Solutions
Homework #4
Introduction to Algorithms/Algorithms 1
600.363/463
Spring 2016

Due on: Thursday, Feb 25th, 11.59pm
Late submissions: will NOT be accepted
Format: Please start each problem on a new page.
Where to submit: On blackboard, under student assessment
Please type your answers; handwritten assignments will not be accepted.
To get full credit, your answers must be explained clearly,
with enough details and rigorous proofs.

March 6, 2016

1 Problem 1 (12 points)

You are given $k = O(1)$ unsorted integer arrays $A_1, A_2, \ldots, A_k$. Each array A_i contains n elements s.t. $A_i[j] \in \{1 \ldots n\}$. Another array C is defined as

$$C[j] = \prod_{i=1}^{k} A_i[j] = A_1[j] * A_2[j] * \ldots * A_k[j].$$

Write an algorithm that sorts array C in $O(n)$ time. Prove correctness and provide running time analysis.
For example, consider the case when $k = 3$ and $n = 4$. $A_1 = \{1, 0, 3, 2\}$, $A_2 = \{1, 1, 2, 1\}$ and $A_3 = \{3, 2, 4, 1\}$, then $C = \{3, 0, 24, 2\}$, and we want to sort C in $O(n)$ time.

Solution:
Let's first prove that we can use radix algorithm to sort integer array of size n with elements from the range $[1, \ldots, n^l]$ in linear time (only for $l = O(1)$). As we

know from the class running time of the radix sort is $O(d(n+k'))$ (we use count sort as a subroutine), where d is number of digits needed to represent the largest number in the array, n is the size of the array, and k' is largest value of one digit. Suppose we are given an array with all numbers in base 10 representation. If we will sort digit-by-digit, then $k' = 10$ and $d = \log_{10} n^l = O(\log n)$. Thus total running time is $O(d(n+k')) = O(n \log n)$, which is not linear. But if we will sort "super-digit"-by-"super-digit" , where "super-digit" is just r digits, that is we first apply count sort on last r digits, then on the second last r digits and so on. Then the total number of "super-digits" is equal to total number of digits divided by r : $d = \frac{\log_{10} n^l}{r} = O(\frac{\log_{10} n}{r})$, and maximum value of the one "super-digit" is 10^r. Thus our complexity is $O(\frac{\log_{10} n}{r}(n+10^r))$. Thus if we take $r = \log_{10} n$, we will get complexity $O(n)$. Thus we have shown that radix sort runs in linear time if all elements are from the range $[1, \ldots, n^l]$, where l is constant. Similarly we can prove the same fact using base n representation. In this case $k' = n$ and $d = l = O(1)$ and then running time is $O(d(n+k')) = O(n)$. Same can be shown for the base 2 representation which is more natural in real life.

Back to the problem.

Let us first create an array C. Doing so requires that we compute each element of $C[j]$ (i.e.$A_1[j] * A_2[j] * \ldots * A_k[j]$).

Based on the fact that $A_i[j] \in \{1 \ldots n\}$, we can conclude that $C[j] \in \{1 \ldots n^k\}$.

Now, let us perform the actual sorting operation. We apply radix sort and as we proved above it will run in linear time. Correctness of the radix sort was given in the class (or book).

2 Problem 2 (13 points)

You are given an array A of n integer. All numbers except for $O(\log n)$ are in the range $[1, 1000n^2 - n]$. Find an algorithm that sorts an array A in $O(n)$ time in the worst case. Provide running time analysis and correctness proof for your algorithm.

2.1 Solution

Our algorithm:

1. Create two arrays: array R and array M;
2. Go through array A and if $A[i]$ is not from the range (at most two comparisons), put it into array M, if it's from the range put it into the array R;

3. Sort array M using merge sort;

4. Sort array R using radix sort;

5. Merge two sorted arrays R and M back into array A;

6. output array A.

Correctness proof for radix sort, merge sort and merge step was given in the class (or book). In fact, due to given conditions every element will go either to array M or to array R, neither will go to both, thus $A = R \cup M$ and $R \cap M = \emptyset$. Thus if we sort R and M and merge them, then we get sorted A.

Second step of the algorithm takes $O(n)$ time, while running time for radix sort was proved to be linear in the first two paragraphs of the solution to the problem 1. Here we have $l = 2$, constants do not change anything (you can check it by substituting them into the given proof). Merge sort works in time $O(n' \log n')$, where n' is the size of the array M. But conditions of the problem state that $n' = O(\log n)$, thus running time of the merge sort is $T_M(n) = O(n' \log n') \rightarrow T_M(n) = O(n'^2) \rightarrow T_M(n) = O((\log n)^2) \rightarrow T_M(n) = O(n)$. Running time for the merge sort was also proved to be linear in the class. Thus total running time is $O(n)$.

3 Problem 3 (13 points)

Given an array A of n integers from the range $[1, m^3]$. A is stored as m pairs $(\text{item}, \#\text{ of instances})$. For example, if initially array was stored as

$$1, 2, 2, 3, 2, 3, 2, 1, 2, 5, 1, 5, 4, 3, 2, 1, 7, 2, 3, 6$$

then its new representation is:

$$(1, 4), (7, 1), (2, 7), (5, 2), (4, 1), (3, 4), (6, 1)$$

which you can read as item 1 appears 4 times in A, item 7 appears only once in A, item 2 appears 7 times in A, and so on. Provide an algorithm that finds k-th smallest integer in the array in $O(m)$ time with running time analysis and correctness proof for your algorithm.
Note, there is no dependency on n in time complexity.

```
KthSmallest(A, k)
A := radix(A[·][0]);
C := 0;
for i := 0 to m do
    C := C + A[i][1];
    if C >= k then
        return A[i][0];
```

3.1 Solution

The algorithm is as follows:

1. Sort the pairs by their first element using radix sort: **radix**$(A[.][0])$.
2. Once sorted, move from pair to pair in order, keeping track of the total number of items that have been passed over by maintaining a counter c which equals to sum of 2-nd entries seen so far. i.e. for each pair $p(a, b) = (A[i][0], A[i][1])$, maintain a running total of $p.b = A[i][1]$ entries. Call this as counter c.
3. After updating counter c with new pair $p(a, b)$, if $c < k$, then we can safely continue without having reached the k-th element.
4. Otherwise, the k-th element must be a member of the current pair. Return $p.b = A[i][0]$.

3.1.1 Running Time:

In the first two paragraphs of solution to the problem 1 we proved running time of the radix sort is linear for $l = 3 = O(1)$.Our own loop makes m iterations with $O(1)$ time each. Thus total time is $O(m)$.

3.1.2 Correctness:

First of all we will use next loop invariant:
$c = \#$ of elements in A less than $A[i + 1][0]$.
Proving initialization and maintenance is trivial.

Initialization:
Before the very first iteration $c = 0$, it is true that there is no elements in A less than $A[0][0]$, as long as A is sorted by first index.

Maintenance:
If c was equal to # of elements less than $A[i][0]$ after $i-1$ iteration, then after i-th iteration # of elements less than $A[i+1][0]$ is equal to # of elements less than $A[i][0]$ plus number of instances of $A[i][0]$, which is equal to $A[i][1]$. But we udpate c as $c+=A[i][1]$. Thus maintenence is proved.

Termination:
Algorithm returns $A[i][0]$ only when # of elements less than $A[i+1][0]$ is large than k while # of elements less than $A[i][0]$ is less than k. Thus k-th smallest number is equal to $A[i][0]$.

4 Problem 4 (12 points)

Given two integer arrays A of size n and B of size k, and knowing that all items in the array B are unique, provide the algorithm which finds indices j', j'', such that $B \subseteq A[j' : j'']$ and value $|j'' - j'|$ is minimized or returns zero if there is no such indices at all. For full credit, your algorithm must run in $O(nk)$ and use at most $O(n)$ of extra memory.

Algorithm

Intuitively, this problem involves checking each element of A to see if it's also in B, so we need to make searching in B fast, thus it's better to sort B at first in order to do a binary search. We also want to keep record of each element of B for whether it is in A so we need a counter to count the number of distinct elements of B already seen in A. Further more, we want to minimize the size of the continuous slice in A which contains all elements in B, so we need to record the rightmost position of each $B[i]$ in A, and check each time when we see another $B[i]$ in A if it's possible to shrink the length of the slice.
The pseudo code is shown below.

Proof of correctness

We first prove that the algorithm returns 0 if and only if A does not have all elements of B. Since we have an array C with each element corresponding to each $B[i]$ and a counter s that increments by 1 when each $B[i]$ is first seen in A, it's guaranteed that if $s < k$, A does not contain all the elements in B, and if $s == k$, A contains all elements in B. So the criteria for returning 0 is correct.

Second let's prove that the slice returned by the algorithm contains all elements of B. The first time when $j1$ is set, it is the index of the first $B[i]$ seen in A; and the

```
main(A[n],B[k])
if n<k then
    return 0   // A is even shorter than B
s=0   // the counter for counting distinct B[i]s in A
C[k]=[-1]*k   // the rightmost indices of B[i]s in A
Sort(B)   // can use quick sort or merge sort, although this step is not
  required for full credit.
j1=j2=-1   // initialize the two ends of the slice in A
for i in {0 to n-1} do
    j=lookInB(A[i])   // j=index of A[i] in B if found, otherwise -1
    if j==-1 then
        i++ break
    else
        if j1==-1 then
            j1=i   // the starting point of the first slice
        if C[j]==-1 then
            // The first time seeing B[j] in A
            s++
            if s==k then
                j2=i   // Found the first slice in A that contains B
        C[j]=i   // Let C[j] store the position of the newly found B[j] in A
        if j2>-1 then
            // After the first full slice is found
            x=min(C)   // return the smallest index in A of elements of B
            if abs(j2-j1)>abs(i-x) then
                // found a shorter slice, modify j1 and j2
                j1=x
                j2=i
if s<k then
    return 0
return j1, j2
```

```
lookInB(x) // We use binary search for sorted B
l1 = 0
l2 = len(B)-1
while l1<=l2 do
    mid = (l1+l2)//2
    if B[mid]==x then
        return mid
    else
        if x<B[mid] then
            l2=mid-1
        else
            l1=mid+1
return -1   // x is not in B

min(C)
a=+infinity
for i in {0 to len(C)-1} do
    if a>C[i] then
        a=C[i]
return a
```

first time when $j2$ is set, it is the index of the last $B[i]$ first seen in A. Since $j1$ has not changed yet, $A[j1 : j2]$ (including$A[j2]$) must contain all elements in B. After $j2$ is set ($j2 > -1$), we can modify $j1$ and $j2$ based on the length of the slice. Each time we change them, we make $j1$ be the smallest index of the elements in B, and $j2$ be the current index i, which is the maximum possible, so all B elements lie in between. Thus the slice returned by the algorithm contains all elements of B.

Last let's prove that the returned slice has the minimum possible length. Every time we modify $j1$ and $j2$, it requires that $abs(j2 - j1) > abs(i - x)$, that is, the new slice should be shorter than the old one, so the lengths of the recorded slices must decrease monotonically. Now we only need to prove that the shortest slice (or one of the shortest slices) must be recorded. Assume that this slice is $A[j1' : j2']$. When we iterate to $i = j2'$, it must be the case that $j1 < j1'$, $j2 < j2'$. Since $A[j1' : j2']$ contains all elements in B, the minimum value in C, x, must be contained in $[j1', j2']$, and since $A[j1' : j2']$ is the shortest slice, we must have $x == j1'$ otherwise we can make it shorter by letting $j1' = x$. And as we assumed, $abs(j2 - j1) > abs(j2' - j1')$, so according to our algorithm, we'll set $j1 = j1'$ and $j2 = j2'$, thus this shortest slice must be recorded. Proof done.

Analysis of complexity

Sorting B cost $O(kLogk)$ if we use merge sort. The outer loop in the main function is $O(n)$. Within the main loop, $lookInB(A[i])$ uses binary search so it costs $O(Logk)$ and $min(C)$ compares each element in C so it takes $O(k)$. All other steps take constant time. So the total running time is $O(kLogk) + O(n)(O(Logk) + O(k)) = O(nk)$ since $k < n$.

The extra space used besides A and B is the array C and some numbers, so it's $O(k) = O(n)$.