Homework #2
Introduction to Algorithms/Algorithms 1
600.363/463
Spring 2013

Yifan Ge

February 12, 2013

1 Problem 1

1.1 Algorithm description

1. Use *Quicksort* to sort the array A, this can be done in $O(n\log(n))$ as the array is of size n.

2. Since A is sorted, duplicated elements are now consecutive. Traverse A to count the length of each part of duplicated elements. Add the cube of the length to a variable S (S is initialized to be 0 at the beginning of this step). After the traversing is done, S is the third frequency moment we need.
   As A is traversed only once, this step can be done in $O(n)$. As a result, the whole algorithm works in $O(n\log(n))$ time.

1.2 Pseudocode

Algorithm 1.1. *THIRD_FREQUENCY_MOMENT(A)*

```
1: A ← Quicksort(A)
2: i ← 1
3: j ← 1
4: S ← 0
5: for j ← 2 to n do
6:     if A[i] != A[j] then
7:         S ← S + (j − i)^3
8:         i ← j
```

9: ***end if***
10: ***end for***
11: $S = S + (n - i + 1)^3$
12: ***return*** S

1.3 Proof of correctness

We prove the correctness of this algorithm by induction.
First, for the base case that $n = 0$, the loop from line 5 to line 10 will not get executed. Line 11 computes the answer, which is 0.
Assume the algorithm is correct for all the $k_1 < k_2$, we want to prove the correctness holds for $n = k_2$. Since the array is sorted, all the same integers are consecutive. We assume there are c ds at the end of the array and there are no other ds elsewhere in the array. As the algorithm is correct for all the $k_1 < k_2$, it is correct for the case that $n = k_2 - c$. When the loop from line 5 to line 10 ends, S holds the third frequency moment of subarray $A[1..k_2 - c]$, and $i = k_2 - c + 1$. Line 11 adds c^2 to S, which is the third frequency moment of the whole array of integers according to the definition.

2 Problem 2

2.1 Algorithm description

1. For both of the tasks, first use *Quicksort* to sort A and B respectively. This step works in $O(n \log(n))$ time.

2. - To find all numbers that appear both in A and B, use a strategy similar to the merge operation in *Mergesort*. We use two indices i and j starting from the beginning of A and B. If $A[i] < B[j]$, we add i by 1; if $A[i] > B[j]$, we add j by 1; if $A[i] = B[j]$, we increase both i and j by 1 and record the number $A[i]$ as it appears both in A and B. The algorithm ends when either i or j grows beyond n. In this step, A and B are looped separately, so the time complexity is $O(n)$.
   - To find the numbers that only appear in A, we still use i and j to loop through A and B. What differs from the first algorithm is, we record $A[i]$ when $A[i] < B[j]$. Also, after this step ends with $i < n$, the remaining integers in A $A[i + 1..n]$ all only appear in A and should be recorded. Finding numbers only appearing in B is symmetric, we record $B[j]$ when $A[i] > B[j]$, and also need to record the remaining integers in B after the step ends. This step works in $O(n)$ time.

As a result, the total time complexity of the algorithms for the two tasks are both $O(n \log(n))$.

2.2 Pseudocode for task 1

Algorithm 2.1. *COMMON_ELEMENTS(A, B)*

1: $A \leftarrow quicksort(A)$
2: $B \leftarrow quicksort(B)$
3: $i \leftarrow 1$
4: $j \leftarrow 1$
5: $count \leftarrow 0$
6: $r \leftarrow []$
7: ***while*** $i \leq n$ ***and*** $j \leq n$ ***do***
8: ***if*** $A[i] < B[j]$ ***then***
9: $i \leftarrow i + 1$
10: ***else if*** $A[i] > B[j]$ ***then***
11: $j \leftarrow j + 1$
12: ***else***
13: $count \leftarrow count + 1$
14: $r[count] \leftarrow A[i]$
15: $i \leftarrow i + 1$
16: $j \leftarrow j + 1$
17: ***end if***
18: ***end while***
19: ***return*** r

2.3 Pseudocode for task 2 (finding numbers only appearing in A)

Algorithm 2.2. *ELEMENTS_IN_A(A, B)*

1: $A \leftarrow quicksort(A)$
2: $B \leftarrow quicksort(B)$
3: $A[n + 1] \leftarrow \infty$
4: $B[n + 1] \leftarrow \infty$
5: $i \leftarrow 1$
6: $j \leftarrow 1$
7: $count \leftarrow 0$
8: $r \leftarrow []$
9: ***while*** $i \leq n$ ***and*** $j \leq n$ ***do***

```
10:    if A[i] < B[j] then
11:        count ← count + 1
12:        r[count] ← A[i]
13:        i ← i + 1
14:    else if A[i] > B[j] then
15:        j ← j + 1
16:    else
17:        i ← i + 1
18:        j ← j + 1
19:    end if
20: end while
21: while i ≤ n do
22:    count ← count + 1
23:    r[count] ← A[i]
24: end while
25: return r
```

2.4 Proof of correctness

2.4.1 Task1

We argue that, at the beginning of each iteration of the loop from line 7 to line 17, all the common elements of A and B less than $minA[i], B[j]$ have been stored in r. We view this as the loop invariant.

Initialization:

Prior to the first iteration of the loop, we have $i = j = 1$. As there is no number existing in the two arrays which is less than $minA[i], B[j]$, the loop invariant holds.

Maintenance:

For each iteration, there are 3 possible operations.

For the case that $A[i] = B[j]$, as all the common numbers less than $minA[i], B[j]$ have been in c, and both A and B are sorted, after recording $A[i]$ and increase i and j by 1, the loop invariant holds. For the case that $A[i] < B[j]$ and $A[i] > B[j]$, the loop invariant still holds.

Termination:

The loop terminates when either i or j exceeds n. As the largest numbers in A and B are the sentinels (∞), all the common numbers less than $minA[i], B[j]$ are all recorded. And since $minA[i], B[j]$ is either the largest number of A or the largest number of B, all the common number of A and B have been recorded.

2.4.2 Task2

- Case 1: The loop from line 9 to line 20 ends with $i > n$.
  In this case, all the elements in A have been looped through. For each elements in A, there are only two possibilities: it can be a common element of A and B, or it can be an element only appearing in A. As we are now doing the opposite thing for each elements in A in this algorithm to what we did in the algorithm for task 1, plus we have proved the correctness of the algorithm for task 1, we conclude this algorithm is also correct.

- Case 2: The loop from line 9 to line 20 end with $i = m$ and $m \leq n$.
  We know that at the end of the loop from line 9 to line 20, the algorithm can get all the elements appearing only in $A[1..m]$ from the proof of case 1. And since $m \leq n$, the loop ends with $j > n$. So $A[m]$ is larger than the all the integers in $B[1..n]$. Because A is sorted, all the elements in $A[m..n]$ are larger than all the elements in $B[1..n]$, which means they can only appear in A. After add these remaining elements to r, the algorithm finds all the elements only appearing in A. Therefore the algorithm is correct.