Homework 1

Pratyay Dutta (Codeforces ID : pdutt005)

January 20, 2024

Contents

1 A complex complexity problem

1. $sqrt(3)^(log(n)) = o(n)$
2. $log(log(n)) = o(sqrt(log(n))$
3. $log(n!) = o(nlog(n)$
4. $2^n = o(3^n)$

The proofs are given below:

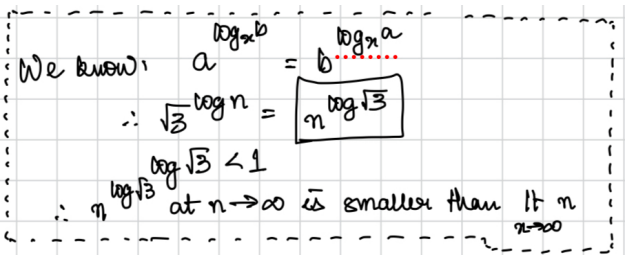


Figure 1: Proof of 1

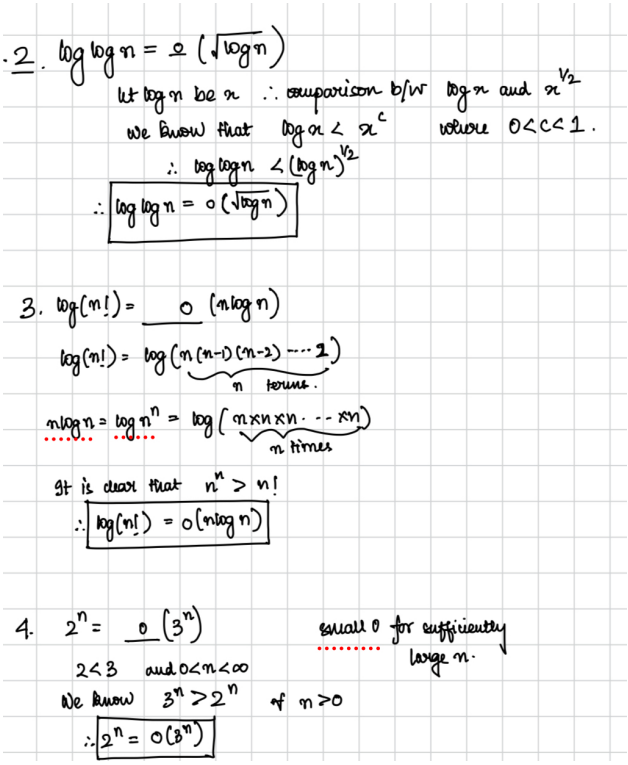


Figure 2: Proof of 2,3 and 4

2 Solve recurrences

Referring from professor's lecture and notes: The Master Theorem is as follows:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) = O(n^c) \text{ for } c < \log_b a \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{if } f(n) = \Omega(n^c) \text{ for } c > \log_b a \text{ and } af(n/b) \leq kf(n) \text{ for some constant } k < 1 \end{cases}$$

1. $T(n) = T(n/2) + nlog(n)$
   $T(n) = \Theta(nlog(n))$
2. $T(n) = 2T(n/4) + n^(1/2)$
   $T(n) = \Theta(n^(1/2)log(n))$
3. $T(n) = 4T(n/4) + n^(1/2)$
   $T(n) = \Theta(n)$

The working and proofs are given below:

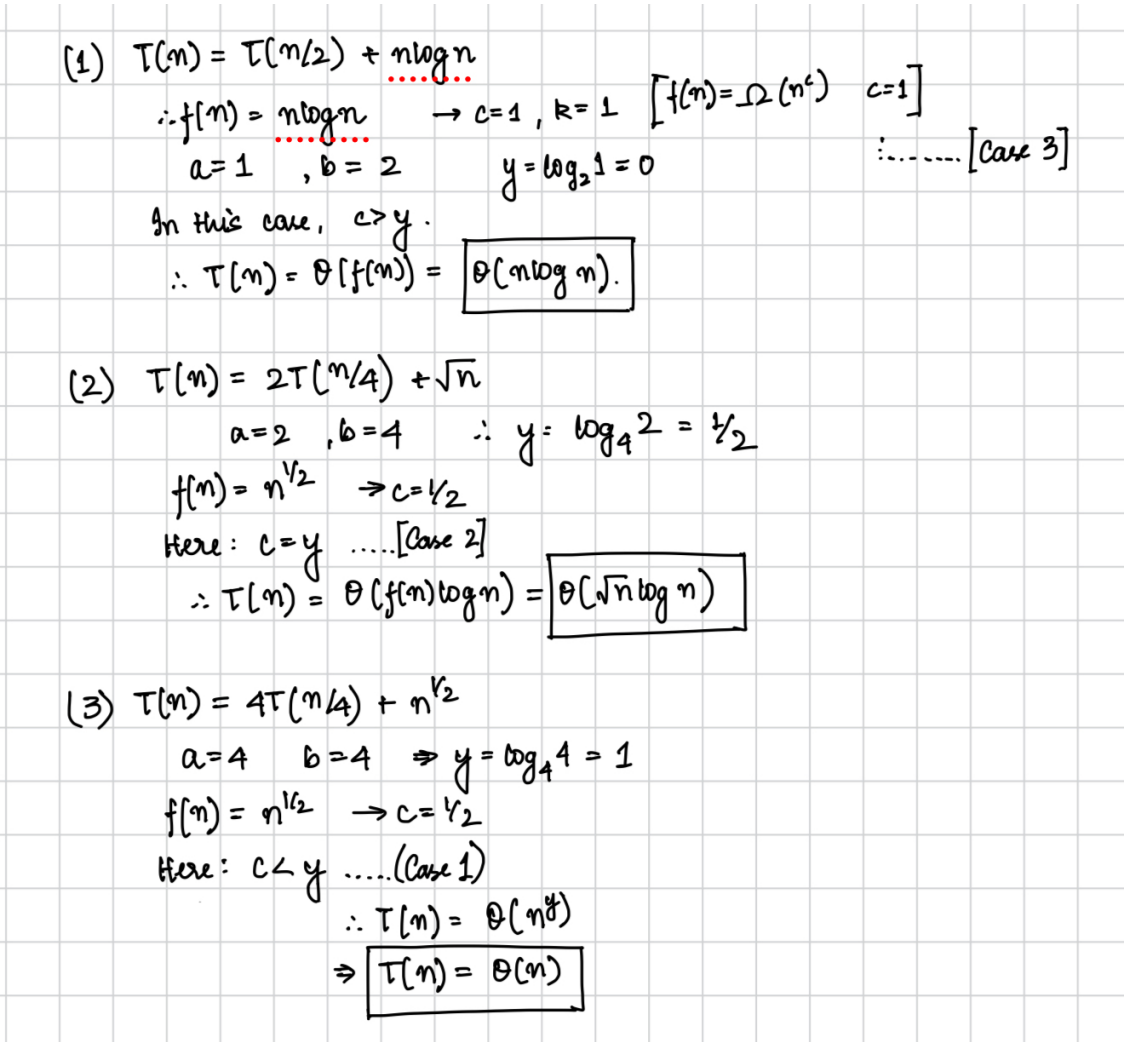


Figure 3: Proof of 1,2 and 3

3 Test the candies

3.1 One bad candy

There is only 1 bad candy. Bad candy has lighter weight. Each weight comparison = 1 dollar.
We use recursion and divide and conquer to solve this problem. We divide the candies in half and compare the two equal numbered parts. If the weight of both halves are equal then the number of candies must have been odd and the bad candy is the one which is left after dividing equally into two halves. If one pile weighs less, then the bad candy is in that pile and we divide that pile and we keep doing this recursively till we reach node level.
For base case, we check if the pile to be divided has 1 candy. If we reach that point then that candy is the bad one.
Explanation: We find the bad candy at the node level. The number of divisions to get to node level = $log_2(n)$ since we are dividing the pile by two everytime. The cost for each time we weigh = 1 dollar. Number of weighs per division = 1. Therefore, total cost = $log_2(n)$
Algorithm:

```
Find (list):
    if len(list) == 1, return list[0]
    mid = len(list)// 2
    left= list[ : mid]
    right= list[mid : 2*mid]
    extra= list[-1]
    if weight(left) == weight(right):
        return extra  //only when odd candies//
    elif weight(left) < weight(right):
        Find(left)
    else:
        Find(right)
```

3.2 One bad candy improvement

We divide the list into three parts. We weigh the first two equal divisions.

- If the weight is equal, then the bad candy is in the last division. We divide that pile recursively.
- If the weights are unequal, then the bad candy is in the lighter pile. We divide that pile recursively.

When a case arises such that the list to be divided is not a multiple of 3. We then make the divisions such that the first 2 piles have equal candies and the third partition has as many more as the remainder we get after dividing the pile by 3. For each time we divide, we make one comparison and spend 1 dollar each time. The number of divisions to get to node level from n candies in this case is $log_3(n)$. Therefore, total cost = $[log_3(n)]$ dollars.
The algorithm is given below:

```
Algo :
    Find3 (list) :
        if len(list) == 1 , return list[0]
        else :
            div = len(list) // 3
            first = list[:div]
            second = list[div:2*div]
            third = list[2*div:]
            if weight(first) == weight(second) :
                    Find3 (third)
            elif weight(first) < weight(second):
                    Find3 (first)
            else :
                    Find3(second).
```

Figure 4: Algorithm

3.3 Prove lower bound

The proof of the lower bound is given below:

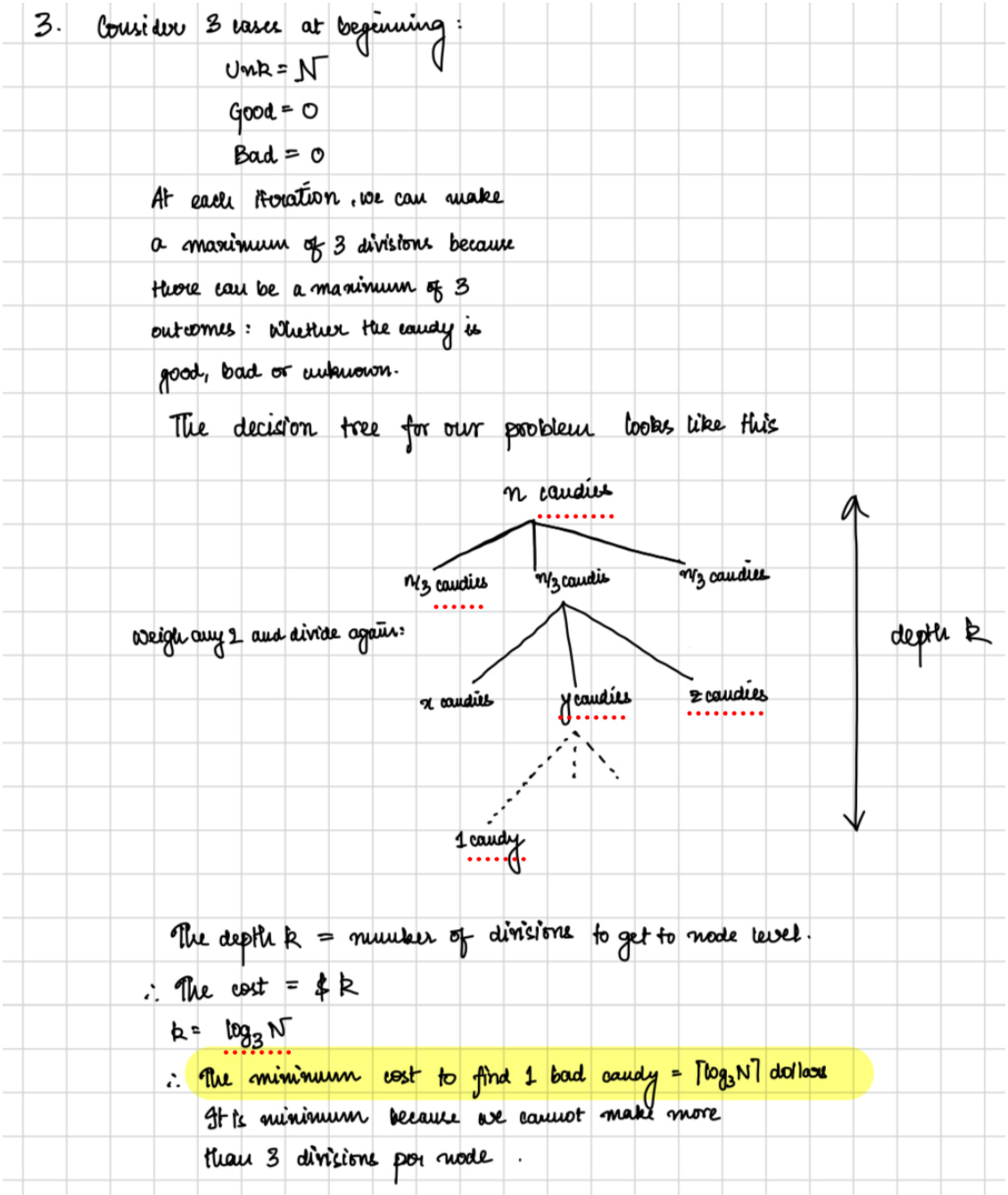
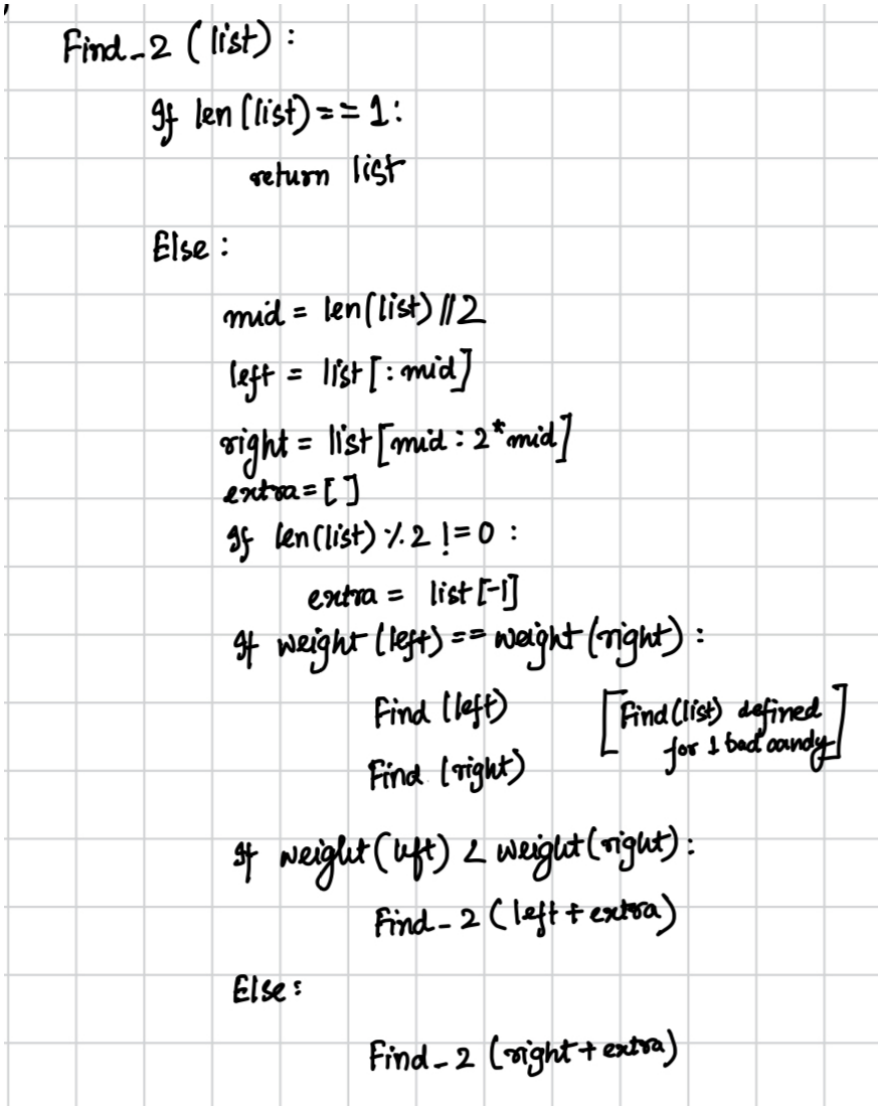


Figure 5: Proof of lower bound

3.4 Two bad candies

We divide the samples into 2 groups. We weigh the two piles.
If they weigh the same, then there is 1 bad candy in each. In this case we divide both.
If either of them weigh less, then at least 1 bad candy is in that pile. In this case, we divide only the group with lesser weight.
In worst case, we divide both piles once and then divide 1 group out of the two all the other times because after dividing twice, there is only one bad candy in each pile. For these, we use the algorithm we used for finding one bad candy. Number of divisions we have to make to get to node level = $log(n)$. Number of weighings per division = 1 [except the one case once where we divide both] Therefore, cost to find both candies = $[log(n)]$
Algorithm:



3.5 k bad candies

Let k = 5 To find 5 bad candies, we want to find only 1 bad candy first.
We divide the pile into two parts. We weigh the two piles.
If the piles weigh the same then there must be 2 bad candies in each pile. We divide either of the piles. Till we reach base case.
If one pile weighs less, then that pile must have at least 3 bad candies. We divide that pile till we hit base case. We find one bad candy for sure in this node since we divide only the piles which have the bad candies. We continue doing this 4 times till we find 4 bad candies while the problem still is a multi bad candy problem. Each iteration has complexity O(log n). So total time complexity for the 4 iterations is O(4log n)
Once we find 4 bad candies, there remains only 1 bad candy in the pile. We use the method that we discussed for finding a single bad candy now.
The time complexity to find the last candy is O(log n).
Therefore total time complexity = O(5log n) = O(log n) Generalising : For k bad candies, we find 1 bad candy k-1 times with the given algorithm and we find the last candy with the algorithm that we used in question 1 of this problem. Total time complexity = O(log n)
Algorithm:

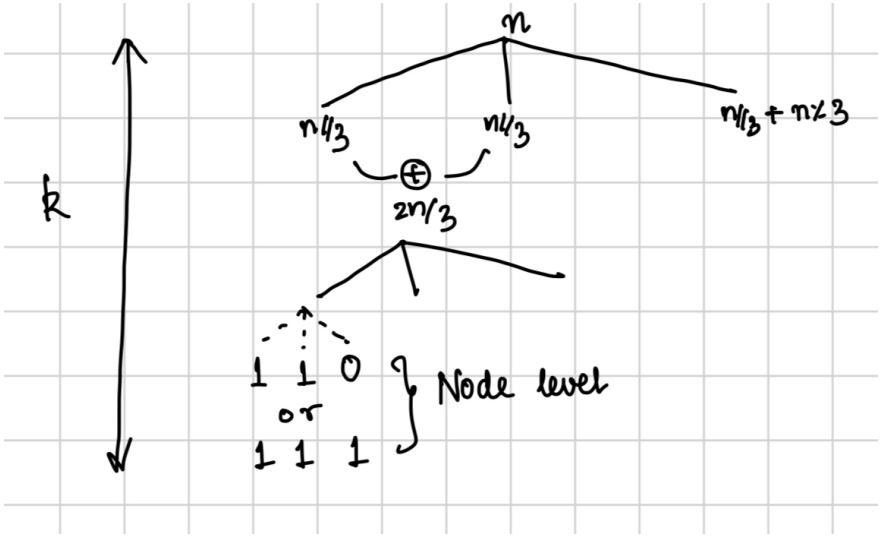
```
Find_k(list):
    if len(list) == 1:
        return (list)
    mid = len(list)//2
    left = list[:mid]
    right = list[mid:2*mid]
    extra = list[-1]
    if weight(left) <= weight(right):
        Find_k(left)        # Dividing left pile in case of equality as well !!
    Else:
        Find_k(right)

find_list = [ ]
while len(find_list) < k:
    find_list.append(Find_k(pile without the found bad candies))
find_list.append(Find(pile without the found bad candies))
```

3.6 Lighter or heavier

We divide the pile into 3 groups G1, G2,G3 such that G1 and G2 has n//3 candies and G3 has n//3 + n%3 candies.
We weigh G1 and G2. If they weigh the same, then G1 and G2 has good candies. We keep one good candy(Cg) aside for future reference. Bad candy must be in G3. We keep dividing this part recursively till we reach <= 3 candies. If G1 and G2 does not weigh the same then the bad candy must be in either of the two groups. We take all candies in G1 and G2 and divide them into 3 parts recursively till we hit <= 3 candies.
Once, we hit <= 3 candies:

- We weigh first and second candies, if they weigh the same then the 3rd candy is the bad candy,
- If they weigh different, then bad candy must be either of the two. We take the one weighing less and weigh it against Cg.
- If they weigh the same then the other candy is the bad candy.
- If the lighter candy is lighter than Cg, then this candy is the bad candy and it is lighter.

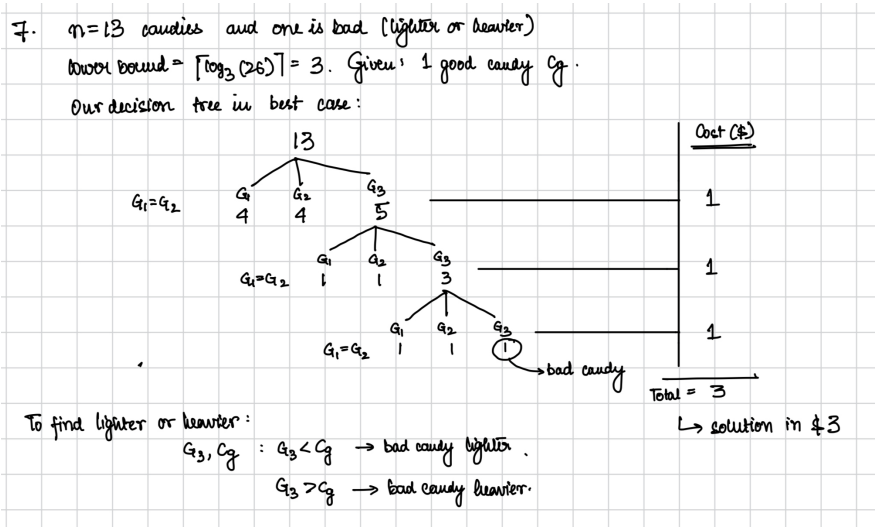
Assuming worst case, we divide 2n/3 candies into 3 parts each time. The decision tree for the problem seems like this :



Number of divisions to reach node level = depth of tree k = $log_3(2n/3)$
Number of comparisons at each division = 1
For base case, we need one more comparison to determine bad candy.
Therefore, total comparisons = total dollars = $log_3(2n/3) + 1 = log_3(2n) - log_3(3) + 1 = log_3(2n)$
This is the minimum cost as well because we can make at most 3 divisions at each point to ensure flow of information. Therefore we need at least $[log_3(2n)]$ dollars to find the bad candy and if it is heavier or lighter.

3.7 Find the one bad candy in 3 dollars

The procedure of finding the one bad candy in 3 dollars with our proposed algorithm is as follows:



4 Finding the minimum value

4.1 Explanation

We know that a function has a minimum value at the point where the derivative of the function is zero if the second derivative at that point is positive. We find out by graphing the function, that the given function has a local minima between 1 and 2. Therefore, the derivative of the function in the range(1,2) is a monotonically increasing function. Therefore, to find the minima of the function, we have to find the root of its derivative. We implement a binary search technique to find the root of the derivative. We first take the two endpoints as starting points. We see if the root exists at either of the points. If not then we start our search. Iteratively, we take the midpoint of the two points and find the value of the derivative at that point.

- If the derivative is less than zero then the root must lie between this point and the point on the right.
- If the derivative is more than zero, then the root must lie between this point and the point on the left.

We update the right and left points accordingly and repeat the process till we find a midpoint which is arbitrarily close to 0 (correct to 10^-6).
Time Complexity : We are dividing our entire sample space in half each time we take the midpoint and search between that and the boundaries. Axiomatically our time complexity is O(log n).
Submission ID : 241261102

4.2 Algorithm

Algorithm 1 Find Minima of a Function

Data: Value of parameter a
Result: Minima of the function at given a

```
// Initializing the two end points 1 and 2  left, right ← 1, 2  max_steps ← 100
// If the root of the derivative is at 2, then return the value of the function at 2  if dx(right, a) = 0 then
    return f(left, a)
// Iterate to find the root of the derivative  for _ in range(max_steps) do
    mid ← (left+right)/2 // Get midpoint
    // If the value at midpoint is less than the threshold, then return the value of the function at that point
    if dx(mid, a) ≤ threshold then
        return f(mid, a)
    // If the value at midpoint is less than zero, then we have to search between that point and the point on
     the right  else if dx(mid, a) < 0 then
        left ← mid
    // If the value at midpoint is more than zero, then we have to search between that point and the point
     on the left  else
        right ← mid
return f(mid, a)
```

5 Sort the train

5.1 Explanation

I first used the brute force method(bubble sort) which had two for loops and operated in n^2 time. For getting $nlogn$ time, i used a divide and conquer method which is a slight modification of the mergesort process. In practise, the sorting happens according to the mergesort algorithm. We keep on taking halves of the array till we reach a point where there is only one element in the array. We then merge the arrays while sorting it. We move through each element in the left and right subarrays and add the smaller element to the new array first. This is the classic mergesort algorithm. I took help from the CLRS book[1] to brush up the algorithm for mergesort. I read it and closed the book and coded it on my own. To count the number of swaps needed we maintain a counter. During the merging process, if an element of the right subarray is smaller than an element in the left subarray, then it is smaller than all the other successive items of the left subarray because the left array is sorted. So the number of swaps required to put that element in the right position will be $len(leftsubarray) - index_(element)$. To demonstrate this with an example:
left subarray = [2,4,6,8,10] right subarray = [1,3,5,7,9] when we are considering the item 6 and item 5: item 5 is smaller than item 6,8,10. So the number of swaps to put 5 in the right position would be $len(leftsubarray) - index_(element) = 5 - index(5) = 5 - 2 = 3$ which is true because we have to swap it with 10 then 8 and then 6. Hence, we update the swap counter by this number = $len(leftsubarray) - index_(element)$

Submission ID : 241533864

5.2 Algorithm

Algorithm 2 Train Sort Algorithm

```
procedure TrainSort(arr)
    swaps ← 0
// If length of the array is less than one, then it is already sorted. Do the following if the length!=1
if len(arr) > 1 then
    // finding the midpoint for divide and conquer
    mid ← ⌊len(arr)/2⌋
    L ← arr[: mid]
    R ← arr[mid :]
    swaps ← swaps + TrainSort(L)                    ▷ Sort the first half
    swaps ← swaps + TrainSort(R)                    ▷ Sort the second half
    i ← 0
    j ← 0
    k ← 0
    while i < len(L) and j < len(R) do
        if L[i] ≤ R[j] then
            arr[k] ← L[i]
            i ← i + 1
        else
            arr[k] ← R[j]
            j ← j + 1
        // If an element on the right array is smaller than an element on the left array, then it must
        also be smaller than all the successive elements of the left array since the left array is sorted in
        ascending order.
            swaps ← swaps + (mid − i)
        k ← k + 1
    // Check if any element is left in either array
    while i < len(L) do
        arr[k] ← L[i]
        i ← i + 1
        k ← k + 1
    while j < len(R) do
        arr[k] ← R[j]
        j ← j + 1
        k ← k + 1
    return swaps
end procedure
```

5.3 Proofs

5.3.1 Optimality check

5.3.2 Algorithm and complexity

The algorithm and the explanation of the process is written in the section 5.1
Time complexity : The number of times we divide to get to node level = log n. The time complexity for the divisions = O(log n)
The time complexity of the merging process while counting the swaps = O(n) because there is one while loop iterating through all the items in the lists.
Therefore, total time complexity = O(nlogn)

5.3.3 Proof of lower bound

6 Being Unique

We have to find if there is at least one item A[k] in the range(i,j) such that $i <= k <= j$. The naive approach that occurred to me first was to check every range possible on the list. This would have required two for loops to iterate through all the possible ranges and the time complexity would have been O(n^2).
I wanted to divide and conquer this problem and I realised that if i divide the list by 2 every time till we reach the node level, then while combining the two lists, we can find all the possible ranges (i,j) in the original list. We can then find out if each range has the unique in range property.
Hence, we first find the midpoint of the list and divide it equally into two parts. We keep dividing recursively till we hit node level. From the node, we merge the two lists and work our way back up.
During merging, we first concatenate the two lists. We maintain a dictionary that contains the frequency of the elements in the merged list. After we create the freuquency dictionary, we see its values. If we find that none of the frequencies are one, then there are no unique elements in the given range and we can safely break and return that the list does not have the unique in range property and we return **False**. If at least one of the elements have frequency=1, then that is the unique element and there is still a possibility that the entire list may have the unique in range property. When we have finished checking for the entire list, and if we find at least one unique element in each of the intervals we saw, this means that the full list has the unique in range property and we return **True**.
The python code that I have written for the implementation is as follows:

```
def unique(l):
    if len(l)>1:
        #Divide into two parts
        mid = len(l)//2
        left = l[:mid]
        right = l[mid:]
        unique(left)
        unique(right)

        #Concatenating the two parts
        merged = left+right

        #The dictionary containing the frequency of each element
        freq = {i:0 for i in merged}

        #Creating the dictionary
        for i in merged:
            freq[i]+=1

        #Checking if any of the frequencies are one
        if 1 not in freq.values():
            return False

    return True
```

Time complexity: Time complexity for dividing till node level : O(logn)
Time complexity of making the frequency dictionary = O(n)
Time complexity of checking the frequencies = O(n)
Total time complexity of operations at each division = O(2n) = O(n)
Total time complexity to check for the entire list = O(nlogn)

References

[1] Cormen, Leiserson, Rivest, and Stein Introduction to algorithms (CLRS). Third Edition Section 16.4, Lemma 16.7

[2] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1), 1959.