INTRO TO ALGORITHM, CSE 241

PROFESSOR ZHANG

Homework 3

Peiyun(Seed) Zeng

EN:437801

Due date: October 5, 2014

PROBLEM 1

Problem 3-1. The counting sort algorithm we saw in class uses $\Theta(n+k)$, since it created another array B to store the sorted results. We wish to reduce the space to $\Theta(k)$ (to keep the counts array) and sort in place. Here is some suggested code to replace lines 10–12 from page 195 in the textbook.

```
1   r ← A.length
2   while r ≠ 0
3       do
4           %Consider a = A[r] and gets its position in the final array
5           a ← A[r]
6           j ← C[a]
7           if j ≥ r
8             then C[a] ← C[a] − 1
9                  while j ≠ r
10                     do b ← A[j]
11                        k ← C[b]
12                        A[j] ← a
13                        a ← b, j ← k
14                        C[b] ← C[b] − 1
15                 A[j] ← a
16          r ← r − 1
```

(a):Does this algorithm sort correctly?If not,correct it.Either way,prove that the resulting counting sort (either this version or the corrected version) algorithm sorts correctly. You must formally prove that the loop terminates, and that on termination, the array A is sorted. (Hint: Prove the following loop invariant: At the start of each iteration, each element A[j] for j > r is either at its correct sorted position or must move left.)

Solution:
This algorithm is clearly correct. task left to prove the correctness of the algorithm, which is on termination, the array A is sorted.
Complete proof of correctness
the complete proof of correctness consists of two parts

1. **partial correctness:** that is on termination, the post condition is satisfied (A is sorted) via combining loop invariant(LI) and termination condition
    a) LI is true initially
    b) LI is true each iteration (the most important part of proof)
    c) LI combined with loop termination condition, post condition is satisfied
2. **termination:** the algorithm will indeed terminate

1. Proof of partial correctness
Let's first state the loop invariant for the algorithm is ***each element A[j] for j > r is either at its correct sorted position or must move left*** **(LI)**

(a): LI is true initially
Oh man, this is trivial. :) In the very beginning r = A.length. thus there is no A[j] for j > r. the LI is automatically satisfied

(b): LI is true at the end of any iteration
use proof by induction here to prove P(n): at any iteration of the loop, n, LH will hold at the end of iteration

- **Base case–P(1):**
  it is essentially the initial situation for LI. when r = A.length,there is no A[j] for j > r. the LI is automatically satisfied.

- **Inductive Hypothesis–P(m):**
  suppose that at the end of m iteration(when $r = r_0$), the LI holds that is each element A[j] for j > r is either at its correct sorted position or must move left
- **Goal Statement–P(m+1):**
  (we try to prove) that at the end of m+1 iteration, LI still holds
- **Inductive step–$\forall P(m) \rightarrow P(m+1)$:**
  we will prove, if IH holds, Goal statement must be true
  at the beginning of m+1 iteration, r is stepped down to $r = r_0 - 1$. Therefore we know for sure,according to IH, each element A[j] for $A > r + 1 = r_0$ must be in its correct position or needs to be moved left. what we try to prove is P(m+1): A[j] for j > r is either at its correct sorted position or must move left. apparently ***the only thing we need to prove then is A[r+1] must be either at the correct position or needs to be moved left***
  suppose at iteration m, if $j < r_0$, which means the correct position (got from C[a]) is to the left of the current position. then A[r] is either needs to be moved left, or it is fixed at right position by previous iterations. if $j \geq r_0$, then **snippet 7 to 15** in the pseudo-code will grantee that the element be swapped to A[r_0] is at correct. That is to say,combining both cases, if our IH is true, it is guaranteed that A[r_0] is either at its correct position or needs to be moved left.
  Thus, to summarize, we just proved two things

  a) $A > r + 1 = r_0$ must be in its correct position or needs to be moved left
  b) A[r_0] is either at its correct position or needs to be moved left.

  Apparently, combining a) and b) we have proved that P(m+1): A[j] for j > r is either at its correct sorted position or must move left.
- **Conclusion via Induction Principle:**
  We have proved that
  - P(1)
  - $\forall P(m) \rightarrow P(m+1)$

  according to the principle of induction, we can finally conclude that **P(n):at any iteration of the loop, n, LH will hold at the end of iteration**. therefore we proved that LI holds at end of any iteration

(c): LI combined with loop termination condition, post condition is satisfied

- LI states that each element A[j] for j > r is either at its correct sorted position or must move left
- at the termination of the loop, r = 0, so A[j] is essentially every element in the array.
- if array is unsorted, there exists some element must be moved to the right
- therefore, the array must be sorted to avoid **CONTRADICTION**

Combining (a), (b), (c) , we have proved the partial correctness of the algorithm

2. Proof of the Termination of the algorithm

since at each iteration, r is decreased by 1, so eventually the loop will exist at r = 0

3. Total correctness

we have proved

- the partial correctness of the algorithm
- the termination of the algorithm

Therefore, the total correctness of the algorithm is guaranteed

(b): What is the asymptotic running time of the above algorithm?Justify your answer.

Solution:

The worst case running time T(n) = O(n)

in this algorithm, guaranteed by snippet whenever an element e is moved, it is put in its final sorted position and is never moved again. accordingly, C[e] in the concatenated frequency table will be decreased by one. Therefore, there will be **no more than n swap** (to put e into the right position) happen in the whole process. therefore **T(n) = O(n)**. I hope every question is as short as this man (c): Can this routine be used as a subroutine for radix sort?

solution:

Apparently **NOT**. WHy?

code snippet 7 to 15 in the algorithm by swapping element in the array around grantees the correctness of the algorithm and also helps to reduce the space to 0(k). However, it sacrifices the stability of the algorithm. Unlike the original counting sort, this new version is **UNSTABLE**. Therefore, RADIX sort cannot employ this as a new subroutine since it requires a stable sorting algorithm for each column

INTRO TO ALGORITHM, CSE 241

PROFESSOR ZHANG

Homework 3

Peiyun(Seed) Zeng

EN:437801

Due date: October 5, 2014

PROBLEM 2

An array $A[1..n]$ has a majority element x if x appears in the more than $n/2$ times. Obviously, not all arrays have a majority element.
(a):Give an efficient algorithm to compute the majority element of the array, if it exists; return null no majority element exists.
Solution:
before we design the algorithm, let's be mathy for a second and prove this useful theorem. also to make life easier, let's say for array have even number of elements, the median is $\lfloor n/2 \rfloor$ at its sorted array

Definition 0.1. Define the majority element x of any list (array) as the the element, of which appearance in the array is more than $n/2$ times.

lemma 0.1. *if x exits, the median of the sorted array must be x*

Proof. prove Lemma 0.1 using contradiction
Suppose that x exists, yet it is not the median and x appears at position i(any appearance of x) of a sorted array, $i \neq \lfloor n/2 \rfloor$. so there are two cases $i > \lfloor n/2 \rfloor$ and $i < \lfloor n/2 \rfloor$

- if $i > \lfloor n/2 \rfloor$
  so x appears at i, that means there exists at least (n/2 + 1) x in the array. the right hand size of the sorted array is $\lceil n/2 \rceil$. if (n/2 + 1) x appears at right hand side, the median must also be x (the majority element), which **CONTRADICTS** our statement that median is not x.

- if if $i < \lfloor n/2 \rfloor$
  so x appears at i, that means there exists at least (n/2 + 1) x in the array. the left hand size of the sorted array is $\lfloor n/2 \rfloor$. if (n/2 + 1) x appears at right hand side, the median must also be x (the majority element), which **CONTRADICTS** our statement that median is not x.

contradiction arrives at the negation of the lemma 0.1. Therefore, Lemma 0.1 must be true. □

now we can start designing our algorithm through employing Lemma 0.1.
A simple algorithm can be, first use counting sort to sort the array. then simply find the median of the array m. then check the number of appearance a of the element m, if it is a is larger than n/2, return m, if not, return none
yet, this algorithm based on sorting is subject to the constrains of counting sort (such as extra space and small range of numbers). WE CAN DO BETTER by using order-statistics method
Algorithm find Majority element
Recall that, OS- Select algorithm(A, i) is the algorithm we discussed in the lecture that will be able to find the ith rank element in array A at a expected running time O(n) **High level Design**

1. use OS-Select (A,$\lfloor n/2 \rfloor$) to find the median of the array, m
2. traverse through the array to check the appearance(a) of m in array A
3. if a is bigger than $n/2$ return m
4. if bigger is less or equal to $n/2$ return None

Pseudo-code

```
Find_majority(A)
  1.   middle = Floor(n/2)
  2. m = OS-seclect(A,middle)
  3. count = 0
  4. for i <- 1 to A.length
  5.    if A[i] = m
  6.       count = count + 1
  7. if count > (n/2)
  8.    return m
  9. else return NONE
```

Proof of correctness

we are taking for granted that OS-Select (A, middle) will successfully return the median of the given array

So after loop (4-6), count will record the appearance of m in the array. then for 7-9, we just simply compare count with n/2. Now there are two cases

1. we return m if the appearance of it (count) if count is bigger than n/2 and it is the **CORRECT** majority element **according to definition 0.1**
2. if the median is unfortunately not the majority element for array A, **according to Lemma 0.1**, the majority element does not exist. so we return NONE, which is **also CORRECT**
   So we have proved the correctness of the algorithm

Time complexity

since OS-Select sucks at worst case and ends up in a O(n^2), which dominates other parts of the algorithm. so T(n) = O(n^2)

more interesting question is what is the expected running time $E[T(n)]$. OS-Select costs O(n) expected running time. for loop 4-7, it apparently takes a linear time and rest of the algorithm costs constant time. Therefore, in total **E[T(n)] = O(n)**

(b): Given an array, a 1/k-plurality element is one that appears more than n/k times in the array. Note that an array may have 0, 1 or several 1/k-plurality elements. Give an O(nk) running time algorithm that returns all the 1/k-plurality elements of an array of size n (or null if none exist).

Solution:

Definition 0.2. Define any element in array A[1...n] has appearance more than n/k as a plurality element

with very similar proof, we can derive Lemma 0.2

lemma 0.2. *any* $1/k$*-plurality element must have rank* $i * n/k$ *for each i in array A[1...n]*

again, this can be proved via an almost identical proof to that of lemma 0.1 now we can start designing our algorithm

High level design

- loop through i<- 1 to k, for each i, we OS-select(A, i*n/k) to get k
- check the appearance of k, if it is bigger than n/k, record it in array B. otherwise do nothing
- check the length of array B. return B if the length is non-zero. return NONE otherwise

Pseudo-Code:

```
Find_Plurality(A,k)
  1. B = []
  2. for i <- 1 to k
  3.     m = i*n/k
  4.     seed = OS-Select(A,m)
  5.     count = 0
  6.     for j <- 1 to A.length
  7.       if A[j] = seed
  8.       count = count +1
  9.     if count > n/k
  10.        B.append(A[i])
  11. if B.length != 0
  12.    return B
  14. else
  15.    return NONE
```

proof of correctness

in the outer loop, i will goes from 1 to k, then at 4, OS-select will find the element seed with rank i*n/k. now for 6-8 inner loop, we get the appearance(count) of seed in the array. if count is bigger than n/k, we record it since it is the plurality element **according to definition 0.2**. if count is less or equal to n/k, according to lemma 0.2, it is not a plurality element. when exits the outer loop, we exhaust all possibility of possible rank of element that is a plurality element.

for 11-15, we return none if B is empty (no plurality element). we return NONE if otherwise. For both situation, we have the **Correct** result.

Therefore we have proved **the correctness of the algorithm**

Time complexity

the outer loop runs k times. at each iteration is the same complexity as $Find_majority$ algorithm. so the total Expected running time for the algorithm is k times the complexity for each iteration, which is $\mathbf{E}[\mathbf{T}(\mathbf{n})] = \mathbf{O}(\mathbf{n} * \mathbf{k})$

INTRO TO ALGORITHM, CSE 241

PROFESSOR ZHANG

Homework 3

Peiyun(Seed) Zeng

EN:437801

Due date: October 5, 2014

PROBLEM 3

Problem 3-3. Weighted 3-Median

You are given an array X containing n objects, $o_1, o_2, ..., o_n$, each of which has two attributes ***value*** (object o_i has value x_i) and positive ***weight*** (object o_i has weight w_i). You may assume that all values are distinct, though the weights need not be. The total weight of all the objects together is $W = \sum_{i=1 \text{ to } n} w_i$

The ***weighted 3-median*** is the element o_k satisfying

$$\sum_{xi < xk} w_i \leq W/3$$

and

$$\sum_{xi > xk} w_i \leq 2W/3$$

Give an efficient algorithm to compute the weighted 3-median. For full credit, your algorithm should run in $\Theta(n)$ time. For partial credit, give a $\Theta(n \lg n)$ algorithm.

Solution:

The weighted 3- median algorithm

(a): Sort the array basing on the weight of the object (**we did not specify what sorting we use here, we will specify when when analyze the complexity of the algorithm**)

(b): loop through the array and calculate the sum $W = \sum_{i=1}^{n} w_i$

(c): now loop through the array again and calculate Sum $S = \sum_{i=1}^{n} xi$, inside the loop we have a probe x_j, where j = i + 1

(d): for each iteration of loop of (c), we calculate $temp = W - S - w_j$

(e): when $temp \leq 2W/3$ we return w_j as our 3 weighted 3-median element

Proof of correctness:

we take for granted that the sorting algorithm in (a) works correctly. so after (a) our array is sorted.through loop through the array, the sum of the weights can be easily calculated in (b).so after (b), W is correctly calculated.

from (c) to (d), we calculate $temp = W - S - w_j$. since $S = \sum_{i=1}^{n} xi$ and $j = i + 1$(so w_j is the next element to w_i in the sorted array). we are essentially checking whether $\sum x_i < x_k(w_i) \leq W/3$, when it is satisfied, since the array is sorted, $\sum x_i > x_k(w_i) \leq 2W/3$ will be also satisfied.

Then if both situations satisfied we return w_j as our weighted 3-median.

we have proved the correctness of the algorithm.

Time Complexity

The expected running time= $E[T(n)] = \Theta(n)$ Version.

(a): we use counting sort to sort the array, which costs a O(n) average running time.

(b)–(e): apparently the whole loop at worst, costs only a linear time since at each iteration, it only costs a constant time to calculate temp.

The average running time for the whole algorithm is then $\mathbf{E}[\mathbf{T}(\mathbf{n})] = \Theta(\mathbf{n})$

The worst case running time = $T(n) = \Theta(n * \log n)$ version

So basically we change our sorting algorithm in (a) to be merge sort, which costs a worst case $n \log n$ running time.

since the other parts are identical which cost O(n). The total worst case running time is then apparently = $T(n) = \Theta(n * \log n)$