Homework #4
Introduction to Algorithms/Algorithms 1
600.463
Spring 2016

Submitted By: Sindhuula Selvaraju
JHED ID:sselvar4
In Collaboratin With: Anchit Pandya, Adhiraj Yadav

February 26, 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 $n = 4$. $A_1 = \{1, 1, 3, 2\}$, $A_2 = \{1, 1, 2, 1\}$ and $A_3 = \{3, 2, 4, 1\}$, then $C = \{3, 2, 24, 2\}$, and we want to sort C in $O(n)$ time.

ANSWER: **Algorithm:** The algorithm for sorting C is:
Where, C is the input unsorted array and ratio is the ratio $\frac{b}{r}$ that radix sort takes in as a parameter.

Algorithm 1: Sorting Array C in $O(n)$ time.

Input : Unsorted array C
Output: Sorted array C

```
function sortArrayC (C[])
{
        RadixSort(C,ratio)
}
```

Proof of Correctness:

This algorithm is correct because Radix sort has been proven to be sound and complete(Proof can be found here).

The maximum number of values in C is dependent on n. Also, $A_i[j] \in \{1 \ldots n\}$ i.e. maximum value for any $A_i[j]$ will be n. Both these conditions make it ideal to use Radix Sort algoritm to sort C.

Running Time Analysis:

Radix sort is used to sort C

The number of arrays in A are linear in k and can be assumed to be constant

Each $A_i[j] \in \{1 \ldots n\}$ and can be sorted using counting sort

(I think this is not part of the question):

The formulation of C involves k multiplications for n elements in each $A_i[j]$. But as k is a constant, C can be formed in linear time $O(n)$

Runtime Analysis of Sorting:

We know that the running time complexity of Radix sort can be given by $O(\frac{b}{r}(n + 2^r))$

Thus, in our case if we substitute r by log n

The value of b is $klogn$.

Thus, if we make the substitutions,we get

T(n) = $O(k(n + n))$ = $O(2kn)$

Ignoring constants we get:

T(n) = $O(n)$

Hence Proved.

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.

ANSWER: **Algorithm:** The algorithm for sorting A is:

Algorithm 2: Sorting Array A in $O(n)$ time.

Input : Unsorted array A
Output: Sorted array A

function sortArrayA $(A[])$
{
for i = 0 to n
{
if $A[i] < 1 \; OR \; A[i] > 100n^2 - n$ **then**
    Insert $A[i]$ into B
else
    Insert $A[i]$ into C;
end
} RadixSort(C) MergeSort(B) Merge(B,C)
}

Proof Of Correctness:
Assume the correctness of Radix Sort, Merge Sort and the Merge Function(from Merge Sort)
The largest value of C can be represented as a function of n i.e $1000n^2 - n$. Hence, it is a case that can be evaluated using a stable sort algorithm. So it we can use Radix sort to sort C.
Since every element is either being classified into either C or B, the combination of B and C will have all elements from A and $B \cap C = \emptyset$

Running Time Analysis:
We first split the A into 2 where B is of size O(\$$logn$)(i.e. the numbers that are not in range $[1, 1000n^2 - n]$) and C is of size n-O($\log n$)
As each element of A is individually checked and places in B or C this can be done in O(n) time.
Applying Radix sort algorithm on C:

We know radix sort gives us a complexity of $O(\frac{b}{r}(n+2^r))$
The largest element here k=$1000n^2 - n$
Thus, for k=$1000n^2 - n$, we have b= $\log_2 k$
Thus, b= $\log 1000n^2 - n$
Substituting these values we get:
$T_C(n) = \log(1000n^2 - n)$
$T_C(n) = \log(1000(n^2 - \frac{n}{1000}))$
$T_C(n) = \log(1000) + \log(n^2 - \frac{n}{1000})$
Ignoring the constant term
$T_C(n) = \log(n^2 - \frac{n}{1000})$
$T_C(n) = \log(n^2 + \frac{-n}{1000})$
$\log(a+b) = \log(b) + \log(1 + \frac{b}{a})$ As answered here.
$T_C(n) = \log(n^2) + \log(\frac{\frac{-n}{1000}}{n^2})$
$T_C(n) = 2\log(n) + \log(\frac{-1}{1000n})$
For a sufficiently large value of n $\log(\frac{-1}{1000n})$ will have a very small value.
$T_C(n) = 2\log(n) = O(n)$
We know that the worst case running time complexity of merge sort is O($n \log n$),
Thus, in this case the O($\log n$) elements in B will be sorted in $T_B(n) = O(\log n \log(\log n))$ time.
The only remaining task is merging these sub arrays which takes $O(n)$ time.
Thus:
$T_A(n) = O(n) + O(\log n \log(\log n)) + O(n) = O(n)$
Hence Proved

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.

ANSWER: **Algorithm:** The algorithm for finding k^{th} smallest element in A:
k1 is a counter that stores the number of values read so far
The flag is used to check if the k^{th} element exists in the list
A is stored in the form of key value pairs that can be iterated over

Algorithm 3: Finding k^{th} smallest element in A

Input : Array A with (element,count) pairs

Output: $k^{th} element of A$

function findKth $(A[], k)$
$\{$
RadixSort(A);
$k_1 = 0$;$flag = 0$; foreach k_2 in sizeof(A) $\{$ $k_1 += k_2$;
if $k_1 \geq k$ **then**
    print A[k_2];
    $flag = 1$;
    break;
else
    continue;
end
if $flag == 0$ **then**
    print ”not in list”;
else
    continue;
end
$\}$
$\}$

Proof of Correctness:

As there are m pairwise elements in the array and the maximum value of the array is of the form m^3 and can be evaluated using a stable sort algorithm we can therefore use radix sort here.
Array A is of the form :
((k1,v1),(k2,v2),......(k_m, v_m))
If the kth smallest element we want to find lies between t_n and t_{n+1} i.e $k_n < k <= t_n + 1$ where t_i denotes the sum value of all v_i in A up till the ith element,i.e $t_i = \sum v_i, where\ v_i \in A$ then the kth smallest element corresponds to k_{n+1} in A where k_i corresponds to the key till that iteration.
If at the end the element is still not found a not found message is displayed.
Proving by Induction:

Base Case:

Assume k$\leq v_1$
$\implies$ t$_1 = v_1$
$\implies$ k$\leq t_1$
Thus in this case the k^{th} smallest element corresponds to the first key k_1

Induction Hypothesis:

Assume k = r and that $t_{n-1} <$ k $\leq t_n$
hence the kth smallest element in this case would be k_n

Induction Step:

Let the k = r+1
There are 2 possible cases:
Case 1:
$t_{n-1} <$ k $\leq t_n$
From the induction hypothesis the k^{th} smallest element is the nth key in array A ie

Case 2:
$t_n <$ k $\leq t_{n+1}$
And thus in this case the k^{th} smallest element will correspond to the $n + 1^{st}$ key in array A i.e k_{n+1}

Thus, we have successfully proved that for any value k,
if $t_{n-1} <$ k $\leq t_n$
then the nth key in array A is the k^{th} smallest element

Hence Proved.

Running time Analysis:
The first part involves Radix Sort.
There are m pairwise elements in the array and the largest key value of that array is of the form (m^l, m^3) is $A[m] \leq m^3$ in this case and hence Radix sort can be used.
We know that the running time complexity of Radix sort can be given by $O(\frac{b}{r}(m+2^r))$
r can be substituted by $\log m$
The value of b is $k \log m$.
Thus, if we make the substitutions,we get
$T_1(n) = O(k(m+m)) = O(2km)$
Ignoring constants
$T_1(n) = O(m)$
In the second part we have the loop that iterates over every key in A to calculate the of the values up to that point.
The loop will have maximum m passes, the complexity to sum up and compare with k will be $T_2 = O(m+m) = O(2m) = O(m)$
Thus, on adding the two parts we get:
$T(n) = O(m)$ +$O(m)$ =$O(m)$

Hence Proved

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 all elements of B belong to $A[j' : j'']$ and value $|j'' - j'|$ is minimized or returns zero if there is no such indices at all.
For example, consider array $A = \{1, 2, 9, 6, 7, 8, 1, 0, 0, 6\}$ and $B\,\{1, 8, 6\}$, then you can see that $B \subseteq A[1 : 6]$ and $B \subseteq A[4 : 7]$, but at the same time $7-4 < 6-1$, thus algorithm should output $j' = 4$ and $j'' = 7$.
For full credit, your algorithm must run in $O(nk)$ and use at most $O(n)$ of extra memory. Prove correctness and provide running time analysis.

ANSWER: Algorithm:
The algorithm for finding j' and j" is: NOTE :

1. Assume that binary search algorithm returns the position at which the element is found or else returns -1
2. Assume C is java equivalent of HashMap¡Integer,Integer¿ C[] = new HashMap¡Integer,Integer¿[k]

```
function findjs (A[], B[])
{
MergeSort(B)
C = []                    # C is of size k
diff = 0;
minj = 0;
maxj = 0;
k = 0;
foreach i in sizeof(A)
{
j = BinarySearch(B,A[i]);
if j ≠ −1 then
    C[j] = (A[i],i);
    k += 1
    if k ≥ sizeof(B) then
        (diff,minj,maxj) = evaluate(C,diff);
    else
        continue;
    end
else
    continue;
end
}
} if k¡sizeof(B) then
    return 0;
else
    print j' = minj, j" = maxj and difference = diff;
end
}
function evaluate(C[], diff) { min = 0; max = 0; diff2 = 0; foreach (k,v) in
C { if v < min then
    min = v
else
    continue;
end
if v > max then
    max = v
else
    continue;
end
}
diff2 = (min − max)
if diff2≤ diff then
    return(diff2,min,max)
else
    return(diff,min,max);
end
}
```

Proof of Correctness:

We know the complexities of Binary search and mergesort
Array C will have k elements only if all elements of B have occured atleast once while looping over A.
On the occurence of elements of B i.e. when C is full, the evaluate function is called and will compute the highest and lowest key values in the key-value pair A and also gives the difference.
For every successful search of B[j] in A, the corresponding C[j] is updated and evaluate function is called again.
After the loop on A is completed the values of minimum and maximum indices are returned.
If at the end of the loop if the length of C is not k, we return 0.
Thus, since every subset is considered once at a time by analysing every possible subset of size k of array B in A.
This makes sure that all elements of B have a corresponding value in C before the evaluate funcion is called and so no elements of B can be missed out.

Example:
Assuming indices start from 1
A=[1,8,6,2,1]
B=[1,8,6]
Sorted B =[1,6,8]
Now for case 1:
The loop variable has iterated till A[3].
Since all elements of B have occurred at least once, the size of C is 3.
Now, C is (1,1),(6,3),(8,2)
The maxj value is 3 and minj is 1 and the diff value is 2
When the loop iterates to A[5], i.e 1, the value in C updates to
C is (1,5),(6,3),(8,2)
The new maxj is 5, the new minj is 2 and the difference is 3.
Since the previous difference was lower the program will return j' = 1, j" = 3 and diff = 2

Running Time Analysis:
Merge sort to sort array B takes$O(k \log k)$ time
Searching for A[i] in B[j] using binary search leads to $\emptyset(logk)$ for 1 element.
Evaluate function takes $O(\log k)$ time each for finding minimum,maximum and constant time for finding the difference and returning the values c
$T_{evaluate}(k) = O(\log k) + O(\log k) + c = O(\log k + c)$

Since these complexities are multiplied with the looping term n, it will not exceed nk.
Thus, the total time complexity will be
T(n) = $k \log k + n.(logk + k + c) = O(nk)$

Space Complexity:
The only additional space needed is to store array C of size k and constant space to store highest,lowest and minimum distance.
$S(n) = O(k)$
as $k \leq n$
$S(n) = O(n)$
Hence Proved.