CSCI 570 Analysis of Algorithm Homework - 2

Name: Aagam Shah
USC ID: 5420023430

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.

Ans)
Let's consider the amortized cost for function 0 and function 1 to find out the correct amortized cost analysis for function 2.

Function 0
Consider the input given to function 0 for a sequence of n input ranging for 1 to 8 is :

N	1	2	3	4	5	6	7	8	9
Function 0	1	$\log 2$	1	$\log 4$	1	2	1	$\log 8$	1

Here we can have 3 cases possible:
1. The Number is a power of 2
2. The Number which are odd
3. Rest of the remaining numbers

Case 1:

For powers of 2,function-0 will return true and the cost of function-0 in this case is:

$$cost\ when\ n\ \in 2^i\ i.e., 2,4\ and\ 8$$

$$\log 2 + \log 4 + \log 8 = 1 + 2 + 3 = 6$$

In general, case 1 will run $\log n$ times. Hence, the total cost in general will be:

$$\sum_{i=1}^{\log n} cost\ of\ n\ when\ n\ \in 2^i = \sum_{i=1}^{\log n} \log 2^i = \log 2 + \log 4 + \log 8 + \cdots + \log 2^{\log n}$$

$= 1 + 2 + 3 + \cdots + \log n$

Sum of n natural numbers= $\frac{n(n+1)}{2} = \frac{\log n(\log n+1)}{2}$

Case 2:

For Odd numbers the function will return false thus we will have the cost as 1. As we know we have n/2 odd terms we have generalized solution as

$$\sum_{i=1}^{\frac{n}{2}} 1 = \frac{n}{2}$$

Case 3:

For non-powers of 2, function0 will run a variable number of times.
For example, for x=6, it will run 2 times, for x = 10, it will run 2 times and so on. Hence, assume the cost per operation i for this case to be c where c<log(i). In general, case 3 will run:
(total number of operations-number of operations with 2^i power-number of operations having odd number)= $\left[n - \frac{n}{2} - \log n\right] = \left[\frac{n}{2} - \log n\right]$ times

the total cost, will be of the order = K$\left[\frac{n}{2} - \log n\right]$

Calculating the total cost for function0

Therefore, the total cost of $function0$ for a sequence of n operations is:

$$Total\ cost = cost\ of\ case\ 1\ +\ cost\ of\ case\ 2\ +\ cost\ of\ case\ 3$$
$$= \frac{\log n\,(\log n + 1)}{2} + \frac{n}{2} + K\left[\frac{n}{2} - \log n\right]$$

Since the dominating term in the above summation is of order total cost will be less than $O(n)$.

Therefore, amortized cost of $function0$ is Total Cost / No of Operation

Amortized Cost = Total Cost / No of Operation

= n / n

= O(1)

Function1:

Consider the implementation for Function 1. Here we know that function 1 will run based on the if condition of the Function0 and then for those we will run for n times i.e. only when we have powers of 2. where the if condition will come true. And for the rest of the cases, it will run 1 time. Consider an example down below:

N	1	2	3	4	5	6	7	8	9
Function 1	1	2	1	4	1	1	1	8	1

Cost of Function 1 in general will be given as

$$= cost\ of\ if\ block + cost\ of\ running\ the\ else\ block$$
$$= cost\ of\ operation\ when\ we\ have\ n\ \in 2^i + Cost\ of\ else\ block - remaining\ operations$$
$$= \sum_{i=0}^{\log n} 2^i + \sum_{i=1}^{\text{n}-(1+\log n)} 1$$
$$= \left(1 + 2 + 2^2 + 2^3 + \cdots + 2^{\log n}\right) + \left(n - (1 + \log n)\right)$$

Using the Geometric Progression infinite series equation we get

$$S_n = a\left[\frac{r^n - 1}{r - 1}\right]$$

where,

a = first term , n = number of terms , r = common ratio

$$= 1\left[\frac{2^{1+\log n} - 1}{2 - 1}\right] + (n - (1 + \log n))$$

$$Applying\ Log\ Property : a^{\log_a b} = b$$

$$= 2n - 1 + n - 1 - \log n$$
$$= 3n - \log n - 2$$

Here n will be the dominating function thus we take total cost of order O(n)

Therefore, amortized cost of $function1$ is Total Cost / No of Operation

Amortized Cost = Total Cost / No of Operation

= n / n

$= O(1)$

Function2:

Here the function 2, it will run N number of times the function 1. Thus we can see that the total cost of function 2 = n^2

$$\text{Amortized Cost} = \text{Total Cost} / \text{No of Operation}$$
$$= n^2 / \text{n}$$
$$= O(\text{n})$$

Amortized Cost of Function2 is $O(\text{n})$

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?

Ans)

Let's consider the Fred Hacker table and modify it to increase the size of the array by 2 every time it gets full.

Inserted Elements	Old Size	New Size	No of copy
1	1	-	-
2	1	3	1
3	3	-	-
4	3	5	3
5	5	-	-
6	5	7	5
7	7	-	-
8	7	9	7

Here as we can see that we have assumed that we inserted elements from 1 to 8 and then we calculated the number of times we copied the element from the old array to the new array. Total cost here can be given as the (Cost to Insert Elements) + (Cost to Copy the elements from Old Array to New Array)

Cost to Insert Elements = No of Elements * 1 = 8 * 1 = 8

Cost to Copy from Old Array to New Array = 1 + 3 + 5 + 7 = 16

$$\text{Amortized Cost} = \frac{Total\ Cost}{No\ of\ Insertions}$$

$$\text{Amortized Cost} = \frac{(Cost\ to\ Insert\ Element\ ) + (Cost\ to\ Copy\ from\ Old\ Array\ to\ New\ Array)}{No\ of\ Insertions}$$

$$Amortized\ Cost = \frac{8+16}{8}$$
$$Amortized\ Cost = \frac{24}{8} = 3$$

Let us generalize this solution for ‘2n’ number of operations. We get consider that we copy the numbers from 1 + 3 + 5 + 7 … so we can see this is the sum of odd numbers. If we see that we have 2n number of operations, we will have N operations will cost us 0 and other n operations will cost us the following:

Cost to Copy Elements = 1 + 3 + 5 + 7 + …. + (2n-1)
= $\sum_{i=1}^{n} 2i - 1$

Applying Arithmetic Progression to find out the sum

$$S_n = \frac{n}{2}(2a + (n-1)d)$$

Where $S_n = Sum\ of\ N\ Elements, n = no\ of\ elements, a = first\ term\ , d = common\ difference$

Let’s find out common difference d
$d = t_n - t_{n-1} = t_2 - t_1 = 3 - 1 = 2$

$$S_n = \frac{n}{2}(\ 2 * 1 + (n-1) * 2)$$
$$S_n = \frac{n}{2}(\ 2 + 2n - 2)$$

$$S_n = \frac{n}{2}(\ 2n)$$

$$\boldsymbol{S_n = n^2}$$

Let's calculate the Amortized cost now:

$$\text{Amortized Cost} = \frac{(Cost\ to\ Insert\ Element\ ) + (Cost\ to\ Copy\ from\ Old\ Array\ to\ New\ Array)}{No\ of\ Insertions}$$

$$Amortized\ Cost\ = \frac{2n + n^2}{2n}$$

$$Amortized\ Cost\ = \frac{n(2+n)}{2n}$$

$$Amortized\ Cost = \frac{(2+n)}{2}$$

$$Amortized\ Cost\ = O(n)$$

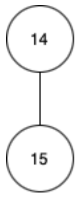
As we can see here the Amortized cost per operation in Fred's table is O(n)

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.

Ans)
P(k): Consider that when we remove the root node of kth order Binomial Tree, we get k binomial trees of smaller orders

$$B_k = B_{k-1} + B_{k-2} + \ldots + B_1 + B_0$$

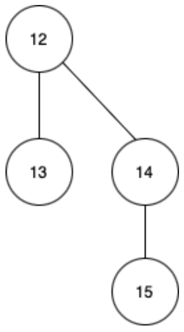
Base Case:
Let's consider that we have a Binomial Tree of order 1 i.e., B_1 when we remove the root node of this tree, we get 1 Binomial Trees of B_0.



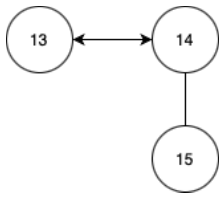
When we remove the root node i.e., 14 we get only a Binomial Tree having a root node as 15 i.e. we get a tree of Order B_0

Consider another base case

Let’s consider that we have a Binomial Tree of order 2 i.e., B_2 when we remove the root node of this tree, we get 2 Binomial Trees of B_1 and B_0.



When we remove the Node 12 from Binomial Heap H1, we get 2 Trees of Order B_1 and B_0



Inductive Hypothesis:

Let’s assume that the hypothesis of when we remove the root node of a Binomial Tree B_{k-1} , it gives us k-1 binomial trees of smaller order

$$B_{k-1} = B_{k-2} + B_{k-3} + .. + B_1 + B_0$$

This equation holds true for B_{k-1} $where\ k \ \in order\ of\ Binomial\ Tree$

Inductive Step:

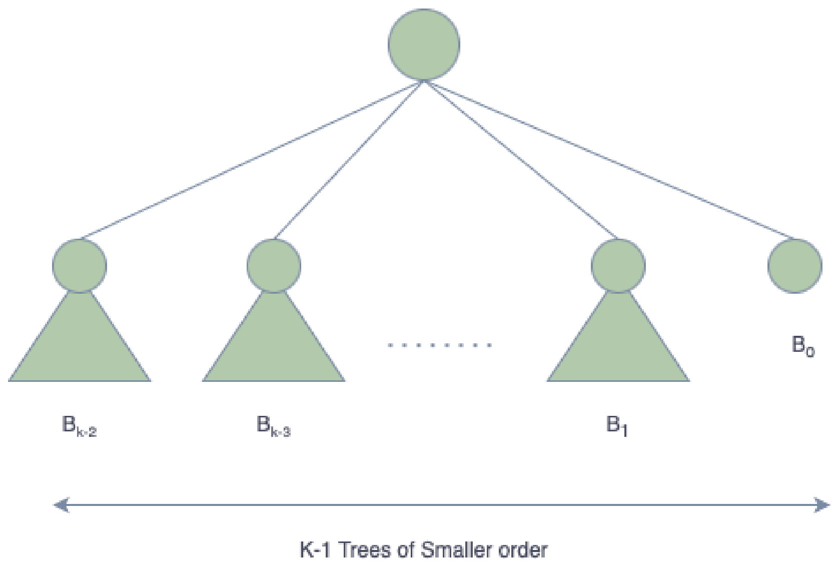
We need to prove that when we remove the root node of Binomial Tree B_k it gives us k binomial trees of smaller order

Proof Inductive Step:

From the Definition of Binomial tree, we know that Binomial Tree is formed by joining 2 Binomial Tree of Order K-1.

$$B_k = B_{k-1} + B_{k-1}$$

Assume we remove the root of one of the Tree B_{k-1} we will get trees of smaller order by inductive hypothesis. We can remove the Node from any one of the B_{k-1} tree and keep the other one as it is.



As seen in this image we know that when we remove the Root node of B_{k-1} tree we will get then we will get k-1 trees of smaller order

$$B_k \; = B_{k-1} + \boldsymbol{B_{k-2}} + \boldsymbol{B_{k-3}} + .. + \boldsymbol{B_1} + \boldsymbol{B_0}$$

The highlighted part is the one which is generated by remove the root node from the one tree B_{k-1} and keeping the other as it is.
Thus, we can see here that when we remove the Root node from a Binomial Tree, we get k trees of smaller order.

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.

Approach:
We create a Min Heap with the cost of all Ropes. We then remove the 2 minimum elements from the heap and sum them up and insert them again in the heap. Until we have just 1 element left in the heap that will be the minimized cost of connecting all the ropes.

Input: A list of length of ropes L1, L2, L3 .. Ln
Output: Minimum cost to connect all ropes.

Algorithm:

```
1. Initialize an empty Min Heap
2. Insert all the elements in the Min Heap
3. Repeat until we have more than 1 element in the heap
    a. Remove the minimum and 2nd minimum element from heap
    b. Sum those 2 elements
        i. Sum = Minimum + 2nd Minimum
    c. Insert the Sum back into the heap
    d. Heapify the Heap to maintain its heap property
4. Return the root node of the heap
```

Insert number in min heap ():

```
1. Insert the element in last leaf node of the heap
2. Heapify the binary heap:
    a. For Nodes in Heap which has been affected by insert
        i. If (parent > children node) we swap the 2 nodes
            1. Children node = Parent
            2. Parent = Children node
```

Remove number in min heap ():

```
1. Replace the last leaf node with the root element in the min heap
2. Remove the last leaf node of the min heap
3. Heapify the Min Heap
    a. For all Nodes in Tree which has been affected by delete
        i. If (parent > children node) we swap the 2 nodes
            1. Children node = Parent
            2. Parent = Children node
```

Here the sum is the total cost of adding the ropes if found by removing the 2 minimum element and adding their sum back into the heap. By following such a pattern, we will always have the minimum cost. The time complexity will be O(nlogn) since we will do n insertion and deletion at max in this case.

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.

Let's consider the approach as we will create a max heap of size M elements i.e., the number of sorted lists. We solve this question using the local and global approach where we calculate the range locally and only save it to global if the range is less than the globally defined one. Initially we insert the initial 1 element from M List's in the Min heap. We also find the max from these elements and store in lmax that is locally calculated max value. We remove the root node from the min heap and find the range of the elements = local max - min from root of heap. If this range is less than globally defined range we will update the min, max and range of the global variable. If range is less, we won't update it and keep the range as it is. We will then insert the next element from the List from which minimum element was removed.

Heap used in this case will store 3 items in it

1. Element Value
2. Index of the list from which it is taken
3. Index of the next element to be picked up from list

Input: M list of N elements containing numbers in sorted order
Output: Range of Elements

Algorithm:

1. Initialize a Min Heap of Size M which can contain M elements at max
   a. Structure of Min Heap will be
      i. Value (Stores the value)
      ii. Index of the list from which it is taken
      iii. Index of the next element to be picked up from list
2. Initialize Variables for Local and Global Values
   a. Lmin = ∞ (Stores the local Minimum values and is set to ∞ initially)
   b. Lmax = -∞ (Stores the local Maximum values and is set to -∞ initially
   c. Gmin = ∞ (Stores the Global Minimum values and is set to ∞ initially)
   d. Gmax = -∞ (Stores the Global Minimum values and is set to -∞ initially)
   e. LRange = ∞ (Stores the local Range and is set to ∞ initially)

```
        f. GRange = ∞ (Stores the Global Range and is set to ∞ initially)
        g. FList = empty ( To store all elements which are in range)
    3. Insert the initial M elements into the heap from the input 2-D array
        a. InsertIntoHeap(arr[i][0])
    4. Find the Max from the inserted M elements initially and store it in Lmax Variable
        a. For Element at 0th Index in all M List:
            i. Lmax = Math.max (Lmax, arr[i][0])
    5. Loop till we don't reach end of any of the M List:
        a. Remove the Root node from the heap (Min element from the Heap)
            i. Lmin = Root node of the heap
        b. Calculate the LRange
            i. LRange = Lmax - Lmin
        c. If Global Range > Local Range update Global Variables
            i. GRange = LRange
            ii. Gmax = Lmax
            iii. Gmin = Lmin
        d. Add new element into heap which is the next element to the current min
           element
    6. Find the Elements which are within the Gmin and Gmax
    7. Loop over all M List
        a. Find 1 element from each list where Number > Gmin and Number < Gmax
        b. Add the element in FList
    8. Return the Global Variable - Flist and GRange
```

Runtime Complexity: $O(N * M)$ here we know that to find the min, max and range we take $O(N * Log\ M)$ since heapification won't take more than $Log\ M$time, but when we need to find 1 the element from all the list and add them to the list it will take O$(N * M)$ thus our net time complexity is O$(N * M)$

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.

Ans 6)

Data Structure: Let's make a data structure containing 2 heaps: 1 Min Heap and 1 Max Heap and we will jumble elements amongst them to make the other operations in the required

New Created Data structure = Min Heap + Max Heap

Here in this case, we will always have the sorted elements in the heap where the medians will be on the root nodes. And if we split the input into 2 equal parts, we can see that the left half will be stored in the max heap and the right half will be stored in the min heap. Th

1. **Find Median in O (1):**

Let's consider that in these 2 Heaps we have the elements are already inserted in them. Here while insertion we will send the higher numbers to min heap and lower numbers in max heap making sure that the median of the list stays in the root nodes. Consider the scenario when elements are even. In such a case the median will be the sum of the root nodes of the heap, and we will calculate the median by taking their average. On the other hand, for odd number of elements the median is always stored in the max heap since we will have the greater number of elements in the max heap than the min heap when the number of elements is odd.

Input: List of N numbers that are already stored in our new data structure containing 2 heaps - min heap and max heap.
Output: Median of the heap
Algorithm:

```
1. If n is odd number (n % 2! = 0)
    a. Median = root of Max Heap
2. If n is even number (n % 2 == 0)
    a. Number 1 = root of Max Heap
    b. Number 2 = root of Min Heap
    c. Median = $\frac{Number\ 1 + Number\ 2}{2}$
3. Return Median
```

Since in this we are just getting the root of Max Heap or taking the average of Root of Min heap or max heap. We know that the retrieval of the root elements takes constant time i.e., O(1) time for retrieval of the median in this data structure.

2. Extract Median in O (log n)

In our newly created data structure, we know that the median element is always at the root node of the binary heaps. If the number of elements in the list are odd we remove the element of the max heap or else if the number of elements are even then we will remove the root element of both the min heap and max heap
Input: List of N numbers that are already stored in our new data structure containing 2 heaps - min heap and max heap
Output: List of N number that are in our data structure where the old median element has been removed

Algorithm:

```
1. If the number of elements is odd (num % 2! =0)
    a. Replace the last leaf node with the root element in the max heap
    b. Remove the last leaf node of the Max heap
    c. Heapify the binary tree
        i. For all Nodes in Heap
            1. If (parent < children node) we swap the 2 nodes
                a. Children node = Parent
                b. Parent = Children node
            2. Update the HashMap with updated location
    d. Return the median element deleted
2. If the number of elements is even (num % 2==0)
    a. Replace the last leaf node with the root element in the max heap
    b. Remove the last leaf node of the Max heap
    c. Heapify the Max Heap
        i. For all Nodes in Heap
            1. If (parent < children node) we swap the 2 nodes
                a. Children node = Parent
                b. Parent = Children node
            2. Update the HashMap with updated location
    d. Replace the last leaf node with the root element in the min heap
    e. Remove the last leaf node of the min heap
    f. Heapify the Min Heap
        i. For all Nodes in Tree
            1. If (parent > children node) we swap the 2 nodes
                a. Children node = Parent
                b. Parent = Children node
            2. Update the HashMap with updated location
    g. Return median = (Root of Min Heap + Root of Max Heap) / 2
```

Since in this max we can swap the nodes is till the height of the heap i.e., O(log n) and thus the time complexity of extract median is dependent on the heapify of the heap which results to O(log n) in worst case.

3. Insertion in O (log n)

We insert elements in the 2 heaps in an ordered fashion making sure that the elements that are higher are store in the min heap and the lower value elements are stored in max heap, if we were to divide the input into 2 equal halves. We also store their accurate location of each node in a HashMap <Key,Value> where key can have the number to be stored and Value will have their address.

Input: List of Numbers which are in unsorted order
Output: None (Element inserted in the Data structure)
Algorithm:

```
1. Initialize 2 empty heap, 1 min heap and 1 max heap
2. For all numbers in list:
    a. If the number < root of max heap
        i. Insert number in max heap(number)
        ii. Insert the location in the HashMap < Number, Address >
    b. Else
        i. Insert number in min heap
        ii. Insert the location in the HashMap < Number, Address >
        c. If no of elements in Max heap > no of elements in Min heap
                i. Extract the root node of max heap
               ii. Insert the number in min heap (number)
              iii. Update the location in HashMap
        d. Else
                i. Extract the root node of Min heap
               ii. Insert the number in Max heap (number)
              iii. Update the location in HashMap
```

```
Insert number in max heap ():
    1. Insert the element in last leaf node of the heap
    2. Heapify the binary heap:
        a. For all Nodes in Heap
                i. If (parent < children node) we swap the 2 nodes
                    1. Children node = Parent
                    2. Parent = Children node
Insert number in min heap ():
    3. Insert the element in last leaf node of the heap
    4. Heapify the binary heap:
        a. For all Nodes in Heap
                i. If (parent > children node) we swap the 2 nodes
                    1. Children node = Parent
                    2. Parent = Children node
```

Since in this case the max number of swaps that can happen are the height of the binary heap i.e., O (log n) time complexity. And we will store them in the HashMap and at most updating in HashMap would be O(log n) this net time complexity of insert is O(log n)

4. Delete in O (log n)

Input: List of Numbers which are in unsorted order and the element to be deleted,
Algorithm

```
    1. Find the Number's Address in the HashMap which should take constant time
    2. If the number is in Max Heap
        a. Replace the last leaf node with the root element in the max heap
        b. Remove the last leaf node of the Max heap
        c. Heapify the Max Heap
                i. For all Nodes in Heap
                    1. If (parent < children node) we swap the 2 nodes
```

            a. Children node = Parent
            b. Parent = Children node
3. Else If the number is in Min Heap
    a. Replace the last leaf node with the root element in the min heap
    b. Remove the last leaf node of the min heap
    c. Heapify the Min Heap
        i. For all Nodes in Tree
            1. If (parent > children node) we swap the 2 nodes
                a. Children node = Parent
                b. Parent = Children node

Since in this case the max number of swaps that can happen are the height of the binary heap i.e., O (log n) time complexity

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.

Ans 7)

Input: Array of Plants planted in a row
Output: Array of Lamps to be places such that each plant is less than 4meters away

Algorithm:
The problem can be solved by the greedy algorithm. The algorithm works as follows:
1. Initialize the set of lamps to be empty.
2. Sort the plants along the road in increasing order of their distances from the western endpoint.
3. Place the first lamp at the right of the first plant 4 meters away.
4. Iterate through the remaining plants and place a Lamp at the right of the current house 4 meters away. if it is more than four miles away from the closest existing base station.
5. Repeat step 4 until all plants have been processed.

Time Complexity: $O(N * Log\ N)$, here the time complexity is $(N * Log\ N)$ because we need to sort the plants initially and then we need to place the lamp after every 4 meters. Thus, sorting in the quickest order takes $(N * Log\ N)$ and then placing the lamp will take O (1) thus net time complexity is $O(N * Log\ N)$

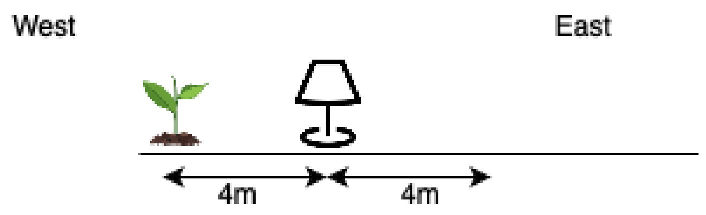
Proof of Correctness:

Predicate P(k): Consider that our algorithm ALG is as good as the Optimal solution OPT

Consider our Algorithm has the Lamps at location: $S_1, S_2, S_3 \ldots.\ S_m$
Optimal Solution places the Lamps at location: $T_1, T_2, T_3 \ldots.\ T_i$

Proof by Induction on Lamps

Base Case:
Consider only 1 Plant and the Algorithm will place the Lamp at location that is 4m away from the Plant to the right. Even Optimal Algorithm will do the similar approach placing the lamp at the similar location < 4miles apart since we only have this one approach. This will satisfy the predicate.



Inductive Hypothesis: Assume the algorithm works for c-1 lamps, where c is a positive integer.

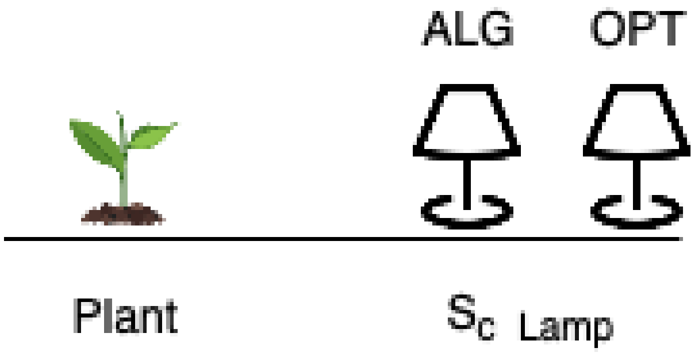
Inductive Step: We need to prove that this works for c lamp.

Applying Inductive Hypothesis now we know that our algorithm is not as worse as the optimal solution.
Let's consider that for c-1 lamp we have already placed. Now when we encounter a new plant that is > 4 meters from the c-1 lamp to right. We have 2 choices for putting the lamp over optimal solution

Consider the 2 scenarios where our lamp could have been placed

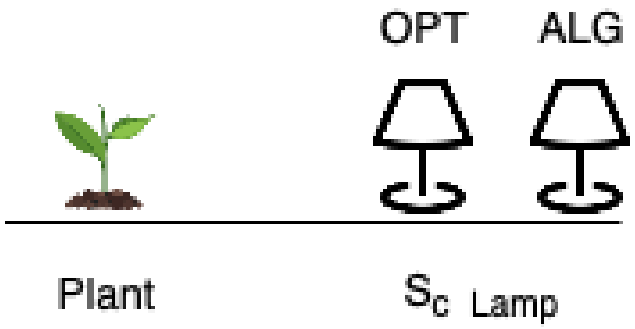
Case 1 : In this case we can see that Optimal Solution places the lamp after what our Algorithm has placed, Such a scenario is impossible to exist, Our Algorithm will always place a lamp after walking 4 meters from the plant and in this case Optimal cannot place it > 4 meters. It will violate the optimality of the solution and it contradicts the fact that a Plant needs a lamp which is at max 4 meters away from it.

Case 1



Case 2 that can exist since if we place at 4m the optimal solution can place at <=4 meters. So, by contradiction we can prove that our algorithm will place the lamp at the location better than or equal to optimal solution.

Case 2



Hence it proved that for k plants our algorithm will is at least as good as the optimal solution and not worse. Thus, we have proved the correctness of our algorithm.

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.

Ans 8)

Input: List of Plant's Minimum amount of water to stay alive (l1,l2,l3 ..ln) and List of bottle with capacity received (c1,c2,c3 .. cn)
Output: True or False if the arrange is possible or not

Algorithm:

1. Sort the Water Capacity needed by plants (l1,l2,l3..ln) in ascending order
2. Sort the Bottle of water (c1,c2,c3 ..cn) in ascending order
3. Loop through n items
    a. If (l[i] < c[i])
        i. return **false** since the arrangement cannot fulfill the requirement
4. Return **true** since we could find all l[i] which were >= to c[i] and thus the arrangement is now optimal

The time Complexity is O(nlogn) since we are sorting the array and then we will be traversing and matching if water needed is satisfied by bottle which takes O(n) so in total it will take at max O(nlogn) in the worst case time complexity.

Proof of Correctness:

Consider our ALG will use bottles A1,A2,A3 … An while the optimal has used the sequence O1, O2, O3 … On.

Predicate P(k): Consider that our algorithm ALG is as good as the Optimal solution OPT to create the arrangement from the available water bottles to water the plants.

Proof by Induction on Bottles:

Base Case: Let's consider that we have only 1 Plant which needs L1 amount of water to stay alive and then we have 1 Bottle we have received which has a capacity C1 there can be 2 cases
1. $L1 > C1$- In this case the output will be false since we cannot fulfill the requirement of the plant using the bottle having water C1
2. $L1 \leq C1$- In this case the output will be true since we can fulfill the requirement by giving the plant l1 amount of water
Here the Optimal and our Algorithm have only a single choice and it satisfies the predicate.

Inductive Hypothesis:
Assume that the solution works for k-1 bottles and k-1 plants

Inductive Step: We need to prove that our Algorithm works for K bottles and K plants.

Proof for Inductive Step: We know that we have sorted our Input's in ascending order and our algorithm works well for all input where the number of bottles and plants are k-1.

Consider we have kth plant which needs minimum water Lk and Bottle with water of capacity Ck. Each bottle here can only water 1 plant as per the given algorithm and we have already fulfilled it for plants 1 to k-1. Here we can have 2 cases possible

Case 1: We consider that we have an arrangement for k-1 bottles. Which means plants 1 to k-1 have already been satisfied by the 1 to k-1 bottles of water. And then for now Kth bottle we check if $Ck \geq Lk$ then we can say that that plant can be watered, and an arrangement exist but if that is not the case where $Ck < Lk$ Then we consider that such an arrangement cannot exist.

Consider that kth bottle capacity Ck < kth plant Lk need we can swap and fix the arrangement. By **contradiction** we know that this case cannot exist since we are sorting the input in ascending order and thus if the all the bottles we have encountered before are of either the same or size less than the bottle we have right now. Thus the rearrangement by swapping of the bottle will never give us an arrangement.

Case 2: We consider that we did not have an arrangement for k-1 bottles. Which means plants 1 to k-1 were not satisfied by the 1 to k-1 bottles of water. Thus we can say there won't be any solution for kth bottle.

Consider a case where if for the m bottle where $1 < m < k-1$ we say that if we borrow the next upcoming bottles and then use their water, then can an arrangement work out. This case cannot exist by **Contradiction** when our algorithm sorts the input list of water bottle and plants needs for water. If we take a m+1th bottle for mth plant then it might happen that for the m+1th plant will be short of water and then if we keep going forward then in the end the last plant won't have enough water and our arrangement will fail and we will return false.

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.

Ans 9)

Input: N Different empty bottle's p1, p2...pn and W as the water that we need to store
Output: The number of bottles that will be needed to fill water W. Our assumption here is that W will always be less than or equal to sum of all the capacity of empty bottles.
Algorithm:

1. Sort all the empty bottles in descending order of their capacity
2. Initialize a counter for the number of bottles needed to store the entire water W, and set it to zero.
3. For each bottle in the sorted list:

    a. If the water filter has water more water than current bottle capacity, then subtract the bottle capacity and increment counter
        i. Water Filter Capacity = Water Filter Capacity - Bottle Capacity
        ii. Counter = counter + 1
    b. If the water filter contains less water than the current bottle's capacity, fill the current bottle with the remaining water, and break out of the loop.
4. Return the counter.

// Additional Edge case which can occur
In the case where $W \geq \sum p_i$ then our Algorithm will return -1 this is an addition we can add to algorithm to cover

```
If Water filter contains water even after all water is exhausted
        Return -1
    Else
        Return counter
```

Time Complexity : O(nlogn) since we will be sorting our input array which takes O(nlogn) at worst case and then we will traverse the array to get the minimum number of bottles needed which takes O(n) at max. so the net time complexity is O(nlogn) in the worst case.

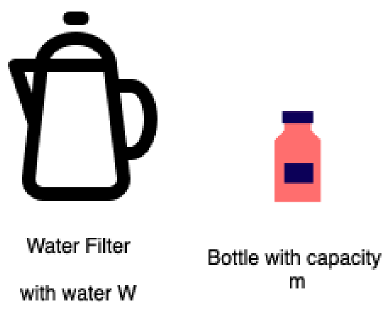
Proof of Correctness:

Predicate P(k): Our Algorithm chooses the fewest number of bottle's which are less or equal to the number of bottles of Optimal

Consider our Algorithm uses the Bottles ALG: $P_1, P_2, P_3 \ldots P_k$
Consider Optimal Solution uses the Bottles ALG: $O_1, O_2, O_3 \ldots O_c$

Proof by Induction on Bottle's

Base Case: Consider a case where we only have k = 1, which means we only have 1 bottle to hold all the water. Our Greedy Algorithm will choose the largest i.e., the only available bottle and even optimal will choose the same one since this is the only feasible solution that will produce an optimal solution



W <= M

Inductive Hypothesis: Suppose the statement holds for k-1 bottles, i.e., the greedy algorithm produces an optimal solution when there are k-1 bottles needed to hold the water.

Inductive Step: We need to prove that it works for k bottles.

Let k be the fewest number of bottles needed to store the water and let S be some optimal solution.

Let's denote the 1st bottle chosen by our algorithm be p'. Here we can have 2 cases

1. If S contains p'

Then we can see that our algorithm and the optimal algorithm is the most optimal one and it will work.

2. If S doesn't contain p'

In this case as we can see that S did not consider the biggest bottle p' since our algorithm will choose the bottle with maximum capacity. Thus, it proves that all bottles in S are smaller than capacity p'. We can then remove any bottle and empty its water in p' and create a new set solution S' = S - {p} U {p'} which will also contain the same k number of bottles. Thus S' will become the optimal solution and as we know in Induction if we can solve it for a subproblem we can apply the same logic to solve the bigger problem, thus by solving it at the level of 1st bottle we can say for any number of k bottles this algorithm will work by Induction