CS 3510 Section B: Design & Analysis of Algorithms August 28th 2024

Problem Set 2: Divide & Conquer and Arithmetic

YOUR NAME HERE Due: September 6th 2024

- Please type your solutions using LaTeX or any other software. Handwritten solutions will not be accepted.
- Your algorithms must be in plain English & mathematical expressions, and the pseudo-code is optional. Pseudo-code, without sufficient explanation, will receive no credit.
- Unless otherwise stated, all logarithms are to base two.
- If we ask for a specific running time, a correct solution achieving it will receive full credit even if a faster solution exists.

Problem 1: An Upward Trajectory

(25 points)

You have been working hard to improve your grades recently, and are curious to determine whether you are succeeding or not. So you decide to take a list of your assignment grades in chronological order, and determine if they are showing an upward trajectory.

You want to find the number of times when, on any assignment, you exceeded your grade on a previous assignment. For example, if your grades were [82, 100, 70, 98, 100], you would output 6. Specifically, when you received a 100% on the 2nd assignment (1-indexed), that exceeded the 82% you received on the 1st assignment; when you received a 98% on the 4th assignment, that also exceeded the 82% you received on the 1st assignment plus the 70% on the 3rd assignment; and when you received a 100% on the last assignment, that exceeded your grades on the 1st, 3rd, and 4th assignment, creating a total of 6 instances of performance improvements.

Your goal is to design a divide-and-conquer algorithm for this problem that runs in time $\mathcal{O}(n \log n)$. Remember to justify your algorithm's correctness, show your recurrence, and prove runtime using the Master Theorem.

Hint: Is there a step in the traditional merge sort algorithm where we might be able to easily get this information?

Solution:

- *Algorithm:*

```
counter = 0 ##global variable

def performanceChecker (A):
    counter = 0
    performanceCheckerHelper (A)
    return counter

def performanceCheckerHelper (A) :
    if n == 1: return A
    L = performanceCheckerHelper ( A [1.. floor ( n /2)]
    R = performanceCheckerHelper ( A [ floor ( n /2) +1.. n]
    return mergePlus (L , R)

def mergePlus ( x [1... k ] , y [1... m ]) :
    if x is length 0
    return y
    if y is length 0
    return x
    if x [1] <= y [1]:
     counter + len[y]
```

```
return x [1]. join ( mergePlus ( x [2... k ] , y [1... k ]) )
else :
return y [1]. join ( mergePlus ( x [1... k ] , y [2... k ]) )
```

- *Correctness:* I decided to attempt the algorithm by implementing a traditional merge sort algorithm and count the number of times an element in the right subarray is greater than the first element in the left subarray during the merging process, and afterwards merging the first element in the left subarray before going onto the next. This way, the merged array would have the largest value in order for the next merge, and since we know all elements in the right array goes in the order of increasing, if the L[1] is less than R[1], that means its less than all right array elements, thus we increase the count accordingly.
- *Time Complexity Analysis:*
- $T(n) = aT(n/b) + n^d$
- $T(n) = 2T(n/2) + O(n^1)$
- $T(n) = O(nlog(n))$

Problem 2: Emptying Tanks

(25 points)

Aqua has been assigned the task of emptying n water tanks before the supervisors return in h hours. The i-th tank contains $tanks[i]$ liters of water, where $tanks$ is an n-sized array.

She can choose how fast to drain the water, with a fixed rate of r liters per hour. Each hour, Aqua picks one tank and drains up to r liters from it. If the tank has less than r liters left, she drains the remaining water and does not drain any more that hour.

Aqua prefers to take her time but must ensure that all the tanks are empty before the supervisors return.

Using a divide and conquer approach, find the minimum rate r that Aqua needs to set in order to empty all the tanks within the given h hours. Justify your algorithm's correctness and runtime.

Solution:

- *Algorithm:*

```
def emptyTank (h, tanks):
    def canEmptyAllTanks(rate):
        total_hours = 0
        for i in tanks:
            total_hours += (i + rate - 1) / rate
        return total_hours <= h

    left = 1
    right = max(tanks) ##O(n)
    while left < right:
        mid = (left + right) / 2
        if canEmptyAllTanks(mid):
            right = mid
        else:
            left = mid + 1

    return left
```

- *Correctness:* This method is a divide and conquer algorithm that does it similarly to a binary search function. It first finds the mean value of the amount of water and tests its hypothesis with the canEmptyAllTanks method. If it works its going to try again but setting the rate to the mean between the current rate and lowest rate that works, if it doesn't work it's going to find a rate that is the mean of the current rate and the amount of water in the largest tank. It will repeat the process until it finds the smallest rate that works according to the canEmptyAllTanks method. Which will

return the final rate.

- *Time Complexity Analysis:* max(tanks) is O(n), canEmptyAllTanks is also O(n). Since the Tree will have O(logn) levels due to its similarity with binary search
- T(n) = O(n) + O(nlogn)
- T(n) = O(nlogn)

Problem 3: Maximum Bananas

(25 points)

A monkey currently has b rotten bananas and seeks to get rid of as much bananas as possible. There are n portals laid out in order and each portal has a multiplier that can be applied on the current amount of bananas the monkey has. For example, going through a portal with a multiplier of 0.5 will halve the amount of rotten bananas the monkey has. These multipliers are represented as a 1-indexed array of non negative numbers A of length n. The monkey is allowed to start at some portal $1 \leq i \leq n$, applying multipliers to his bananas up to some index $j \geq i$ *without* skipping indices. For example, given the array $A = [3, 0.1, 0.2, 100, 0.75, 4, 0.1]$ and $b = 100$ bananas, the monkey's best course of action would be to start at position 2 and stop at position 3. The maximum number of bananas that the monkey could get rid of is $100 - (100 \cdot 0.1 \cdot 0.2) = 98$.

Design an $O(n \log n)$ Divide & Conquer algorithm to find the maximum amount of bananas the monkey can get rid of. Justify correctness of your algorithm, provide a recurrence, and prove its runtime. **You can assume that multiplication is fixed point and is $O(1)$.**

Solution:

- *Algorithm:*

```
def bananaStart (A, b):
    return b - (maximumBananas (A, 1, n) * b)
def maximumBananas (A, start, end):

    if start == end: return A

        L = maximumBananas (A, start, end - start)
        R = maximumBananas (A, end - start + 1, end)
        P = minProduct(A, start, end - start, end):
    return min(1, L, R, P)

    def minProduct(A, start, mid, end):
        left = float('inf')
        product = 1
        for i in range(mid, start - 1, -1):
            product *= arr[i]
            if product < left:
                left = product
        right = float('inf')
        product = 1
        for i in range(mid + 1, end + 1):
            product *= arr[i]
            if product < right:
                right = product
```

```
return min(left * right, left, right)
```

- *Correctness:* I decided to attempt the algorithm by implementing a recursive divide-and-conquer algorithm where we divide the array into two, and use them recursively to call the maximumBananas method again. Afterwards, we go to the minProduct method, which finds the lowest product of arrays from the midpoint to the start and end of the array respectively, we then find the min value of either the left, right, or a product (works because starts from midpoint and goes to each end, thus without skipping indices). Afterwards we then try to find the min value between the min value from minProduct and the two recursively called maximumBananas methods and the default value of 1 (if not going through an array is the best). We then return the lowest value.
- *Time Complexity Analysis:* minProduct is O(n) and min is O(1) . $T(n) = aT(n/b)+n^d$. $T(n) = 2T(n/2) + O(n^1)$. $T(n) = O(nlog(n))$.

Problem 4: Laptop Sleeves

(25 points)

You have n laptops of different sizes and n differently sized laptop sleeves. There is one-to-one relationship between each laptop and their corresponding, perfectly-fitting laptop sleeve. You can try to match each laptop to the right sleeve by determining whether the laptop is larger than the sleeve, is smaller than the sleeve, or fits exactly in the sleeve. **However, there is no way to compare two laptops or two sleeves together.** The problem is to match each laptop to its sleeve. Design a randomized divide-and-conquer algorithm to solve this problem with an expected $O(n \log n)$ running time. Give a brief explanation of your algorithm's correctness and runtime.

Solution:

- *Algorithm:*

```
def totalSort (L, LS, l, u):
    laptopSort (L, LS, l, u)
    Ls2 = [0] * len(L)
    for i in LS:
        Ls2[binarySearch(L, LS(i))] = LS(i)

def laptopSort (L, LS, l, u):
    def partition(L, LS, l, u)
        pivot = LS[u]
        i = l - 1
        for j in l...u-1
            if L[j] < pivot
                i++
                swap L[i] and L[j]
            else if L[j] = pivot
                swap L[j] and L[u]
            swap L[i+1] and L[u]
        return i + 1

    if l < u
        pivotIndex = partition(L, LS, l, u)
        laptopSort(L, LS, l, pivotIndex - 1)
        laptopSort(L, LS, pivotIndex + 1, u)
    laptopSort(L, LS, 1, n)

    binarySearch(L, x):
low = 0
high = len(L) - 1
mid = 0
while low <= high:
```

```
mid = (high + low) // 2
if L[mid] < x:
    low = mid + 1
 elif L[mid] > x:
    high = mid - 1
else:
    return mid
```

- *Correctness:* First the totalSort method is called, which calls the laptopSort method. The laptopSort method is similar to a quicksort, but instead of comparing to the pivot element in the L array, it compares to the pivot element in the LS array. Because of this, an extra step is added to check if L[j] = pivot (fits exactly in the sleeve), in which case L[j] is swapped with L[u] to make the quicksort work normally. (if L[j] = L[u] nothing needs to be changed). After doing the normal steps a quicksort does, we are brought back to the totalSort method with a sorted L array and an unsorted LS array. We then use binary search for i in LS and place LS[i] into the index (result of binary search of LS[i]) into the newly created array Ls2[binarySearch(L, LS(i))]. Thus giving us the two matching arrays L and Ls2.
- *Time Complexity Analysis:* BinarySearch is O(logn) and laptopSort is average time O(nlogn) b/c it is based almost exactly on quicksort (Assuming we are not looking at worse case like the Ed discussion said). T(n) = O(nlogn) + n * O(logn) = O(nlogn) + O(nlogn) = 2 O(nlogn) = O(2nlogn) = O(nlogn)