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

Problem Set 2: Divide & Conquer and Arithmetic

Wesley Lu 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:

   Steps:
   1. Perform Merge Sort on the inputted array and recursively divide the array until the base case.
   2. Combine the two sorted sub-arrays ($arr1$ and $arr2$). Use a two pointer method to merge the subarrays.
   3. Initialize two pointers i and j both starting at 0 which represents $arr1$ and $arr2$ respectively. *counter* should've keep initalized to keep track of upwards trajectories and returned.
   3.1 If $arr1[i] \geq arr2[j]$, increment index j.
   3.2 If $arr1[j] < arr2[j]$, add to the counter $arr2.length - j$ and increment i; this will return the amount of integers greater than element j. Incrementing i insures that other elements also follow this rule.
   3.3 Continue this until either i or j are out of bounds for $arr1$ and $arr2$ respectively.
   4. When merging is finished, return the counter.

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

Solution:

The reason this algorithm works when combining arrays is due to the nature of merge sort which returns two sorted arrays. Given two sorted arrays $arr1$ and $arr2$, there will be two points, i and j for $arr1$ and $arr2$ respectively. An upward trajectory is counted where $arr[i] < arr2[j]$. When combining the two arrays, the pointers start at position 0 or the start of the array.

If $arr1[i] \geq arr2[j]$, no upward trajectory occurs and the pointer j is moved up one until there is an element in $arr2$ that is greater.

If $arr1[i] < arr2[j]$, a upward trajectory occurs. This means that all the element greater than pointer j is also greater than the element at pointer i. The counter is updated by adding $arr2.length - j$ to get all elements greater than pointer j. Then update pointer i by adding one to find more upward trajectory. The linear search stops when i and j are greater than their array's respective lengths.

Example:

$$A = [115, 132, 146, 88, 136, 72, 89, 87],\ counter = 0$$

$$[115, 132, 146, 88] \quad [136, 72, 89, 87]$$

$$[115, 132] \ \ [146, 88] \ \ [136, 72] \ \ [89, 87]$$

$$[115]\ [132]\ [146]\ [88]\ [136]\ [72]\ [89]\ [87]$$

$$arr1 = [115], arr2 = [132], i = 0, j = 0$$

$$arr1[i] < arr2[j],\ n = arr2.length = 1,\ counter\ += n - j,\ \text{and}\ i{+}{+}.$$

(However, i is out of bounds. Thus continue to the next combine.)

$$arr1 = [146],\ arr2 = [88],\ i = 0,\ j = 0$$

$$arr1[i] \geq arr2[j] \implies j{+}{+}\ \text{(Pointer j is out of bounds, continue.)}$$

$$arr1 = [136],\ arr2 = [72],\ i = 0,\ j = 0$$

$$arr1[i] \geq arr2[j] \implies j{+}{+}\ \text{(Pointer j is out of bounds, continue.)}$$

$$arr1 = [89],\ arr2 = [87],\ i = 0,\ j = 0$$

$$arr1[i] \geq arr2[j] \implies j{+}{+}\ \text{(Pointer j is out of bounds, continue.)}$$

$$[115, 132]\ \ [88, 146]\ \ [72, 136]\ \ [87, 89]$$

$$arr1 = [115, 132],\ arr2 = [88, 146],\ i = 0,\ j = 0$$

$$arr1[i] \geq arr2[j],\ 115 \geq 88 \implies j{+}{+},\ j = 1$$

$$arr1[i] < arr2[j],\ 115 < 146 \implies i{+}{+},\ count{+}{=}arr2.length - j,\ count{+}{=}1$$

$$arr1[i] < arr2[j],\ 132 < 146 \implies i{+}{+},\ count{+}{=}arr2.length - j,\ count{+}{=}1$$

(i is out of bounds)

$$arr1 = [72, 136],\ arr2 = [87, 89],\ i = 0,\ j = 0$$

$$arr1[i] < arr2[j] \implies i{+}{+},\ count{+}{=}arr2.length - j,\ count{+}{=}2$$

$$arr1[i] \geq arr2[j] \implies j{+}{+}$$

$$arr1[i] \geq arr2[j] \implies j{+}{+} \text{ (point j is out of bounds.)}$$

$$[88, 115, 132, 146]\ \ [72, 87, 89, 136]$$

$$arr1 = [88, 115, 132, 146],\ arr2 = [72, 87, 89, 136],\ i = 0,\ j = 0$$

$$arr1[i] \geq arr2[j] \implies j{+}{+},\ j = 1$$

$$arr1[i] \geq arr2[j] \implies j{+}{+},\ j = 2$$

$$arr1[i] < arr2[j] \implies count{+}{=}arr2.length - j,\ count{+}{=}2,\ i{+}{+},\ i = 1$$

$$arr1[i] \geq arr2[j] \implies j{+}{+},\ j = 3$$

$$arr1[i] < arr2[j] \implies count{+}{=}arr2.length - j,\ count{+}{=}1,\ i{+}{+},\ i = 2$$

$$arr1[i] < arr2[j] \implies count{+}{=}arr2.length - j,\ count{+}{=}1,\ i{+}{+},\ i = 3$$

$$arr1[i] \geq arr2[j] \implies j{+}{+},\ j = 4\text{(point j is out of bounds}$$

return count = 9

3. Describe the recurrence and runtime of your algorithm.

Solution:

The merging and counting is done linearly. The overall time taken by the algorithm is

$$T(n) = 2T(n/2) + O(n)$$

For the constants $a = 2, b = 2, d = 1$ in the runtime $d = \log_b a$ thus

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

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:

Steps:
1. Initialize two boundary $left = 1$ and $right = 1$
2. Loop through the array $tanks$ to find the maximum for right
3. Perform binary search to check if Aqua can drain all the water within h hours.
4. Get the middle value which is $(left + right)/2$
5. If Aqua can finish all tanks within h hours, set $right$ equal to middle. This means that the new search space has the current speed as the max because we are trying to find the minimum working value.
6. If Aqua cannot finish all tanks, then set $left$ equal to middle + 1. The values left of middle are all not workable.
7. Loop until $left = right$ meaning that the minimum value, r, is found.

Pseudocode

```
minimumValue(int[] tanks, int h):
    left = 1
    right = 1

    // Find the maximum speed within search
    for (int tank : tanks):
        right = max(right, tank)

    // Loop until r is minimized
    while (left < right):
```

```
            // Middle point is calculated
            mid = (left + right) / 2

            // Calculate the current working speed
            hours = 0;
            for (int tank : tanks):
                hours += ceiling(tank / mid)

            // Set new boundaries
            if hours <= h: right = mid
            else: left = mid + 1

        return right
```

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

Solution:

Given the input $tanks$ and total hours h, I initialize the boundaries for my search space. It can be assumed that the hours h is going to be greater than or equal to the length of $tanks$, $tanks.length \leq h$. If the maximum working speed r is $max(tanks[i])$, it cannot finish all piles given that each pile will take 1hr. Therefore, $right$ is set to the maximum of a value within $tanks$. This value is guaranteed to finish all tanks, but the minimum rate. The boundary $left$ is set to 1. It's guaranteed that if Aqua can finish at a fixed time r, than he can finish in $r+1$ rate. Intuitively, it also means that if Aqua cannot finish at a fixed r, than he cannot finish in $r-1$ rate. Within the search space, there will be some value that is the minimum of the all working speed such that you can search within $[1, 2, 3, ..., max(tanks[i])]$. Binary search will continuously update the boundary until it converges on the minimal working rate r for Aqua to finish within h hours. The boundary are going to update by checking if the current mid value is able to finish all tanks. Thus, the algorithm will successfully find the correct output r.

3. Describe the recurrence and runtime of your algorithm.

Solution:

The algorithm first loops through the array to set the boundary $right$ in linear time, $O(n)$. Binary Search halves the search space within each iteration. The recurrence is $n/2$ as binary search splits the search space into 2 with only one sub-problem. The range of search is between 1 and $max(tanks[i])$. This divides the algorithm into $\log max(tanks[i])$. Lets assume $m = max(tanks[i])$, this binary search function takes $O(\log m)$. Within the binary search, it takes linear time to find out how many hours it would take to process all tanks with the current speed mid, $O(n)$. Therefore, the

runtime is

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

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:

Steps:
1. Perform a divide and conquer algorithm. Initialize int $l = 0$ and $r = n - 1$ where n is the length of the given *double* array.
2. Within the $minProduct()$ function, establish the base case where the right and left pointer point to the same element.
3. Initialize the middle index to recursively divide the array into two parts.
4. Recursive call $minProduct()$ on the left and right side, $(1, m)$ and $(m + 1, r)$
5. Call $fromMiddleIndex()$ function to handle minimum subarray product that cross over the middle point.
6. Within the $fromMiddleIndex$ function, calculate the left side product starting from the middle index and accumulate the smallest product. Also do the same for the right side starting from $m + 1$. Return the minimum $min(leftProduct * rightProduct, leftProduct, rightProduct)$.
7. After receiving the minimum product return the maximum number of bananas $b - (b * min) = maximumBananas$

Pseudocode

```
// Return the maximum bananas removed given a double array
// with multipliers
maximumRemovedBananas(double arr[], int l, int r, int b):
    double minProduct = minProduct(arr, l, r)
    return b - (b*minProduct)
```

```
minProduct(double arr[], int l, int r):
   if (l == r): return arr[1]

   m = (l + r) / 2

   return min(minProduct(arr, 1, m), minProduct(arr, m + 1, r),
            fromMiddleIndex(arr, l, m, r)

fromMiddleIndex(double[] arr, int l, int m, int r):
    leftProduct = 1
    rightProduct = 1
    minLeft = MAX_VALUE
    minRight = MAX_VALUE

    // Get the left from mid
    for (int i = m; i >= 1; i--):
        leftProduct *= arr[i]
        if (leftProduct < minLeft): minLeft = leftProduct

    // Get the right from mid
    for (int i = m+1; i <= r; i++):
        rightProduct *= arr[i]
        if (rightProduct < minRight): minRight = rightProduct

    // Reutrn minProduct between all three
    return min(minLeft*minRight, minLeft, minRight)
```

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

Solution: Given an input of an array, this algorithm always correctly produces the maximum value. The key itself is to the find the sub-sequence produces the smallest product. The function $minProduct$ always recursively splits the array at the mid-point. The algorithm uses $fromMiddleIndex$ to handle cases where the smallest product sub-sequence can spans across the middle point. This divide and conquer algorithm ensure that both the left, right, and spanning the middle cases rae checked for all possible subsequence.

3. Describe the recurrence and runtime of your algorithm.

Solution:

The *double* array is recursively divided into two halfs using the mid point $m = \frac{l+r}{2}$. This results in two subproblems. The algorithm calculates the minimum product that crosses the midpoint using $fromMiddleIndex()$ where it iterates over the left

and right side of the array in linear time. This takes $O(n)$ in the conquer steps. Therefore the runtime is:

$$T(n) = 2T(n/2) + O(n)$$

Using the constants $a = 2, b = 2, d = 1$ then $d = \log_b a$. Therefore the algorithm has a runtime of:

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

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:

   Steps:
   1. The base case is if there is only one delegate in an array, the largest party size is 1
   2. Recursively split the delegates into two list using the midpoint m
   3. Introduce a delegate from the left half to the right half. If they're in the same group, merge and update the largest counter. If not, then the largest party is the $max(arr1, arr2)$
   4. The largest value is returned.

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

   Solution:

   Using a merge sort like algorithm works because it breaks down the problem into subproblems where they are solved. Initially, the list is divided into two smaller subproblems until there is just one left - base case. Then, the merge step finds the majority party in each half by continuously comparing. If the two halves are the same party then the party is majority for both. If it's different parties, you would need to recount by introducing delegates.

3. Describe the recurrence and runtime of your algorithm.

   Solution: The algorithm divides n delegates into size $n/2$ recursively which takes T(n/2). The merging step takes $O(1)$ if it's the same majority party and $O(n)$ if they are different parties. There is two subproblems that are being solved. Therefore, the recurrence relation is:

   $$T(n) = 2T(n/2) + O(n)$$

   Given the constants $a = 2, b = 2, d = 1$ then $d = \log_b a$. According to the master's

theorem, the runtime of this algorithm is:

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