CS 3510 Section B: Design & Analysis of Algorithms August 30th 2024

Problem Set 2: Divide & Conquer and Arithmetic

Jinseo Lee Due: September 5th 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?

1. Describe (in words) your algorithm (to a degree where someone could implement it if given your answer).

   Solution:

   I will use the merge sort algorithm to divide the array in half until the size of each array equals 1, just like how division process of the traditional merge sort works. However, when merging them back together, I will be tracking the number of elements on the left that is smaller than the current element in the right array. The sum of those counts is the output.

2. Argue why your algorithm is correct, meaning given some input, why does your algorithm produce the correct outputs?

   Solution:

   The algorithm is correct for any input as it adapts the divide and conquer method, recursively dividing the array and counting while adding up. Since it constantly checks for the inversion from the smallest piece to the entire array, the algorithm concisely counts the number of two adjacent elements in an ascending order. For example, the array $[82, 100, 70, 98, 100]$ is divided in half, producing $A_l = [82, 100, 70]$ and $A_r = [98, 100]$. Then, A_l divides into $A_{ll} = [82, 100]$ and $A_{lr} = [70]$. The left one is then divided into 82 and 100, and the division for A_l is done. At the same time, A_r is divided into 98 and 100, and the division is done. When merging back up, the counter is added by 1 as 100 is larger than 82. The next step does not add anything to the

sum as 70 is not bigger than any elements in the left. Merging 98 and 100 adds 1 to the counter as 100 is bigger than 98. When merging $[82, 100, 70]$ and $[98, 100]$, the algorithm compares the first element in the right array, 98, with the left array. Since 98 is larger than 82 and 70, 2 is added to the sum. Then, it compares 100 with the left array and adds 2 to the sum as 100 is bigger than 82 and 70. Therefore, the sum equals 6 at the end.

3. Describe the recurrence and runtime of your algorithm.

Solution:

There are two subproblems as merge sort algorithm divides the array in half, with each problem size of $n/2$. The extra work of $O(n)$ is done when merging back the arrays. $T(n) = 2T(n/2) + O(n)$. $(a = 2, b = 2, d = 1)$. Since $log_b(a) = log_2(2) = 1 = d$, case 2 of Master Theorem applied. The runtime is $O(n^d log(n) = O(nlog(n))$.

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.

1. Describe (in words) your algorithm (to a degree where someone could implement it if given your answer).

   Solution:

   My algorithm will use binary search to find the minimum fixed rate 'r' that allows Aqua to get her work done within the designated time 'h'. First, by going through the array, it will find the largest element, x, in the array and sets the range of 'r' to [1,x]. Then, it will start finding the amount of time it will take to finish the work with the median rate, int $(1 + x)/2$. If it works with such value, then it recursively calculates the range by getting the median of the median on the left side. If it doesn't work, on the other hand, it would recalculate by proceeding to the right side of the range.

2. Argue why your algorithm is correct, meaning given some input, why does your algorithm produce the correct outputs?

   Solution:

   The algorithm I have came up with effectively finds the minimum rate r by narrowing the range of r and finding the smallest value that satisfies the time condition. Specifically, it uses binary search as it is certain that the array contains a series of every possible rates, which must be in an ascending order. It narrows down to the smallest possible rate by continuously ignoring the other half. For example, let tanks $= [3, 6, 10]$ where n = 3 and h = 5. Since the biggest element equals 10, the range is [1,10] and the r is 5. It takes 1 hour to drain 3L, 2 hours for 6L, and 2 hours for 6L, which sums up to 5 hours. As it is working, the algorithm proceeds to the left half of the range and recursively recalculate the r value. $(1 + 5)/2 = 3$, r is now 3. It takes 1 hour for 3L, 2 hours for 6L, and 4 hours for 10L, which sums up to 7 hours. Since it's not working, the algorithm now tries 4. It takes 1 hour for 3L, 2 hours for

6L, and 3 hours for 10L, which adds up to 6 hours. Since it's also not working, the r value narrows down to r = 5.

3. Describe the recurrence and runtime of your algorithm.

Solution:

The runtime of recursively calculating the r value depends on its initial range. Let the range be X, then its runtime is $log_2(X)$. Furthermore, the calculation is done for every tanks in the array, which does O(n) works. Therefore, the overall runtime for my algorithm is O(log(X)) x O(n) = O(nlog(X)).

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 efficient 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.

1. Describe (in words) your algorithm (to a degree where someone could implement it if given your answer).

   Solution:

   The algorithm first checks if there are any value in A that is below 1. If all elements are bigger than or equal 1, it returns the smallest element. If not, it proceeds to the recursive division method of merge sort until it reaches the individual element. Then, it merges them back up, tracking the minimum product between each element. Specifically, it checks for the minimum product of the following: minimum product of the left array, right array, and the boundary (last element of the left array times first element of the right array.) This process continues until it explores the entire array, ensuring that every product of the subarrays are identified. Once it finds out the minimum reduction of the entire array, it subtracts (minimum reduction times 100) from 100.

2. Argue why your algorithm is correct, meaning given some input, why does your algorithm produce the correct outputs?

   Solution:

   My algorithm is correct as it uses the divide and conquer method to consider every minimum reductions that can be produced by the series of element in the array. When merging the arrays back up, it identifies the minimum by thoroughly comparing the left half, right half, and their boundaries. For example, the array [0.1, 0.2, 100, 1, 0.2] is divided in half, producing [0.1, 0.2] and [100, 1, 0.2]. The right array is again divided in half, producing [100] and [1, 0.2] The left array's min product is 100 itself, boundary is 100 (100 times 1), and that of right is 0.2 ($0.2 < 1$), which is the current minimum ($0.2 < 100$). Merging back up, the minimum product of the left array is 0.02, that of the right array is 0.2, and boundary is 20. Therefore the minimum

product of the entire array is 0.02, and the monkey could get rid of his 100 - (100 * 0.02) = 98 rotten bananas.

3. Describe the recurrence and runtime of your algorithm.

Solution:

My algorithm recursively divides the array in half, producing two sub-problems with each size of n/2. Additionally, it does the extra O(n) work when merging back the arrays and finding the minimum product. Therefore, the recurrence equation is T(n) = 2T(n/2) + O(n), and the runtime is O(nlog(n)) as case 2 of master theorem applied ($log_b(a) = log_2(2) = 1 = d$).

Problem 4: Political Parties

(25 points)

You are a visitor at a political convention (or perhaps a faculty meeting) with n delegates; each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to; in particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any pair of delegates belong to the same party by introducing them to each other. Members of the same political party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.

Write a divide and conquer algorithm to find the largest political party (i.e. the party with the most members).

1. Describe (in words) your algorithm (to a degree where someone could implement it if given your answer).

   Solution:

   My algorithm will recursively divide the whole visitors at a political convention in half, until there is only one person left in each divided group. Then, it will merge each group back up, identifying whether there are any visitor sharing the political party. Specifically, it will first check whether the people in the same group smiles with each others. Then, it will identify number of people in the particular friendly group from the merged group. In this process, the algorithm does not have to compare every single members with one another; rather, it could merely compare one member from the dominant group with the others to identify their party.

2. Argue why your algorithm is correct, meaning given some input, why does your algorithm produce the correct outputs?

   Solution:

   My algorithm is correct as it divides the big group of visitors into several pieces (recursively in half), facilitating the careful identification of the party from the smallest group to the entire visitors. Since it is guaranteed that there exists one major political party, the corresponding party will eventually emerge as a dominant group (Even there was another potential dominant party, it's impossible for them to surpass the actual largest party.) For example, given the array indicating one's the political party, [A,B,A], it is then divided in half, producing [A] and [B,A]. The right group is further divided into [B], and [A]. Neither the left group nor the right group has the dominant party. [B] and [A] is then merged back up, forming [B,A] again without a distinct party. However, when combining [A] with [B,A], the largest party, A, is identified.

3. Describe the recurrence and runtime of your algorithm.

Solution:

The algorithm recursively divides the problem (visitors of size n) in half with each subproblem size of n/2. Also, it does an extra work of O(n) when merging those groups back and identifying the largest party. a is 2, b is 2, and d is 1 as the recurrence equation is shown as following: T(n) = 2T(n/2) + O(n). $log_b(a) = log_2(2) = 1 = d$, case 2 of master theorem applied. Therefore, the runtime is $O(n^d log(n)) = O(nlog(n))$.