CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2015

Solutions to Problem 3 of Homework 5 (12 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, October 13*

We will consider a variant of the counting sort algorithm that sorts an n element array A whose elements are integers between 1 and k.

(a) (4 pts) Recall the code of standard counting sort.

```
CountingSort($A, B, k,$)
1 let $C$ be a new array
2     For $i = 1$ to $k$
3         $C[i] = 0$
4     For $j = 1$ to $n$
5         $C[A[j]] = C[A[j]] + 1$
6     For $i = 1$ to $k$
7         $C[i] = C[i] + C[i-1]$
8     For $j = n$ to 1
9         $B[C[A[j]]] = A[j]$
10        $C[A[j]] = C[A[j]] - 1$
11 Return $B$
```

After the run of the counting sort, somebody accidentally erased both array A and B, but only left array C. Design an $O(n)$ procedure recoverBfromC that still reconstructs the sorted array B from C. Why does it only work when sorting integers?

Solution:

Psuedocode

```
Algorithm: RecoverBfromC(C, k)
B ← NewArray($n$)
for $j$ ← $k$ to 0 do
    if $j$ is 0 then
        while $C[j] \neq 0$ do
            $B[C[j]-1] = j$
            $C[j] = C[j]-1$
        end
    end
    else
        while $C[j] \neq C[j-1]$ do
            $B[C[j]-1] = j$
            $C[j] = C[j]-1$
        end
    end
end
Return B
```

Algorithm 2: Algorithm to recover B from C in $O(n)$ time

Counting Sort can be used only on integers because the values of A are used as the indices of C and as the indices can only be integers (and not floating point numbers).However, it is possible to use it on characters but they have to be mapped to integers so that they can be used as indices. □

(b) (4 pts) Now, consider the following variant of the Counting Sort algorithm that sorts the n-element array A whose elements are integers between 1 and k.

```
1   CountingSortFast($A, k, n$)
2       For $i = 1$ to $k$
3           $C[i] = 0$
4       For $j = 1$ to $n$
5           $C[A[j]] = C[A[j]] + 1$
6       $b = 1$
7       For $i = 1$ to $k$
8           For $j = 1$ to $C[i]$
9               $A[b] = i$
10              $b = b + 1$
11      Return $A$
```

Formally argue the correctness of CountingSortFast.

Solution:

Proof of Correctness

Base Case

$n = 1$, Let $A[1] = k$

The first two *for* loop sets $C[1 \dots k-1] = 0$ and $C[k] = 1$ The third *for* loop sets $A[1] = k$ and then returns A.

Therefore, A is sorted. Hence the base case true.

Induction Hypothesis

At the end of i^{th} iteration of the *for* loop, A is sorted until $\sum_{j=1}^{i} C[j]$ terms, i.e $A[0 \dots b_i]$ is sorted where b_i is the value of b at the end of i^{th} iteration

Induction Step

At the end of the i^{th} iteration, b is increased by one value. Now in the $(i+1)^{th}$ iteration, the value of b is one more than that of the previous iteration i.e $b_{i+1} = b_i + 1$

At the end of i^{th} iteration, $A[b_i] = i$. Therefore, in the beginning of $(i+1)^{th}$ iteration, $A[b_{i+1}] = i + 1 \implies A[b_i + 1] > A[b_i]$

By the induction hypothesis we already know that $A[0 \dots b_i]$ is sorted. Therefore, $A[0 \dots b_{i+1}]$ is sorted ($\because A[b_i + 1] > A[b_i]$)

Hence by Induction, we can conclude that the given algorithm is correct

$\square$

(c) (4 pts) Notice that while CountingSort makes in time $3n + 2k$ array assignments, but CountingSortFast makes only in time $2n + k$ array assignments. Further CountingSortFast does not use an extra array B used by sc CountingSort. Explain what the problem with CountingSortFast. In which realistic use cases would one prefer slightly slower CountingSort? Justify your answer.

Solution:

The CountingSort procedure has the property of being stable. i.e It always preserves the relative order of equal elements even after the sorting. If two elements $A[i]$ and $A[j]$ such that $A[i] = A[j]$ and $i < j$ then after the elements are sorted, $A[i]$ will appear before $A[j]$ in B. This is because we iterated through A backwards and we decrement $C[i]$ everytime we see i, the relative ordering of duplicate elements is preserved whereas CountingSortFast is not stable, the relative ordering of the same elements may or may not be preserved

For radix sort to work, the sorting aglorithm performed each time from the least significant digit to the most significant digit has to be stable. So if CountingSortFast is used, radix sort might not work

For example, consider that we have a list of first names and last names. Now our task is to sort these names by last name and then by the first. We could use a stable sorting algorithm on the first name. Now all the names are sorted by their first names. Next, we stable sort by the last names. After these two sorts, the names are primarily sorted by their last names. In case, the last names are same, they get sorted by their first names as the relative ordering is preserved. If we use CountingSortFast in this case, the names might not get sorted because it doesn't guarantee the relative ordering in the case of same last names

□