Intro to Algorithms HW4

Jiayao Wu jwu86

February 24th 2016

1 Problem 1

Algorithm overview:

Basically, we run the radix sort on array C.

1. First compute array C using the formula given and we know the largest possible number is n^k.

2. Assume $C[i]$ can be represented by b digits and each group of digits within $C[i]$ has r digits, so the total number of groups is $\frac{b}{r}$. Then, we apply counting sort from the right most group to the left most group one by one. In the end, array C is sorted.

Here is the pseudo code:

For j = 1, 2...n

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

For each group of $\frac{b}{r}$ digits from right to left

run CountingSort on this group

Prove correctness:

We can use induction to prove it. When there is only one group where $m = \frac{b}{r} = 1$, then we know array C is sorted because we are using a sorting algorithm on it.

We suppose the algorithm works for m groups of digits. We need to prove it also works for m + 1 groups. Suppose, after we already applied counting sort on m and $m + 1$ groups, $\exists$ two numbers c_i and c_j in $m + 1$ group and $\exists$ two numbers b_i and b_j in m group where $i < j$. There are two cases:

Case 1: if $c_i < c_j$, then we know array C is sorted given $i < j$.

Case 2: if $c_i = c_j$, but at the same time, we know $b_i < b_j$, so array C is still sorted.

The case of $c_i > c_j$ is impossible since we already applied sorting algorithm on $m + 1$ group.

Therefore, we conclude that the radix sort algorithm works correctly.

Running time analysis:

The largest number in array C is n^k because the largest number in each array A_i is n and according to the definition of C, we are multiplying k times.

Then, the number of digits $b = \log_{10} n^k = k \log_{10} n$. The largest number in each group would be 10^r. Also, the cost of computing array C is $O(n * k) = O(n)$.

Since we know, for Radix Sort, the running time is $O(m(n + k))$ where m is the number of group of digits, n is the size of the array and k is the largest number in each group of digits. Therefore, $O(m(n + k)) = O(\frac{b}{r}(n + 10^r))$. Let 10^r be $\theta(n)$, so $r = \log_{10} n$. Substitute this into the previous formula, we get: $O(\frac{b}{r}(n + 10^r)) = O(\frac{bn}{\log_{10} n}) = O(kn) = O(n)$ since $b = k \log_{10} n$ and $k = O(1)$.

Therefore, it is $O(n) + O(n) = O(n)$.

2 Problem 2

Algorithm overview:

We apply two separate sorting algorithms here. One for numbers in the range $[1, 1000n^2 - n]$ which is radix sort and another for $O(\log n)$ numbers which we can use merge sort.

1. Separate array A in two parts (create 2 new arrays). One part has the range $[1, 1000n^2 - n]$ and we call it A_1. The other part is the remaining $\log n$ numbers and we call it A_2.

2. Run Radix sort on array A_1. Suppose the item in $A_1[i]$, where i is from 1 to $1000n^2 - n$, can be represented by b digits. Assume there are r digits in each group of digits in the number, so the total number of groups in each item is $\frac{b}{r}$. Then, we apply counting sort on A_1 from the right most group of digits to the left most group of digits one by one. In the end, A_1 is sorted.

3. Apply merge sort on array A_2.

4. Merge the resulted array A_1 and A_2 back into array A.

Here is the pseudo code:

For each group of $\frac{b}{r}$ digits in items from right to left in array A_1 {:
    run CountingSort on this group
}
Run mergeSort(A_2)
Merge(A_1, A_2) into array A

Prove correctness:

First, prove radix sort works. We can use induction to prove it. When there is only one group where $m = \frac{b}{r} = 1$, then we know it is sorted because we are using a sorting algorithm on it.

We suppose the algorithm works for m groups of digits. We need to prove it also works for m + 1 groups. Suppose, after we already applied counting sort on m and $m + 1$ groups, $\exists$ two numbers c_i and c_j in $m + 1$ group and $\exists$ two numbers b_i and b_j in m group where $i < j$. There are two cases:

Case 1: if $c_i < c_j$, then we know it is sorted given $i < j$.

Case 2: if $c_i = c_j$, but at the same time, we know $b_i < b_j$, so it is still sorted.

The case of $c_i > c_j$ is impossible since we already applied sorting algorithm on $m + 1$ group.

Therefore, we conclude that the radix sort algorithm works correctly.

Since we already know that merge sort correctly sorts an array, so the group of $\log n$ numbers is sorted afterwards. The merge process also correctly works because we have learned this in class.

Therefore, since each part of the algorithm works correctly, so it works correctly in the end to sort array A.

Running time analysis:

First, the process of copying array A into 2 arrays is $O(n)$ because we have n items in total. Then, let's analyze run time for radix sort. The largest number in

array A_1 in $[1, 1000n^2 - n]$ is $1000n^2 - n = O(1000n^2)$ which is the upperbound. Then, the number of digits $b = \log_{10} 1000n^2 = 3 + \log_{10} n^2 = 3 + 2\log_{10} n$. The largest number in each group would be 10^r.

Since we know, for Radix Sort, the running time is $O(m(n + k))$ where m is the number of group of digits, n is the size of the array and k is the largest number in each group of digits. Therefore, $O(m(n+k)) = O(\frac{b}{r}(n+10^r))$. Even though the number of array for this part is $n - \log n$, but we can still consider it as n because n dominates. Let 10^r be $\theta(n)$, so $r = \log_{10} n$. Substitute this into the previous formula, we get: $O(\frac{b}{r}(n + 10^r)) = O(\frac{bn}{\log_{10} n}) = O(2n) = O(n)$ since $b = 3 + 2\log_{10} n$.

The third part is to run merge sort on the remaining $\log n$ numbers, so the run time would be $O(\log n \log\log n)$ which is clearly $< O(n)$.

In the final merge step, it will take $O(n)$ time because we have n numbers in total.

Therefore, the total running time is $O(n)+O(\log n \log\log n)+O(n) = O(n)$.

3 Problem 3

Algorithm overview:

We run the radix sort on the numbers from the m pairs in array A. Then as array A is sorted, start from lowest number, we add up the number of instances of the number until we get k and the corresponding number will be the $k-th$ smallest number.

1. Run Radix sort on array A to sort the items in the pairs.Suppose the item in $A[i]$, where i is from 1 to m, can be represented by b digits. Assume there are r digits in each group of digits in the items, so the total number of groups in each item is $\frac{b}{r}$. Then, we apply counting sort on A from the right most group of digits to the left most group of digits one by one. In the end, array A is sorted on the items.

2. Let $c = 0$, start from $A[i]$, we increment c by the number of instances in $A[i]$ until $c \geq k$. Then, return the corresponding item which is the $k-th$ smallest integer in the array.

Here is the pseudo code:
Algorithm: FindKthSmallest(A, b, r)
For each group of $\frac{b}{r}$ digits in items from right to left in array A {:
    run CountingSort on this group
}
Let $c = 0$, $i = 0$
while ($c < k$) {
    $c = c + A[i]$.instance
    $i++$
}
return $A[i]$.item

Prove correctness:

First, prove the sorting is correct. We can use induction to prove it. When there is only one group where $m = \frac{b}{r} = 1$, then we know array A is sorted because we are using a sorting algorithm on it.

We suppose the algorithm works for m groups of digits. We need to prove it also works for m + 1 groups. Suppose, after we already applied counting sort on m and $m+1$ groups, $\exists$ two numbers c_i and c_j in $m+1$ group and $\exists$ two numbers b_i and b_j in m group where $i < j$. There are two cases:

Case 1: if $c_i < c_j$, then we know array A is sorted given $i < j$.

Case 2: if $c_i = c_j$, but at the same time, we know $b_i < b_j$, so array A is still sorted.

The case of $c_i > c_j$ is impossible since we already applied sorting algorithm on $m+1$ group.

Therefore, we conclude that the radix sort algorithm works correctly.

Next, prove the number returned by the algorithm is the $k-th$ smallest number. As array A is already sorted, we know $A[0]$.item $\leq$ $A[1]$.item $\leq$ .... $\leq$ $A[m]$.item. Also, each item is associated with number of instances for

that item. As we increment variable c by $A[i]$.instance, we know c denotes the $c-th$ smallest number. Therefore, when $c \geq k$, the while loop terminates and we have found the $k-th$ smallest. Return $A[i]$.item because i represents where we are in the array.

Thus, this algorithm works correctly.

Running time analysis:

The largest number in array A is m^3. Then, the number of digits $b = \log_{10} m^3 = 3\log_{10} m$. The largest number in each group would be 10^r because each group has r digits.

Since we know, for Radix Sort, the running time is $O(m(n+k))$ where m is the number of group of digits, n is the size of the array and k is the largest number in each group of digits. Therefore, $O(m(n+k))$ in this case $= O(\frac{b}{r}(m+10^r))$. Let 10^r be $\theta(m)$, so $r = \log_{10} m$. Substitute this into the previous formula, we get: $O(\frac{b}{r}(m+10^r)) = O(\frac{bm}{\log_{10} m}) = O(3m) = O(m)$ since $b = 3\log_{10} m$.

Therefore, the radix sort runs in $O(m)$.

We know for the while loop, it runs in $O(m)$ because $k \leq m$.

Thus, in total, it runs $O(m) + O(m) = O(m)$.

4 Problem 4

Algorithm overview:

I used hash table to keep track of each window of array that contains B by going through array A once.

1. Create a hash table called C of size k. Since we know all the items in array B are unique, we can assume that each item has unique hash code and assume all k items can correspond to k spots in C through some hash codes. The values in hash table is initialized to 0. At the same time, create integers max, min and s all initialized to 0. s keeps track of the length of each window that contains all items in B. max and min corresponds to two end points of a window that contains all items in B.

2. Go through array A from the start. Check if the hash code of $A[i]$ is in the range 1 to k. If not, this means $A[i]$ is not in array B, we can move on to the next number in A. If hash code of $A[i]$ is in the range 1 to k, then we update the corresponding value in hash table with i, the index of $A[i]$. Keep doing so until there is no 0 in the hash table. If in the end, there is still some 0 in the hash table, return 0 because we know not all items in B are found in A.

3. By going through the hash table once, find the max and min values, put them into array max and min. Set $s = max$ - min.

4. Then continue going through array A. Each time, when we find any $A[i]$ that is in B, we update the corresponding value in hash table with i. Then repeat step 3 afterwards.

5. Repeat step 4 until we reach the end of array A.

6. In the end, return min and max which correspond to j' and j''.

Here is the pseudo code:

```
HashTable C = new HashTable of size k initialized to 0
int max, min, s, i, n = 0;
while (n < k) {
    If ( i == n + 1) }
        return 0
        //since the hash table still contains 0 even if we have run through A
once
    }
    if(hashCode[A[i]] not in the range of 1 to k) {
        i++
    } else {
        if (C[hashCode[A[i]]] = 0) n++
        C[hashCode[A[i]]] = i
        i++
    }
 }
 max = findMax(C), min = findMin(C), s = max - min
while (i ≠ n) {
    if(hashCode[A[i]] not in the range of 1 to k) {
```

$i++$
} else {
C[hashCode[$A[i]$]] = i
i++
int $maxNew$ = findMax(C), int $minNew$ = findMin(C)
int $sNew$ = $maxNew$ - $minNew$
if ($sNew < s$) replace max, min, s with $maxNew, minNew, sNew$
}
}
return min and max

Prove correctness:

First, prove the case of return 0 is correct. We return 0 when $i = n + 1$ which means that we have reached the end of array A, yet, we still have 0 in the hash table, which meas some values in B doesn't exist in A. Therefore, this is the right condition to return 0.

Next, prove the algorithm works in the case of returning j' and j''.

The first while loop enables us to fill the hash table with corresponding index for each item in B that's in A. It's done correctly. We only exit the while loop if each value in hash table is not 0.

Then, max and min are used to represent the end points of a certain window that contain all items in B. We know this is true because they are computed by traversing all the values in the hash table, and all other numbers are in the middle of min and max. Again, the values in the hash table represent the index of corresponding number in A. s is used to keep track of the smallest window so far.

In the second while loop, each time when we found $A[i]$ that's also in B, we replace the corresponding index and we know this window still contains the same contents because we only updated the index. Afterwards, we only update max, min, s if the newly calculated length of the window is smaller than s. In other words, we only update those values if we have found a smaller window that contains all the items in B. If we found a window that's equal or larger than previous one, we don't update the values. Therefore, in the end, max and min represent the end points of the smallest window and their difference is minimized. And returning them give the correct j' and j''.

Running time analysis:

Since each item in B is unique and also we know each item in B corresponds with one spot in the hash table, so make changes in the hash table is $O(1)$. In the first while loop where we change 0's in hash table with index, the running time is at most $O(n)$ because we at most need to go through array A once. Then computing max and min is $O(k)$ because we need to go through the hash table of size k to find them.

Then in the second while loop, the while loop itself at most executes $O(n-k)$ times since we already go through some part of the array in the first while loop. Inside the while loop, when we compute max and min, again, it's $O(k)$ times,

same reasoning as before.

Therefore, if we sum each part up, the running time is $O(n)+O(k)+O(k(n-k)) = O(nk)$.

As for the space complexity, since we only created a few variables and a hash table of size k, then the extra memory space we need is just $O(k) < O(n)$.