Introduction to Algorithm Homework 2

Mou Zhang

February 3rd, 2020

1 Problem 1 (12 points)

Given a list of n integers $x_1, \ldots, x_n$ (possibly negative), find the indices $i, j \in [n]$ ($i \neq j$) such that $x_i \cdot x_j$ is maximized. Your algorithm must run in $O(n)$ time.

Solution: There are two possible ways the max product comes from: the product the two largest positive integers and the product of two smallest negative integers. So what we need to do is to find the two biggest positive integers and two smallest negative integers by scanning the whole array and comparing all elements in the array in $O(n)$ time.

Proof of correctness:

Proof. It is obvious that the max product only comes from the product of 2 biggest positive numbers and 2 smallest negative numbers. Firstly ,the product must be positive so that both factor must be positive or negative at the same time. For the positive side, if A[i], A[j] are the biggest 2 positive numbers, than for any other positive number A[k]($k \neq i, j$), $A[i] \ldots A[j] > A[i] \ldots A[k] A[i] \ldots A[j] > A[k] \ldots A[j]$. Similarly, for negative factors, they must be the smallest negative integers.

The corner case here is that we don't have enough positive and negative numbers. Only when the length is 2 can cause the problem. And under this circumstance, the biggest product is the product of A[0] and A[1].

So the only thing we need prove now is that this loop can get 2 biggest positive numbers and 2 smallest negative numbers we want.

Initialization:

We start to show the the loop invariant holds before the first iteration. In lines 5-8, all for numbers are equal to 0, which means that number are not recorded yet. Notice that the product with a factor of 0 is 0, it means that if we don;t have enough

Algorithm 1 Find max product

```
1: procedure FIND MAX PRODUCT(A)
2:     if A.length = 2 then
3:         return A[0] × A[1]
4:     end if
5:     biggest_number_index ← 0
6:     second_biggest_number_index ← 0
7:     smallest_number_index ← 0
8:     second_smallest_number_index ← 0
9:     for i ← 1, A.length do
10:        if A[i] > A[biggest_number_index] then
11:            second_biggest_number_index ← biggest_number_index
12:            biggest_number_index ← i
13:        else
14:            if A[i] > A[second_biggest_number_index] then
15:                second_biggest_number_index ← i
16:            end if
17:        end if
18:        if A[i] < A[smallest_number_index] then
19:            second_smallest_number_index ← smallest_number_index
20:            smallest_number_index ← i
21:        else
22:            if A[i] < A[second_smallest_number_index] then
23:                second_smallest_number_index ← i
24:            end if
25:        end if
26:    end for
27:    if A[biggest_number_index] × A[second_biggest_number_index] >
       A[smallest_number_index] × A[second_smallest_number_index]) then
28:        return biggest_number_index, second_biggest_number_index
29:    else
30:        return smallest_number_index, second_smallest_number_index
31:    end if
32: end procedure
```

positive or negative numbers, the product gonna be 0, which is definitely not the final answer. So in the start part, we find out that all four numbers are 0, which means the 2 biggest number and the 2 smallest number are all 0, representing these four numbers in A[1..0], which is empty.

Maintenance:

Induction hypothesis (IH): suppose statements that we have the 2 biggest positive numbers and 2 smallest negative number for the A[1..t] for the loop invariant hold after iteration number t.

Induction step (IS): let's prove it will still hold for A[1..(t + 1)] after iteration number t + 1.

If the new number A[t + 1] is bigger than the previous biggest number, than the new biggest number becomes A[t + 1](line 11), the original biggest become the second largest number(line 12). Otherwise, if A[t + 1] is smaller than the original biggest number and bigger than the original second largest number, then the new second largest number becomes A[t + 1]. If A[t + 1] is smaller than the original second largest number, nothing happens. After this process, we find out that biggest number and second biggest number are kept well in this iteration. Similarly, 2 smallest numbers are kept as well. So for iteration number t + 1, the statements that we have the 2 biggest positive numbers and 2 smallest negative numbers for the A[1..(t + 1)] holds.

Termination:

Finally, we examine what happens when the loop terminates.The condition causing the for loop to terminate is that $i > A.length = n$. Because each loop iteration increases i by 1, we must have $i = n + 1$ at that time. After the whole loops, the statements that we have the 2 biggest positive numbers and 2 smallest negative numbers for the A[1..A.length] holds. Therefore, we have the 2 biggest positive numbers and 2 smallest negative numbers for the whole array. □

Proof of running time

Proof. To scan the whole array takes $O(n)$ time, and each comparison and assignment takes $O(1)$ time. So the total time complexity is $O(n) \times O(1) = O(n)$ □

2 Problem 2 (12 points)

Let S be an array of integers $\{S[1], S[2], \ldots, S[n]\}$ such that $S[1] < S[2] < \cdots < S[n]$. Design an algorithm to determine whether there exists an index i such at $S[i] = i$. For example, in $\{-1, 2\}$, $S[2] = 2$.

Your algorithm should work in $O(\log n)$ time. Prove the correctness of your algorithm.

Solution: We can use binary search for this problem. Suppose T[n] = A[n] - n. Notice that $S[1] < S[2] < \cdots < S[n]$, so we have $S[i] + 1 \leq S[i + 1]$ for any i in range (1,n - 1). Then $T[i + 1] - T[n] = S[i + 1] - (i + 1) - (S[i] - i) = S[i+1]-S[i]-1 \geq 0$, so T[n] is monotonic non-decreasing and in order. Therefore, binary search works for this array.

Algorithm 2 Check if S[i] = i exists

```
1:  procedure CHECK IF S[I] = I EXISTS(S)
2:      for i ← 1, n do T[i] ← S[i] − i
3:      end for
4:      l ← 1
5:      r ← n
6:      while l ≤ r do
7:          mid ← (l + r)/2
8:          if T[i] = 0 then
9:              return true
10:         else
11:             if T[i] > 0 then
12:                 r ← mid − 1
13:             else
14:                 l ← mid + 1
15:             end if
16:         end if
17:     end while
18:     return false
19: end procedure
```

Proof of correctness:

Proof. We know that T[n] = A[n] - n. Notice that $S[1] < S[2] < \cdots < S[n]$, so we have $S[i] + 1 \leq S[i + 1]$ for any i in range (1,n - 1). Then $T[i + 1] - T[n] = S[i+1] - (i+1) - (S[i] - i) = S[i+1] - S[i] - 1 \geq 0$, so T[n] is monotonic non-decreasing and in order. The original requirement is equal to find index i which

$T[i] = 0$. Therefore, binary search works for this array.

Now Let's prove this binary search by induction:
Base Case. In the case where n=1, we know left=right=m. Obviously, this is only possible answer to the question, so what we need to so is to check weather it's valid.

Inductive Step. This binary search works as long as left $-$ right $\leq$ k. Our goal is to prove that it works on an input where left $-$ right = k + 1. There are three cases, where x = a[m], where x $<$ a[m] and where x $>$ a[m].
For case x = a[m], it's obviously correct.
For case x $>$ a[m], We know because the array is sorted that x should be between a[left] and a[m-1]. The length of new searching domain is definitely smaller than original left to right(which is k). Therefore, due to the induction hypothesis, this must be correct.
For case x $<$ a[m], similarly, the statement is correct.
Therefore, we can conclude that binary search here is correct.

□

Proof of efficiency:

Proof. Every iteration in binary search, we half the searching domain. Suppose $n \leq 2^k$, then it takes maximum k steps to find the answer. In each step, we use O(1) time to check and decide the next searching domain. So the time complexity is $O(k) \times O(1) = O(\log n) \times O(1) = O(\log n)$ □

3 Problem 3 (13 points)

We say a 3-tuple of positive real numbers (x_1, x_2, x_3) is *legal* if a triangle can have sides of length x_1, x_2 and x_3. Given a list of n positive real numbers $\{x_1, \ldots, x_n\}$, count the number of unordered 3-tuples (x_i, x_j, x_k) that are legal. For example, for the numbers $\{3, 5, 8, 4, 4\}$, $(3, 4, 5)$ is a legal tuple while $(4, 4, 8)$ is not.

Your algorithm should run in $O(n^2)$ time. Prove correctness of your algorithm.
EDIT: You may give an $O(n^2 \log n)$ time algorithm and get full-credit.

Solution: Let's assume the 3 edges of a triangular are a, b, c and length(a) ¡ length(b) ¡ length(c). In this algorithm, at first, we sort the whole array in advance. After that we enumerate a and c while using a pointer to point out the place of minimum possible value of b. Finally, we count the number of possible values in the array for every a and c.

Algorithm 3 count all tuples

```
1:  procedure COUNT ALL TUPLES(X)
2:      count = 0
3:      X = Sort_In_Ascending_Order(X)
4:      for i ← 1, n − 2 do
5:          j ← i + 1
6:          k ← i + 2
7:          while k ≤ n do
8:              while j ≤ k and A[j] ≤ A[k] − A[i] do
9:                  j ← j + 1
10:             end while
11:             count ← count + (k − j)
12:             k ← k + 1
13:         end while
14:     end for
15:     return count
16: end procedure
```

Proof of correctness:

Proof. Let's assume the 3 edges of a triangular are a, b, c and length(a) ¡ length(b) ¡ length(c). In this algorithm, at first, we sort the whole array in advance. After that we enumerate a and c while using a pointer to point out the place of minimum possible value of b. Finally, we count the number of possible values in the array for every a and c.To sum them up, we have considered all possible tuples in this

array. Now let's prove this algorithm works by induction.

Initialization:

Let's assume a = x[i] and do not change in this induction. Before entering the loop for k, count is equal to number of tuples with a = x[ii](ii ¡ i).

Maintenance:

Induction hypothesis (IH): suppose statements that we have calculated all tuples with a = x[i] and c = x[k] for the loop invariant hold after iteration number i and k. Induction step (IS): let's prove it will still hold for c = x[k + 1]. When c become x[k + 1], first we need to move j so that $A[j] > A[k] - A[i]$(which comes from the definition of triangle that the sum of two edges is bigger than the third edge, the difference of two edges is smaller than the third edge). Since the array is in ascending order and i does not change while k increase, A[k] - A[i] is definitely increasing. Therefore, we don't need to calculate all j before the previous j(Let's call it j_0), because it's in ascending order. And after find the j we want, we can figure out that all values between A[j] and A[k - 1] are suitable for being the second longest edge with edge a and c. So (k - j) is the number of suitable values for serving as the second longest edge with edge a and c. Let's add it to count. The statement is proved.

Termination:

After k goes to n, every possible triangles starts with x[i] are calculated into count. So we can move forward to x[i + 1]. Finally, i become n - 2 and all the loop end. In each iteration of i, we calculate the number of all possible triangles with the shortest edge as x[i], and in each iteration of k with a = x[i], we calculate the number of all possible triangles with the shortest edge as x[i] and longest edge as x[k]. Therefore, the correctness of this algorithm is proved. □

Proof of efficiency:

Proof. In the whole algorithm, we get for loop in line 4, which counts for $O(n)$. we get 2 while loops in line 7 and 8, each is supposed to count for $O(n)$. However, we notice that in these 2 while loops, k changes from i + 2 to n with no decrease, and j changed from i + 1 to k($k \leq n$) with no decrease. These two while loops are working separately with $O(n)$ time complexity each. So the total time complexity of these two while loops in lines 7-13 is $O(n)+O(n) = O(n)$. Since everything we do in loops are $O(1)$, we have the total time complexity is equal to $O(n) \times O(n) = O(n^2)$. □

4 Problem 4 (13 points)

You are given one unsorted integer array A of size n. You know that A is almost sorted, that is it contains at most m inversions, where inversion is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$.

1. To sort array A you applied algorithm Insertion Sort. Prove that it will take at most $O(n + m)$ steps.

   Solution:

   Proof. Let's look at the pseudo code of Insertion Sort.

Algorithm 4 Insertion Sort

```
1:  procedure INSERTION SORT(A)                ▷ Sort Array A with Insertion Sort
2:      for j ← 2, A.length do
3:          key ← A[j]
4:          i ← j − 1
5:          while i > 0 and A[i] > key do
6:              A[i + 1] = A[i]
7:              i = i − 1
8:          end while
9:          A[i + 1] = key
10:     end for
11: end procedure
```

   We find out that in this algorithm, everything except for the while loop only requires $\Theta(n)$ time.Observing the while loop, we can find out that what each iteration of the while loop do is to swap adjacent pair of out of order elements A[i], A[i + 1]. Such a swap decreases the number of inversions by Since there are no other means to reduce the number of inversions, the total number of iterations of the while loop must be equal to m. Thus, the time complexity is $\Theta(n + m)$ and it will take at most $O(n + m)$ steps.

   □

2. What is a maximum possible number of inversions in the integer array of size n?

   Solution: $\frac{n(n-1)}{2}$

Proof. In the worst case, the array is sorted in reverse order, which has the maximum possible number of inversions. In this case, A[0](the first element) is bigger than all n - 1 elements behind it, so the number of inversion pairs starts with A[0] is (n - 1). Similarly, the number of inversion pairs that starts with A[1] is (n - 2)..... The number of inversion pairs that starts with A[n - 1] is 1. The number of inversion pairs that starts with A[n] is 0. Thus the total number of inversions are $(n-1)+(n-2)+...+1+0=\frac{n(n-1)}{2}$ $\square$