Analysis of Algorithms Homework 2

USC ID: 3725520208
Name: Anne Sai Venkata Naga Saketh
Email: annes@usc.edu

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:

First let's find the time complexity for the function0()

Here, we get three cases, one for odd numbers, one for the powers of 2 and the other for the even numbers,

Case 1: Odd numbers:

For function0(), if the input is an odd number, then the function exits with one step (if (x == 0 || (x % 2 != 0)) return false;) by returning **FALSE**

Hence the time complexity of function0() in the case of odd numbers input is $O(1)$.

Example: let's consider an example of input:3
The function0() exits at the step if (x == 0 || (x % 2 != 0)) return false, since the input is an odd number and then it also returns false.

Case 2: Number is a power of 2

For function0(), if the input is an even number that is a power of 2, then the function always divides the input by 2 (if (x > 1) return function0(x/2);) until the input number is equal to 1, and then the function returns true at (if (x == 1) return true;) by returning **TRUE**

Hence the time complexity in the case of even numbers that are a power of 2 is $\boldsymbol{O(log(n))}$

Example: let's consider an example of input:8
The function0() then returns function0(4), and then function0(2), and then returns function0(1) exits at the step if (x == 1) return true, since the input is an even number and a power of 2 it returns true.

Case 3: Number is an even number and not a power of 2

For function0(), if the input is an even number that is NOT a power of 2, then the function always divides the input by 2 (if (x > 1) return function0(x/2);) until the input number is an odd number and then the function exits at the step (if (x == 0 || (x % 2 != 0)) return false). By returning FALSE

Hence the worst-case time complexity in the case of even numbers that are NOT a power of 2 is $\boldsymbol{O(logn)}$.

Example: let's consider an example of input:6
The function0() then returns function0(3), and then function0() returns TRUE as the input is an ODD number.

<u>Now let's find the time complexity for function1()</u>

Here also, we get three cases, one for odd numbers, one for the powers of 2 and the other for the even numbers,

Case 1: Odd numbers:

For function1(), if the input is an odd number, then the function0() returns FALSE and then exits by printing only one print statement.

Hence the time complexity of function1() in the case of odd numbers input is $\boldsymbol{O(1)}$.

Example: let's consider an example of input:3
The function0() fails **FALSE** and exits after printing only one print statement.

Case 2: Number is a power of 2

For function1(), if the input is an even number that is a power of 2, then the function0() will return **TRUE** and the print statement will be executed 'n' number of times, where 'n' is the input.

Hence the time complexity in the case of even numbers that are a power of 2 is $\boldsymbol{O(n)}$

Example: let's consider an example of input:8
The function0() will return **TRUE** and then enters the for loop and prints the number from 1 to n. which is executing 'n' times. Here the time complexity is $O(n)$

Case 3: Number is an even number and not a power of 2

The function0() shall return a **FALSE**, and then exits by printing only one print statement.

Hence the worst-case time complexity in the case of even numbers that are NOT a power of 2 is $\boldsymbol{O(1)}$.

Example: let's consider an example of input:6
The function0() will return **FALSE** and then exits after printing only one print statement.

<u>Now let's find the time complexity for function2()</u>

Now the function2() executes the for loop runs 'n' times, then the time complexity of the function shall be $O(n) * \text{ TC of function1()}$

<u>Finding the Amortized Cost for Function1():</u>

In each input, the following shall be the number of terms under each category.

Number of odd terms : $\boldsymbol{n/2}$
Number of even terms that are powers of 2 : $\boldsymbol{log(n)}$
Number of even terms that are not powers of 2: $\boldsymbol{n/2 - log(n)}$

Computing the total time complexity:

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

$$= \frac{n}{2} + \text{nlog}(n) + \left(\frac{nlog(n)}{2} - \log(n)^2\right)$$

$$= \frac{n}{2} + \text{nlog}(n) + \frac{nlog(n)}{2} - \log(n)^2$$

$$= \frac{n}{2} + \frac{3nlog(n)}{2} - \log(n)^2$$

Ignoring the smaller degree terms $\frac{n}{2}$ and $(\log(n))^2$, then we get $\frac{3nlog(n)}{2}$

$$Amortized\ Cost = \frac{Total\ Cost}{number\ of\ operations}$$

The amortized time complexity for function1() is $\frac{\frac{3nlog(n)}{2}}{n}$

$$= \frac{3nlog(n)}{2n}$$

$$= O(\log(n))$$

The amortized time complexity of the function2() shall be TC of function2() * Amortized TC of function1().

$$= \boldsymbol{O(n\log(n))}$$

Therefore, the Amortized time complexity of function2() shall be $\boldsymbol{O(n\log(n))}$. The tightest bound shall also be $\boldsymbol{O(n\log(n))}$

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?**

Solution: As per what we have discussed in the lecture that doubling the size of the array has the best results on the amortized cost per insertion which is calculated by the below formula.

$$Amortized\ Cost = \frac{Total\ Cost}{number\ of\ operations}$$
$$= \frac{Sum\ of\ Cost\ per\ insertions + sum\ of\ all\ copy\ cost}{number\ of\ operations}$$

Considering that we double the array size while insertions, the amortized cost would be.

Insert operation	**Old size of array**	**New Size of array**	**Array**	**Cost for copy**
1	1	-	1	-
2	1	2	1 2	1
3	2	4	1 2 3	2
4	-	-	1 2 3 4	-
5	4	8	1 2 3 4 5	4
6	-	-	1 2 3 4 5 6	-
7	-	-	1 2 3 4 5 6 7	-
8	-	-	1 2 3 4 5 6 7 8	-
9	8	16	1 2 3 4 5 6 7 8 9	8
10	-	-	1 2 3 4 5 6 7 8 9 10	-

Generalizing the above series of operations by assuming the cost for each insert is '1' and the cost to copy each element into a new array is '1', we get

$$AC = \frac{Cost\ for\ Total\ Insertions + Cost\ for\ total\ copy}{number\ of\ operations}$$

$$= \frac{n + \sum_{k=0}^{k=\log(n)} 2^k}{n}$$

$$= \frac{n + \frac{1(2^{\log(n)+1} - 1)}{2 - 1}}{n}$$

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

$$= \frac{n + 2n - 1}{n}$$

$$= \frac{3n - 1}{n}$$

$$= \frac{O(3n-1)}{n}$$

$$= O(1)$$

Following the similar approach for Fred Hacker's claim, we increase the size of the array by '2' over the previous size for each insert.

Assuming the cost for each insertion is '1' and the cost for each copy operation is '1', we get the following.

Insert operation	**Old size of array**	**New Size of array**	**Array**	**Cost for copy**
1	1	-	1	0
2	1	3	1 2	1
3	3	5	1 2 3	2
4	5	7	1 2 3 4	3
5	7	9	1 2 3 4 5	4
6	9	11	1 2 3 4 5 6	5
7	11	13	1 2 3 4 5 6 7	6
8	13	15	1 2 3 4 5 6 7 8	7
9	15	17	1 2 3 4 5 6 7 8 9	8
10	17	19	1 2 3 4 5 6 7 8 9 10	9…

Steps:

1. Initialize/Consider the array of size '1'
2. From the second insert operation, we increase the size of the array by '2' for every insertion.
3. Once the array size is increased, we copy all the old elements from the old array into the new array.

For the above sequence of operations, the amortized cost per operation is calculated using the formula

$$AC = \frac{Cost\ for\ Total\ Insertions + Cost\ for\ total\ copy}{number\ of\ operations}$$

$$= \frac{n + \sum_{k=0}^{k=n-1} k}{n}$$

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

$$Since, the\ sum\ of\ n\ terms\ in\ a\ AP\ is \frac{n(a+l)}{2},$$

$$= \frac{n + \frac{(n^2 - n)}{2}}{n}$$

$$= \frac{n^2 - n}{2n} + 1$$

$$= \frac{n}{2} - \frac{1}{2} + 1$$

$$= \frac{n}{2} + \frac{1}{2}$$

$$= O(n)$$

Assuming if the size of the array is increased by 2, only when the array is full and not for every insertion. Then we get, the following Amortized Cost.

Insert ope-ration	**Old size of array**	**New Size of array**	**Array**	**Cost for copy**
1	1	-	1	0
2	1	3	1 2	1
3	3	5	1 2 3	0
4	5	7	1 2 3 4	3
5	7	9	1 2 3 4 5	0
6	9	11	1 2 3 4 5 6	5
7	11	13	1 2 3 4 5 6 7	0
8	13	15	1 2 3 4 5 6 7 8	7
9	15	17	1 2 3 4 5 6 7 8 9	0
10	17	19	1 2 3 4 5 6 7 8 9 10	9…

Steps:

1. Initialize/Consider the array of size ‘1’

2. From the second insert operation, **if the size of the array is full**, we increase the size of the array by '2' over the previous size.
3. Once the array size is increased, we copy all the old elements from the old array into the new array and then do the Insertion of the new elements.

For the above sequence of operations, the amortized cost per operation is calculated using the formula

$$AC = \frac{Cost\ for\ Total\ Insertions + Cost\ for\ total\ copy}{number\ of\ operations}$$

$$= \frac{n + \sum_{k=1}^{k=n/2} k}{n}$$

$$= \frac{n + \frac{n/2(1 + 1 + \left(\frac{n}{2} - 1\right)2)}{2}}{n}$$

$$Since, the\ sum\ of\ n\ terms\ in\ a\ AP\ is \frac{n(a+l)}{2},$$

$$= \frac{n + \frac{n/2(2 + n - 2)}{2}}{n}$$

$$= \frac{n^2/2}{2n} + 1$$

$$= \frac{n}{4} + 1$$

$$= O(n)$$

Therefore, by following Fred Hacker's approach, the amortized cost per insertion shall be $O(n)$.

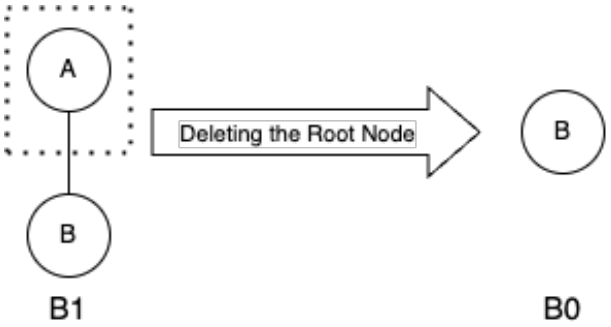
3. Prove by induction that if we remove the root of a k-th order binomial tree, it results in k binomial trees of the smaller orders. You can only use the definition of B_k. Per the definition, B_k is formed by joining two B_{k-1} trees.

Solution:

As per mathematical induction, we must first prove for base case

Step 1: Base Case: Proving the above statement is true for B_1 and B_2

Proving the base case is true for B_1

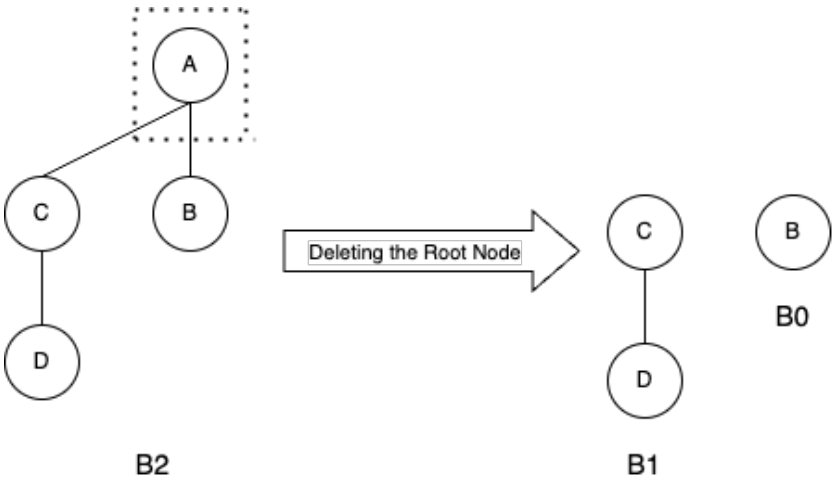


The above tree is a Binomial tree of order '1' (Number of nodes = $2^1 = 2$).

When the root node has been deleted, then it gives rise to a Binomial tree of order '0' (Number of nodes = $2^0 = 1$)

As per the definition of the binomial trees, $B_1 = B_0 + B_0$, therefore when we delete a node, it gives rise to 1 binomial tree B_0.

Proving the base case is true for B_2



The above tree is a Binomial tree of order '2' (Number of nodes = $2^2 = 4$).

When the root node has been deleted, then it gives rise to 1 Binomial tree of order '0' (Number of nodes = $2^0 = 1$) and 1 Binomial tree of order '1' (Number of nodes = $2^1 = 2$)

(Total Number of nodes after deleting the root is 3).

As per the definition of the binomial trees, $B_2 = B_1 + B_1 \Rightarrow B_1 + B_0 + B_0$ therefore when we delete a node, it gives rise to 2 binomial trees of order B_0 and B_1.

Hence, for B_1 and B_2 the given statement, that when the root of a k-th order binomial tree is deleted, it gives rise to k Binomial trees of lower order.

Step 2: Induction Hypothesis: In this step we assume that the given statement is true.

Assuming that the given statement is true for B_k, i.e., when the root node of a binomial tree B_k is deleted, it gives rise to k binomial trees of lower order.

Step 3: Induction Step: We must prove that the above statement is true for a binomial tree of order k+1 (B_{k+1})

As per the definition of a binomial tree B_{k+1} is formed by joining two B_k binomial trees.

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

⇨ $B_{k+1} = B_k + B_{k-1} + B_{k-1}$

⇨ $B_{k+1} = B_k + B_{k-1} + B_{k-2} + B_{k-2}$

⇨ $B_{k+1} = B_k + B_{k-1} + B_{k-2} + B_{k-3} + B_{k-3}$

⇨ $B_{k+1} = B_k + B_{k-1} + B_{k-2} + B_{k-3} + B_{k-4} + B_{k-4}$

⇨ …….

⇨ $B_{k+1} = B_k + B_{k-1} + B_{k-2} + B_{k-3} + B_{k-4} + \ldots\ldots\ldots + B_1 + B_1$

⇨ $B_{k+1} = B_k + B_{k-1} + B_{k-2} + B_{k-3} + B_{k-4} + \ldots\ldots\ldots + B_1 + B_0 + B_0$

Now, if we delete the root node of the binomial tree (B_0), then it gives rise to the below equation.

⇨ B_{k+1} – (root node) = $B_k + B_{k-1} + B_{k-2} + B_{k-3} + B_{k-4} + \ldots\ldots\ldots + B_1 + B_0$

i.e., After deleting the root node of B_{k+1}, it gives rise to k+1 Binomial trees of lower order they are, $B_k + B_{k-1} + B_{k-2} + B_{k-3} + B_{k-4} + \ldots\ldots\ldots + B_1 + B_0$

The given statement is true for binomial trees of order k+1 (B_{k+1})

As per the mathematical induction, we have proved that the given statement for holds true for all values of k. i.e., when the root node of a binomial tree B_k is deleted, it gives rise to k binomial trees of lower 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.**

Solution:

Since we need to get a minimum cost, we always need to join the smallest length of ropes so that the cost would be minimized.

In case, if we join the ropes of higher length, then in the successive iterations of joining them with other ropes, we need to add them again and again, which shall increase the cost of operations.

The Algorithm to minimize the cost of joining ropes for the above would be as follows

Step 1: Initialize a **'cost'** variable to 0

Step 2: Consider/Insert all the lengths of the ropes in an array

Step 3: While array length is greater than 1, do the following steps.

Step 3.1: Initialize a **'sum'** variable to 0

Step 3.2: Sort the array in the ascending order

Step 3.3: $sum = 1st\ term\ in\ ascending\ sorted\ array(1st\ minimum\ in\ the\ array)\ + 2nd\ term\ in\ ascending\ sorted\ array(2nd\ minimum\ in\ the\ array)$

Step 3.4: Remove both the 1st and 2nd minimum terms from the array

Step 3.5: Insert the 'sum' value into the array

Step 3.6: $cost\ =\ cost\ +\ sum$

Step 4: The value of 'cost' returned after the loop shall be the total cost for the entire process.

In the above process, since we are sorting the array 'n' number of times the time complexity will be $O(n^2 log(n))$ for the above algorithm.

So as an alternative we can also use the min-heap so, that the time complexity will be lower.

Algorithm using a min-heap for the above shall be:

Step 1: Initialize a **'cost'** variable to 0

Step 2: Insert all the lengths of the ropes into a min-heap

Step 3: While heap is not empty:

Step 3.1: Initialize a **'sum'** variable to 0

Step 3.2: $sum = 1st\ minimum\ term\ in\ min\ heap + 2nd\ minimum\ term\ in\ min\ heap$

Step 3.3: Remove both the 1st and 2nd minimum terms from the min-heap

Step 3.4: Insert the 'sum' value into the min-heap again

Step 3.5: $cost = cost + sum$

Step 4: The value of 'cost' returned after the loop shall be the total cost for the entire process

The time complexity of the above algorithm shall be lower than the previous case as the Insert and Delete from a heap shall be executed in $O(log(n))$ time.

Example by applying the above algorithm

Steps	**Description**	**L1= 1**	**L2= 2**	**L3= 3**	**L4= 4**	**L5=5**	**L6= 6**	**Cost**
1	Add L1 and L2	3		3	4	5	6	**3**
2	Add L3 to the previous	6			4	5	6	**6**
3	Add L4 and L5	6			9		6	**9**
4	Add L6 to the value of L1 + L2 and L3	0			9		12	**12**
5	Add the remaining values	21						**21**

The total minimized cost for joining the ropes in 21+9+12+6+3 = **51.**

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:

Given that there are M sorted lists that contain a total of N positive integers. We need to make a list such that it includes at least one element from each of the M sorted lists and shall generate a smaller range.

A range of a list is defined as the difference between the maximum and the minimum element of the list.

If the difference between the maximum and the minimum element shall be the smallest, then we shall get the smallest range from the list.

Algorithm to find the smallest range by including each element from each of the M sorted lists is:

Step 1: Initialize a min-heap of size **'M'**.

Step 2: Initialize a variable **'final_min'** and assign a value of **'infinity'** to it.

Step 3: Since all the 'M' lists are already sorted, we insert the minimum element from each of the 'M' lists into the min heap we initialized. **Assuming we create the min-heap, the TC shall be** $\boldsymbol{O(Mlog(M)), where\ M\ is\ the\ number\ of\ sorted\ lists\ given}$

Step 4: The difference between the root and the largest element of the heap shall be the smallest range possible. Assign that difference (maximum element of the heap – minimum element of the heap) to a variable **'heap_min'**

Step 5: If $final\_min > heap\_min$ then,

Step 5.1: $final\_min = heap\_min$

Step 5.2: Apply DeleteMin() on the Min-heap, and then insert the next smallest element from the list to which the deleted element/root of the min-heap belongs to. **TC is** $\boldsymbol{O(log(M))\ for\ deleting\ and\ O(log(M))\ for\ inserting\ the\ element\ into\ the\ min-heap\ where\ M\ is\ the\ heap\ size, and\ the\ number\ of\ sorted\ lists}$.

Step 5.3: else then, the current value of the **final_min** is the smallest range possible.

Step 6: Repeat **Step 4 and Step 5**, until at least one list of the 'M' sorted lists are completely traversed., then we print the elements in the heap to provide the list of elements that provide the smallest range.

TC is $O(N)$ *where N is the total number of positive integers in the given lists*.

Assuming there is a total of 'N' integers and 'M' sorted lists, The worst-case time complexity of the above algorithm would be $\boldsymbol{O(Mlog(M) + 2Nlog(M))}$

Explaining the above algorithm with an example.

Consider the following lists.

M1 = {3,4,6}

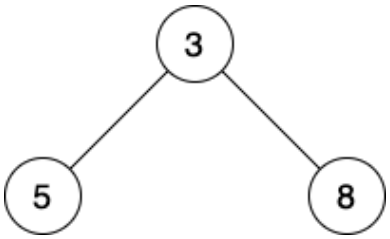
M2 = {5,7,9}

M3 = {8,10,11}

Here 'M' = 3 and 'N' = 9

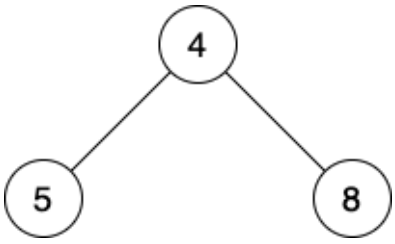
Step 1: We Initialize final_min = infinity, and heap_min = 0

Step 2: As per the algorithm, since the M lists are already in sorted order. We select the minimum/first element from each of the list and insert into a min-heap



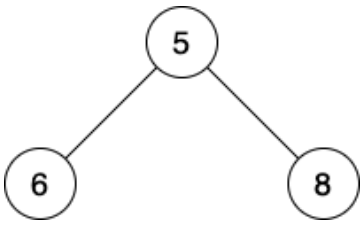
Step 3: Now, the root of the min-heap is '3'. Calculating the $heap\_min\ (max\ of\ the\ heap - min\ of\ the\ heap) = 8 - 3 = 5$

Step 4: Since heap_min < final_min, then final_min = 5, and we perform the deleteMin() operation and then insert the next element from list M1 into the heap which is '4'.

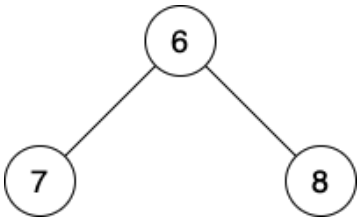


Step 5: Now repeating **Step 3 and Step 4,** we get $final\_min = 4\ (i.e., 8 - 4)$

Step 6: Now repeating **Step 3 and Step 4,** we get $final\_min = 3\ (i.e., 8 - 5)$



Step 7: Now repeating **Step 3 and Step 4,** we get $final\_min = 2\ (i.e., 8-6)$



Since we have completely traversed the list M1, we can stop the execution of the algorithm, and the smallest range containing at least one element from each of the M sorted lists shall be '2'.

Since We have stopped at the end of list M1, the list of elements that generate the smallest range shall be the ones that are in the heap at the end.

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 operations. You are not required to prove the correctness of your algorithm.

Solution:

Given that we need to construct a data structure that has the above properties, to find the median in $O(1)$ time and to extract the median from the data structure should be taking $O(logn)$ time.

The insertion into the data structure and the deletion from the following data structure takes $O(log(n))$ time where 'n' is the number of elements.

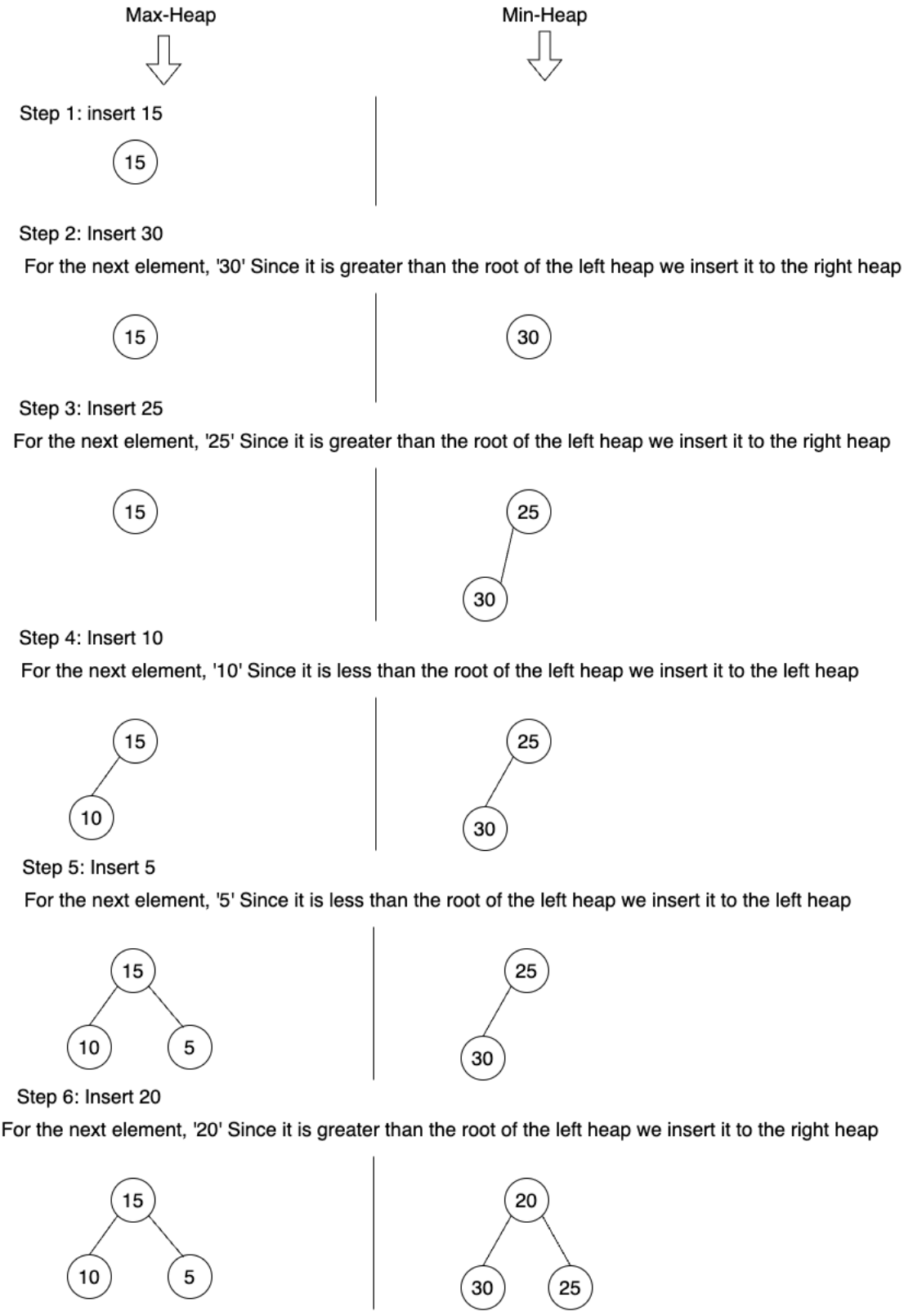
Median: A median is the middle element in a sorted list.

If there are even number of elements in the list, then the median of the list shall be the average of the middle 2 elements.

The Data structure to have the above properties is as follows,

1. We need to construct 2 heaps, one max-heap and one min-heap in such that, assume the left heap we are going to construct is a max-heap and the right heap we construct is a min-heap.
2. While inserting the first element, we can insert it into the left heap (max-heap)
3. For every element 'e' that we insert, based on the below criteria
    a. If the element 'e' is less than the root of the left heap (max-heap), then we need to insert it in the left heap (max-heap).
    b. Else if the element 'e' is greater than the root of the left heap (max-heap), then we insert it in the right heap (min-heap).
    c. Check if the difference in number of elements in the left heap (max-heap) and the right-heap (min-heap) is more than 1, if the difference is more than 1 then, do the following operations
        i. If the number of elements on the left heap is more than the number of elements in the right heap, then do a DeleteMax() operation on the left heap (max-heap) and insert that element in the right-heap (min-heap)
        ii. If the number of elements on the right heap is more than the number of elements in the left heap, then do a DeleteMin() operation on the right heap (min-heap) and insert that element in the left heap (max-heap)
4. After we construct both the min-heap and the max-heap following the above sequence of operations,
    a. If there are even number of elements, then the size of the max-heap and the min-heap shall be the same. The median shall be the average of both the root elements i.e., $\frac{root\ of\ min{-}heap + root\ of\ max{-}heap}{2}$
    b. If the number of elements is odd, and the number of elements in the left heap are greater than the number of elements in the right heap, then the median shall be the root of the left heap (max-heap), else if the number of elements in the right heap are greater than the number of elements in the left heap, then the median shall be the root of the right heap (min-heap).

Explanation with an example: Let's assume we have the following elements: 15, 30, 25, 10, 5, 20. **Followed by in the next page.**



Since there are even number of elements, the median shall be the average of root of both the min-heap and max-heap.

The given operations shall work as follows:

1. **Find Median takes $O(1)$:**

   After we construct both the min-heap and the max-heap following the above sequence of operations,

   If there are even number of elements, then the size of the max-heap and the min-heap shall be the same. The median shall be the average of both the root elements i.e., $\frac{root\ of\ min-heap + root\ of\ max-heap}{2}$

   If the number of elements is odd, and the number of elements in the left heap are greater than the number of elements in the right heap, then the median shall be the root of the left heap (max-heap), else if the number of elements in the right heap are greater than the number of elements in the left heap, then the median shall be the root of the right heap (min-heap).

2. **Extract Median takes $O(log(n))$:**

   Case 1: If the number of elements is odd, and the number of elements in the left heap are greater than the number of elements in the right heap, then the median shall be the root of the left heap (max-heap), To extract the root of the max heap we use DeleteMax() operation which shall take $O(log(n))$ time. Else if the number of elements in the right heap are greater than the number of elements in the left heap, then the median shall be the root of the right heap (min-heap), To extract the root of the max heap we use DeleteMin() operation which shall take $O(log(n))$ time.

   Case 2: If there are even number of elements, then the median is the average of root of the left heap (max-heap) and the root of the right heap(min-heap). To extract the root of the both the heaps we use DeleteMax() and DeleteMin() operations which shall take $O(log(n))$ time.

3. **Insert takes $O(log(n))$ time:**

   For every element 'e' we insert it based on the below criteria

   a. If the element 'e' is less than the root of the left max-heap, then insert it in the left heap (max-heap). Insertion into a heap takes $O(log(n))$ time.
   b. Else if the element 'e' is greater than the root of the left max-heap, then insert it in the right heap (min-heap). Insertion into a heap takes $O(log(n))$ time.

4. **Delete takes $O(log(n))$ time:**
   a. For deleting an element from a max heap, it takes a time complexity of $O(log(n))$
   b. For deleting an element from a min heap, it takes a time complexity of $O(\log(n))$

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:

Given that there are 'n' plants that are planted in a straight line. We can consider them as an array of plants.

Given that we need to place lamps in such a way that every plant receives ample amount of light from any one of the lamps that is placed within 4 meters.

Greedy Algorithm for the above problem shall be:

Step 1: Sort all the given plants from left to right on the given row.

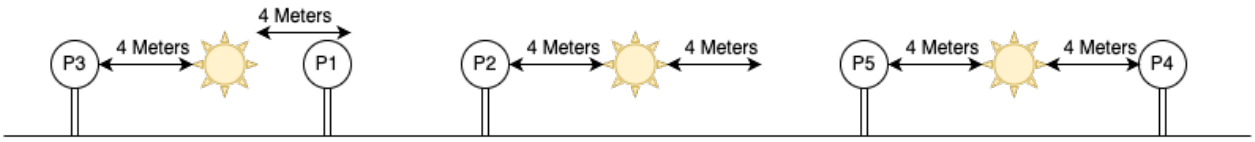
Step 2: From the sorted list of plants, Go to the first plant, then measure a length of 4 meters from the first plant and place a lamp.

Step 3: Mark all the plants that are with 4 meters on either side of the lamp as covered since they shall be receiving ample amount of light.

Step 4: Repeat Step 2 and Step 3 until all the plants in the list are covered and are receiving ample amount of light. (Go to the next uncovered plant, and measure 4 meters and then place a lamp)

Time Complexity of the above algorithm shall be $O(nlog(n))$, where 'n' is the number of plants.

Explanation with an example:



Step 1: Consider we are given plants P1, P2, P3, P4, P5. On sorting them from left to right on the given row (long line) as mentioned in the question. Assume we get the order as P3, P1, P2, P5, P4

Step 2: As per the algorithm, from the sorted list of plants, we go to the first plant P3 and them move 4 meters and place a lamp.

Step 3: Since plant P1 is within the 4 meters range of the lamp we mark it as covered.

Step 4: Then we again move to the next uncovered plant P2, and then measure 4 meters and then place another lamp.

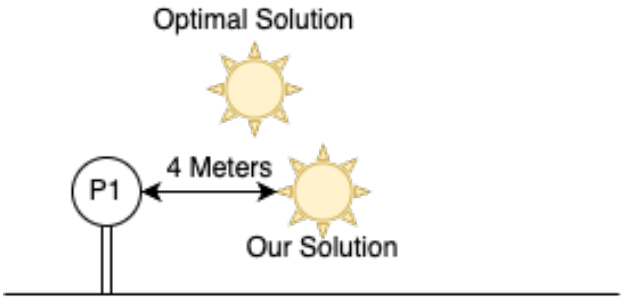
Step 5: We repeat Step 2 and Step 3, until we cover all the plants in the given row and each plant receives ample amount of light(each plant is with in the 4 meters range from a lamp).

Proving the correctness of the algorithm by mathematical induction:

Step 1: Base Case: Assuming that there is only one plant, and we must place a lamp so that it receives ample amount of light.

Since there is only one plant, as per our algorithm, we measure 4 meters from the plant and then place the lamp.

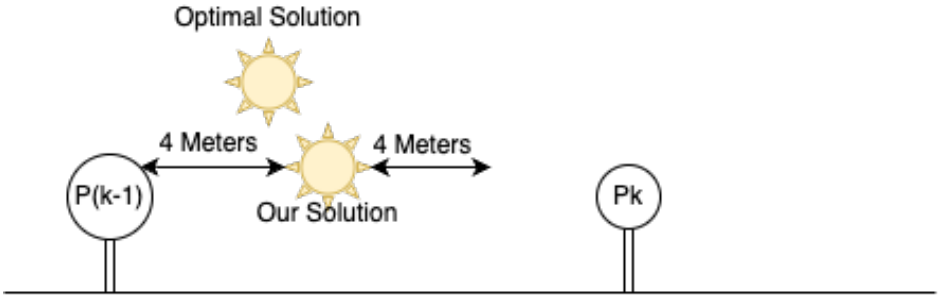
Even in the optimal solution, the lamp is placed in the same exact location or to the left of our lamp so that the plant will be with in 4 meters and will receive ample light.



Step 2: Induction hypothesis: Here in this step, we assume that for k-1 plants, both our algorithm and the optimal solution will work the same.

Step 3: Induction Step: In this step we need to prove that our solution will also work for 'k' plants.

As per Induction hypothesis, we have already assumed that our algorithm will be optimal for 'k-1' plants. Now we add a new plant to the right of the last lamp.

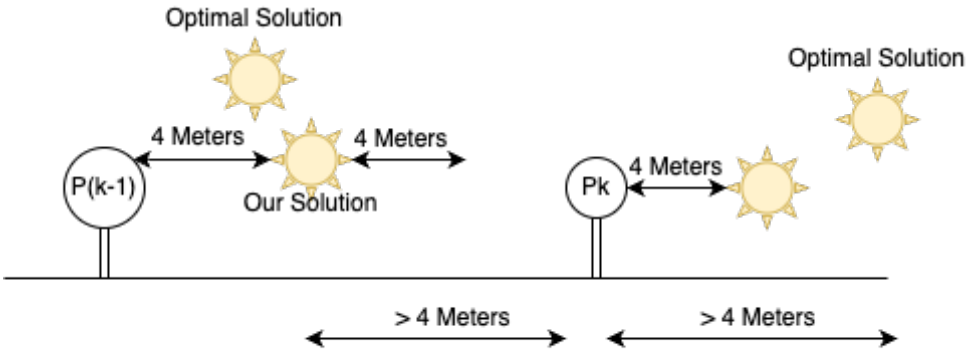


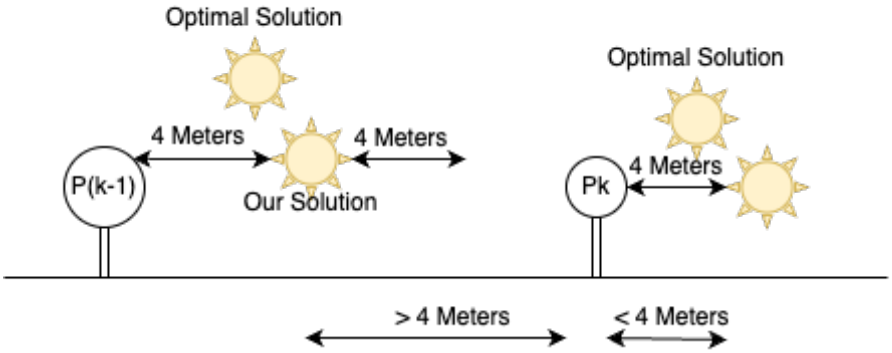
Assuming that for k-1 plants, we have a lamp placed 4 meters after the last plant in the sorted row and the optimal solution is to the left of our algorithm.

If the new plant P_k is not covered by the lamp placed by our algorithm(distance is more than 4 meters), then there is no chance that the new plant P_k will be covered by the lamp placed by optimal solution, because the distance is even greater.

So, from plant P_k, we need to measure a distance of 4 meters and place a new lamp as per our algorithm.

Here we assume that for the Plant P_k the lamp placed by the optimal solution is to the right of the lamp placed by our algorithm.





As per our algorithm, from the first uncovered plant, we measure a distance of 4 meters and then place a lamp. If the lamp placed by optimal solution is to the right of the lamp placed by our algorithm, the distance between the plant and the lamp placed by the optimal solution is more than 4 meters, which is impossible.

So, the lamp placed by the optimal solution should be either at the same location as the lamp placed by our algorithm or to the left of it for the plant to receive ample amount of light and for the solution to be optimal.

Prove by Contradiction:

Assume that our algorithm requires 'k' lamps to be placed to cover all the plants in the greenhouse, and the optimal solution requires 'm' lamps to cover all the plants in the greenhouse, which might be less than or equal to k.

Hence, for our solution to be equal to optimal or better than the optimal approach, we need to prove that $k <= m$

As per the proof of contradiction, we assume that $k > m$

After adding a new plant to the right of the previous lamp placed by our algorithm which is not reachable (the distance between the lamp and the plant P_k is greater than 4 meters).

So as per our algorithm, we need to add a new lamp, after measuring a distance of 4 meters from the new plant P_k.

Even from the lamp placed by the optimal solution the plant P_k will not be reachable (distance shall be greater than 4 meters). So, the optimal solution also adds a lamp within 4 meters of the new plant.

So, on adding a new plant, both the Optimal approach and our algorithm are required to add a new lamp.

Hence by proof of contradiction, we proved that the number of lamps required by our algorithm cannot be more than the number of lamps required by the optimal approach.

Hence, we proved both the Greedy Choice property by Contradiction and the Optimal Substructure by Mathematical induction and that our algorithm shall work for any number of houses as input.

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:

Given that there are ‘n’ plants in the greenhouse, which require l1, l2, l3…,ln amounts of water for them to be alive and the caretaker usually orders ‘n’ water bottles of the capacity equal to max(l1, l2, l3…ln) to water them.

But accidently he received water bottles of capacities c1, c2, c3…,cn and he must use one water bottle completely for each plant and should water the minimum amount or excess amount of water required for the plants.

Here, we need to determine an algorithm to check if it is possible to water all the plants in the greenhouse with either minimum or excess amounts of water and by using one full bottle for each plant.

Algorithm to check if we can come up with such arrangement or not is as follows:

Step 1: Insert the given inputs l1, l2, l3…ln (the minimum quantities of water required by a plant) into an array **‘l’** and sort it in ascending order.

Step 2: Insert the given inputs c1, c2, c3…cn (the minimum quantities of water required by a plant) into an array **‘c’** and sort it in ascending order.

Step 3: For each plant ‘i’ from the ‘n’ plants:

Step 3.1: if $c_i < l_i$, then return “We cannot get such an arrangement”, else PASS (move on to the next plant).

Step 4: Return “Such an arrangement is Possible”

Time complexity of the above algorithm shall be $O(nlog(n))$

Explanation:

As mentioned in the question, we need to water every plant with minimum or excess capacity of water and given a condition that we need to use an entire bottle for each plant.

So, after sorting both the arrays l1, l2, l3…ln and c1, c2, c3…cn in ascending order. For every l_i if the corresponding the c_i is greater than or equal to, it means that there is more water than the required minimum quantity and we can use it to water the plant ‘i’.

During this traversal, even for once, if the value of $c_i < l_i$, and since we must use an entire bottle for one plant, it means that there will be a plant which receives less than the minimum required quantity of water.

Proof of Correctness of the Algorithm:

Step 1: Base Case: Assume that we have one and only one plant, and it requires a minimum water quantity of $l1$, but we have received water quantity of $c1$.

As per our algorithm, if $c1 > l1$ then our algorithm outputs that "Arrangement such that, every plant receives minimum or excess water is possible" else "Such an arrangement, is not possible".

Step 2: Induction Hypothesis: We assume that the above statement holds for 'm' plants. i.e., if there are 'm' plants which require l1,l2,l3…lm minimum capacities of water for them to be alive, but we have received c1,c2,c3…cm capacities of water and our algorithm shall work.

Step 3: Induction Step: Here we need to prove that the above statement from the induction hypothesis holds 'true' even for 'm+1' number of plants.

Assume we have a new plant, P_{m+1}, which requires a minimum water capacity of l_{m+1} but we have received a water bottle of capacity c_{m+1}

In our algorithm, the first step we perform is to sort the water bottles received and the minimum required quantity of water for a plant to be alive in the ascending order.

Case 1: Let's assume that the capacity of the water bottle received c_{m+1} is less than l_{m+1}, then our algorithm returns that such an arrangement shall not be possible.

Here, as per mentioned in the question, we cannot use partial amount of water from a bottle to a plant and, we need to use only one water bottle received to water one plant.

There cannot be such an arrangement possible after sorting the water bottles received and the minimum required quantity of water for a plant to be alive in the ascending order.

Case 2: If $c_{m+1} >= l_{m+1,}$ in this case, our algorithm sorts the minimum water required and the capacities of water received in ascending order and since $\mathbf{c_i >= l_i}$ **(**$c_{m+1} >= l_{m+1,}$**)**.

We can come up with an arrangement such that every plant receives minimum quantity of water required or excess.

Hence, by mathematical induction, we proved that our algorithm can be used to determine if we can come with an arrangement to water all the plants with minimum or excess amount of water for any number of plants.

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 (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:

Since we must fill the water from the filter into minimum number of bottles, it is only possible when we select the bottles of higher capacity.

The Algorithm to fill the water from the filter into minimum number of bottles shall be:

Step 1: Sort the bottles based on their capacity in **descending** order.

Step 2: Consider the quantity of water in the filter to be 'W'

Step 3: Initialize a 'counter' variable to 0

Step 4: For each bottle $\mathbf{p_i}$ in the descending sorted list:

Step 4.1: If water in the filter is not empty (W>0) and water in filter is greater than the bottle capacity (W > $\mathbf{p_i}$)

Step 4.1.1: Fill the bottle $\mathbf{p_i}$

Step 4.1.2: Now, water in the filter shall be W = W - $\mathbf{p_i}$

Step 4.1.3: $counter = counter + 1$

Step 5: The value of counter variable returned will be the fewest number of bottles that are filled with water from the filter.

Time complexity of the above algorithm shall be $\boldsymbol{O(nlog(n))}$

Proof of correctness of the above algorithm:

Every greedy algorithm has two parts, Greedy Choice property and the Optimal Substructure.

To prove the correctness of the greedy algorithms, we need to Prove the Optimal Substructure using Induction and the Greedy Choice property by using Contradiction.

Let's assume that our algorithm**(represented as ALG further in the explanation)** above selects the water bottles $A_1, A_2, A_3 \ldots A_k$ and let's assume that the optimal approach **(represented as OPT further in the explanation)** selects the water bottles $O_1, O_2, O_3 \ldots O_m$.

Proving by Induction on number of bottles:

Step 1: Base Case: Here in this step, we need to prove that the algorithm is working when we have one bottle

Let us assume that the capacity of a water bottle that is being used by our algorithm or the optimal approach is defined by the function: **C(water bottle)**.

If there is one and only one bottle, in which we need to fill the water from the filter, then our algorithm will sort the bottles in descending order (since we have only one bottle, the sorting has no impact) and then select/pick the only bottle.

Similarly, since there is only one bottle to pick and fill the water from the filter, even the optimal approach will pick the same bottle.

$C(A_1) = C(O_1)$

Hence, base case holds true.

Step 2: Induction hypothesis: Here in this step, we assume that $\sum_{i=1}^{k} A_i$ **>= W and** $\sum_{i=1}^{m} O_i$ **>= W**

And **W <=** $\sum_{i=1}^{k} A_i$ **<=** $\sum_{i=1}^{m} O_i$ by combining the above two equations.

Step 3: Induction Step: In this step, we need to prove that the above statement, holds true even after adding a new bottle.

$$\sum_{i=1}^{k} A_i \text{ <= } \sum_{i=1}^{m} O_i$$

Assume we are adding a new bottle of the same capacity and both our algorithm, and the optimal approach shall check and consume/use it if it satisfies the required condition.

If the bottle is of a higher capacity, then our algorithm shall pick it since we are sorting the water bottles in descending order.

Case 1: If the bottle is considered by our algorithm, then we can say that the number of bottles used by our algorithm shall be reduced or be the same as the previous number of bottles. But still the capacity of all the bottles filled shall remain the same.

$$\sum_{i=1}^{k} A_i \text{ + new bottle <= } \sum_{i=1}^{m} O_i \text{ + new bottle}$$

Since the bottle that add, shall be of the same capacity.

$$\sum_{i=1}^{k} A_i + A_{k+1} <= \sum_{i=1}^{m} O_i + O_{m+1}$$

So even after adding this new bottle, there is no change in the inequality.

Case 2: If the bottle we are adding is of smaller capacity than the bottles we used in our algorithm, then we don't consider that bottle. But still the total capacity of the water bottles used by our algorithm does not change.

$$\sum_{i=1}^{k} A_i \text{ + new bottle <= } \sum_{i=1}^{m} O_i \text{ + new bottle}$$

So even after adding this new bottle, there is no change in the inequality.

Even though we add a new bottle, our algorithm sorts the bottles in the descending order and chooses the bottle.

Proof by Contradiction:

Here we need to prove that the number of bottles used by our solution is less than or equal to the number of bottles used by the optimal solution.

i.e., $k <= m$.

As per the method of proving by contradiction, we consider $k > m$

In our algorithm, we are considering that, we sort the empty water bottles by capacity in descending order and then fill them with the water in the filter.

As we sort the water bottles in descending order to fill the water from the filter, we use a smaller number of water bottles or equal number of water bottles when comparing with the Optimal solution.

This is contradicting our statement $k > m$.

Hence, $k <= m$. By this we prove that our algorithm works on any number of input/empty water bottles, and it also uses less or equal number of water bottles when compared to the optimal solution.