Cost-Benefit Analysis Design Document

By: Katimu Lissa

Table of Contents

1. Introduction

During the process of studying the pattern of passengers on a bus route, the average amount of passengers that came on and off the bus were displayed through positive and negative integers. Positive integers would specify an increase in how many passengers got on, while negative integers would specify the decrease. With this in mind the problem now is to find the highest positive change in the averaged number of passengers that frequented the bus route. The inputs for the problem will be the array of positive and negative integers that indicate averaged number of passengers for each stop. The expected output will be the value of the highest change going toward a positive integer. Meaning we are trying to find the highest sum of each averaged value that contributes to going straight from a lower average integer to a higher positive average integer. In order to come up with an effective algorithm for this problem I will first discuss how this problem can be solved using the Merge Sort algorithm. While looking over the Merge Sort algorithm I will also discuss why Merge Sort is a better approach than Binary Search and then showcase the runtime complexity it should produce. Next, I will go over how this problem can also be solved iteratively and what the best runtime complexity would be for that approach. Then lastly, I will complete a test-driven development where already given and new ideas for interesting test cases will be outlined. Through this I will display why each of these test cases exist and how I can possibly make them better through the added interesting test cases.

2. Divide and Conquer

2.1. MergeSort Based

This problem would be better fixed using the Merge Sort approach because when using Binary Search, the purpose is to narrow in on part of a list until the element you want is found. When doing this approach, the list would be restricted to only looking at one half and discarding the other parts. If Merge Sort were to be used, then all the elements would be kept in the array so that the value of finding the crossing point and the max could be acquired. If the bus stops were not back-to-back, divide and conquer would not be an option and we would not be able to use Merge Sort. This is because being back-to-back entails that the amount of people that come off of the bus stop are a part of the people that were already on the last bus and any new passengers are just added on to the amount. Therefore, when trying to implement Merge Sort this issue would not accurately give the highest passenger density for a route. If the sorting step were to be implemented into this algorithm, then the highest passenger densities for each halves of the list would then be randomized by sorting the densities instead of finding the cross section and the max value.

2.2. Initial Algorithm

Pseudo-Code Outline of Algorithm:

```
findHighestPassengerDensity(A){
        if (length(A) = 0) return -1
        if (length(A) = 1) return A[1]
        mid = (A[0] + length(A)) / 2
        B = A[1 … mid]
        C = A[mid + 1 … n]
        rB = findHighestPassengerDensity(B)
        rC = findHighestPassengerDensity(C)
        rA = findCrossPassengerDensity(B, C)
        return (max (rA + rB + rC))
}
findCrossPassengerDensity(B, C){
        if(length(A) = 1) return A[1]
        L = findHighestPassengerDensity(B)
        R = findHighestPassengerDensity(C)
        return (L + R)
}
```

Runtime and Correctness:

The big-O for findHighestPassengerDensity() is O(nlogn). This can be justified through the Master Theorem where the number of sub-problems: a = 2, the input divided by: b = 2, and work done on each call: d = 1. Where $2 = 2^1$, therefore by case 1 the big O will be $O(n^1 \log n)$. I believe this algorithm will be correct due to the process of merge sorting. The algorithm should have a base case, be able to divide the problem into two halves, recursively call the method, and then merge or in this case find the max of the sum of the three values.

Iterative

2.3. Solving the problem with a single loop

If I were to linearly scan for the highest passenger density at each bus stop then the algorithm would no longer be O(n) due to the added loop. If the added loop to find the highest passenger density was added, then the algorithm would turn into an $O(n^2)$ complexity. By using a simple calculation, we can use one loop to go through the array while storing the highest passenger density into a variable. If the highest passenger density at i comes out negative when adding current stops value, then that would mean that more people on the bus came off than there were as many people that initially came on. Therefore, when the bigger number of people coming off is added to the smaller number of people initially on, they will produce a negative outcome. This would make the highest passenger density not accurate since the value of the highest passenger density is supposed to measure the highest positive change. Both the highest passenger density ending at i and the highest passenger density discovered are important because one checks for the highest density from the previous stop to the current stop, while the other checks the highest density from all of the stops that have been gone through so far. If only the highest passenger density ending at i were to be checked then the loop would only be looking for the current highest passenger density pertaining to the two current values. By checking also for the highest passenger density discovered so far it takes into consideration the highest passenger density's that were found towards the beginning of the array.

2.4. Initial Algorithm

Pseudo-Code Outline of Algorithm:

```
findHighestPassengerDensity(A){
        if(A.length() == 0) return -1
        if(A.length() == 1) return A[0]
        int hpd_bustop = 0, hpd_discovered = 0
        for loop from i to A.length(){
                if(hpd_bustop > = 0 && A[i+1] > A[i]) {
                        hpd_bustop at i = hpd_bustop at i-1 + A[i]
                }
                if(hpd_busstop  > hpd_discovered){ hpd_discovered = hpd_bustop }
        }
        return hpd_discovered
}
```

Runtime:

The big-O for findHighestPassengerDensity() is O(n). There is a problem of size n, making the loop go through each element n times and putting the desired value into hpd_bustop. I believe the base cases and the way to find the highest passenger density discovered is correct, but I think the way of finding the highest passenger density at this bus stop could be adjusted depending on the tested results. I would have to see how to make the highest passenger density at this bus stop equal the previous highest passenger density + the current highest passenger density.

3. Test-Driven Development

3.1. The starting Test Suite

Test0 was created to make sure that both of the algorithms are able to satisfy the base case where the algorithm is able to return the value if there is only one element in the array. Test1 tests how the algorithm would react when the base case is implemented but the element is a negative number. Since there is still only one number being given there would be no change and the value should be returned. Test2 shows that both algorithms should find the highest passenger density when given any two elements. This is needed to test that the algorithm can execute the main part of the function outside of the base case properly. Both Test3 and Test4 are created to make sure that both algorithms can run when they are given multiple different values. This makes sure that the algorithm can execute when given any number of elements greater than two.

3.2. Building a Better Test Suite

The first interesting test case would be checking if the algorithm is able to execute correctly when all of the numbers are negative. If all of the numbers were negative the Merge Sort algorithm would return zero, but the iterative code would have to account to check if the loop makes a negative highest passenger density appear. That's why this case value is needed to make sure that the case of having a negative highest passenger density ending at the current bus stop gets taken care of. The second interesting test case would be keeping all of the elements the same number. The purpose of this test is to check if the created algorithm is able to still accurately find all of the highest passenger densities when finding the cross-passenger density method is implemented. The third interesting test case is checking the scenario of if all of the highest passenger densities came out to be the same value even though there are different densities in the array. This test will react similarly to if all of the numbers were the same value, but the test would make sure that the Merge Sort algorithm is working right in finding the cross passenger and when adding the numbers up to find the max. The last interesting test case is if the elements alternated from increasing to decreasing each time in the array. This test case would be good for testing the iterative portion to see if the conditions that are made on the loop are able to accurately only choose the highest passenger densities that are positive.