HW4

Keerthana Golla

September 2023

1 Problem - 1

```
int findMinMaxAbsolute(int A[], int n) {
    int max, min, i;
    if (n % 2 != 0) {
        max = A[0];
        min = A[0];
        i = 1;
    } else {
        if (A[0] < A[1]) {
            max = A[1];
            min = A[0];
        } else {
            max = A[0];
            min = A[1];
        }
        i = 2;
    }
    while (i < n) {
        if (A[i] < A[i + 1]) {
            if (A[i] < min) {
                min = A[i];
            }
            if (A[i + 1] > max) {
                max = A[i + 1];
            }
        } else {
            if (A[i] > max) {
                max = A[i];
            }
            if (A[i + 1] < min) {
                min = A[i + 1];
            }
        }
        i = i + 2;
    }
    int maxMinAbsoulte = |max - min|;
    return maxMinAbsoulte;
}
```

The above algorithm efficiently finds both the maximum and minimum values in an array and then do a absolute difference and return the difference . This algorithm is designed to minimize the number of comparisons needed.

Correctness:

Consider the case where $A[i] < A[i+1]$. In this scenario, there is a chance that $A[i]$ is less than the current minimum ('min') and $A[i+1]$ is greater than the current maximum ('max'). Therefore, $A[i]$ becomes the candidate for the

minimum, and $A[i+1]$ becomes the candidate for the maximum. We can update 'min' and 'max' accordingly as follows:

1. If $A[i] < \min$, update $\min = A[i]$.
2. If $A[i+1] > \max$, update $\max = A[i+1]$.

Now, consider the case where $A[i] > A[i+1]$. In this scenario, $A[i+1]$ becomes the candidate for the minimum, and $A[i]$ becomes the candidate for the maximum. We can update 'min' and 'max' as follows:

1. If $A[i] > \max$, update $\max = A[i]$.
2. If $A[i+1] < \min$, update $\min = A[i+1]$.

The above scenarios proves the correctness of algorithm .In both scenarios, we make three comparisons (in the worst case) to update the maximum and minimum of two elements.

Time Complexity:
For each pair of elements in the array, we perform three comparisons:
First, we compare the elements of the pair to determine which one is smaller and which one is larger.Then, we perform two more comparisons to update the minimum and maximum values, comparing the smaller element with the current minimum and the larger element with the current maximum. The total number of such pairs is $(n-1)/2$ if 'n' is odd, or $n/2-1$ if 'n' is even. Therefore, the total number of comparisons is:

$$\text{Comparisons} = 3 \cdot \left(\frac{n-1}{2}\right) = \frac{3n-3}{2} \quad \text{if 'n' is odd}$$

$$\text{Comparisons} = 3\left(\frac{n}{2}-1\right) = \frac{3n}{2}-3 \quad \text{if 'n' is even}$$

The number of comparisons in worst case is at most $\frac{3n}{2}$. In both cases, the number of comparisons is linearly proportional to 'n', which means the time complexity of this algorithm is $\mathcal{O}(n)$.

2 Problem - 2

Algorithm 1 Find the kth Smallest Element in the Intersection of Two Unsorted Arrays

```
1: procedure kthSmallestIntersection(A, B, k)
2:     Sort both arrays A and B                ▷ Use merge sort which takes O(nlogn) for n length array
3:     ptrA ← 0
4:     ptrB ← 0
5:     count ← 0
6:     while ptrA < length(A) and ptrB < length(B) do
7:         if A[ptrA] = B[ptrB] then
8:             count ← count + 1
9:             if count = k then
10:                return A[ptrA]
11:            end if
12:            ptrA ← ptrA + 1
13:            ptrB ← ptrB + 1
14:        else if A[ptrA] < B[ptrB] then
15:            ptrA ← ptrA + 1
16:        else
17:            ptrB ← ptrB + 1
18:        end if
19:    end while
20:    return −1                               ▷ Value of k is out of bounds
21: end procedure
```

Above is the pseudo code is for finding the Kth smallest element in the intersection of two unsorted arrays .

Correctness:

The algorithm starts by sorting both arrays A and B in $O(n \log n)$ time each. This step ensures that elements are aligned and ready for comparison.

It initializes two pointers, $ptrA$ and $ptrB$, both initially pointing to the start of their respective sorted lists. It also initializes a count variable to keep track of the number of common elements found.

The while loop runs until at least one of the pointers reaches the end of its respective list, which ensures that we check all the elements in both lists at most once.

Inside the loop, the algorithm compares the elements pointed to by $ptrA$ and $ptrB$. There are three cases:

a. If the elements are equal, the algorithm increments the count, and if count equals k, it returns the common element as the kth smallest element.

b. If $A[ptrA] < B[ptrB]$, it means that $A[ptrA]$ is smaller, so $ptrA$ is advanced to the next element in A.

c. If $A[ptrA] > B[ptrB]$, it means that $B[ptrB]$ is smaller, so $ptrB$ is advanced to the next element in B.

If the loop completes without finding the kth smallest element, the algorithm returns -1, indicating that k is out of bounds.

The correctness of the algorithm can be demonstrated by observing that it consistently advances the pointers $ptrA$ and $ptrB$ in a manner that guarantees the comparison of elements in a sorted order. Given that both lists A and B are sorted, the algorithm effectively identifies common elements, and it handles comparisons and edge cases correctly

Time Complexity Analysis ($O(n \log n)$):

The initial sorting of both arrays A and B takes $O(n \log n)$ time, where "n" is the total number of elements in A, B arrays ($O(2(n \log n)) = O(n \log n)$).

After sorting, the while loop runs in $O(n)$ time in the worst case. In the worst case, it iterates through all elements in both arrays once.

Since the dominant factor in the time complexity is the sorting step, the overall time complexity of the algorithm is $O(n \log n)$. $T(n) = \mathcal{O}(nlogn) + \mathcal{O}(n) = \mathcal{O}(nlogn)$

So, the algorithm is both correct and has a time complexity is $O(n \log n)$.