, …, cn in ascending order of their capacities
  b. Sort all the plants in ascending orders of their minimum water requirements (l1, l2, l3, l4, …, ln)
  c. Now iteratively compare i^{th}index water bottle capacity c_i with l_i
  d. If we encounter a c_i < l_i, we can say that it is not possible to water all plants
  e. If we reach the very end of the loop we are running in step c, without encountering the condition in step d, we can say all plants will receive their water requirements.

Proof of correctness:

- **Base Case:**
  Let there be 1 plant and 1 water bottle. If the capacity of the water bottle is more than the plant's water requirement, it would be possible to water the plant. If the capacity of the water bottle is less than that required by the plant, it wouldn't be possible to water the plant.
- **Induction Hypothesis:**
  Assuming that plants with capacities l1, l2, l3, l4, …, lk can be watered by bottles of capacities c1, c2, c3, c4, ..., ck for k>0 and k is an integer
- **Induction Step:**
  We need to check for the (k+1)th plant and water bottle
  Case 1:
  If c_{k+1} is greater than l_{k+1}, we will be able to water the plant with this water bottle.
  Case 2:
  If c_{k+1} is smaller than l_{k+1}, we will not be able to water the plant with this water bottle. In this case we find a water bottle with larger capacity that can be used to water the plant. However this isn't possible as all the remaining plants will require bottles with higher capacities as we have already sorted them in ascending order and we only have n water bottles for n plants.

Hence, we proved correctness by Induction

dnabar@usc.edu

Deven Parag Nabar
USC ID: 7229446568

9. SOLUTION:

- We can perform the following steps by following the greedy approach:
  a. Set a variable filterCapacity = W (capacity of the water filter)
  b. Sort the empty water bottles in decreasing order of their capacities and store them in an array.
  c. Fill the water bottle with the largest capacity which is the first element in the array and increment the index and fill the next water bottle.
  d. Repeat step c till the water filter is empty i.e., filterCapacity = 0
- This will result in the fewest number of water bottles needed to store the entire water
- (Assuming that partially filled water bottles are allowed as mentioned in piazza.)

Proof of correctness:

- **Base Case:**
  Let there be one water bottle with some capacity x litres. In this case we are withdrawing the maximum amount of water possible and hence using the fewest water bottles
- **Induction Hypothesis:**
  Assuming that the algorithm holds true for k water bottles where k>0
- **Induction Step:**
    - We need to now prove that this holds for k+1 water bottle.
    - If the algorithm worked for k water bottles, this means that (k+1)th water bottle is the largest possible water bottle available and would withdraw the largest possible quantity of water at this step.
- Let there be an optimal solution able to do the job in some U number of bottles and our algorithm be able to do it in V number of bottles where (U<V)
- But we have shown that given any number of water bottles we will be pulling maximum amount of water from the water filter.
- Hence U = V and so our algorithm performs as well as the optimal algorithm.
- **Hence, we have proved correctness**.