CMSC 351: CountingSort

Justin Wyss-Gallifent

March 27, 2024

1 What it Does

Sorts a list of (preferably small) nonnegative integers for which we know the maximum. This can be tweaked to do a bit more.

2 How it Works

Counting sort works by counting the number of each integer and creating a new list with that information.

For example consider this list:

`A`	3	10	4	3	4	5	4	3	5	6	1	3	0

Knowing that the maximum value is `10` we create an list `POS` with indices from 0 to 10 such that `POS[k]` contains the count of `k` in `A`. The name `POS` we're using may seem a bit weird but it will make sense soon.

index	0	1	2	3	4	5	6	7	8	9	10
`POS`	1	1	0	4	3	2	1	0	0	0	1
# of	0s	1s	2s	3s	4s	5s	6s	7s	8s	9s	10s

Next we modify `POS` above by replace it with its cumulative version, meaning we successively add each entry to the next.

index	0	1	2	3	4	5	6	7	8	9	10
`POS`	1	2	2	6	9	11	12	12	12	12	13

For a moment observe that this needs to give us the sorted version of `A` which looks like this:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
Sorted `A`	0	1	3	3	3	3	4	4	4	5	5	6	10

Observe that each `POS[i]==j` indicates that in the sorted version of the list if there are any occurrences of `i` then they end at position `j`, meaning index `j-1`.

For example `POS[3]==6` which means that in the sorted list the `3`s end at position 6, meaing index 5, and `POS[4]==9` which means that in the sorted list the `4`s end at position 9, meaning index 8.

Then we construct a new list `ANEW` by going through `A` in reverse order, taking each element, finding the position (and then index) to place it using `POS`, coping it over, then decreasing that entry in `POS` in preparation in case there's another value to come. So think about `A` as the input list, `POS` as the processing list, and `ANEW` as the output list.

We go in reverse order to make the sorting stable. Observe that duplicate values get filled into `ANEW` from right-to-left and so we should reverse through `A` to match that.

You may have wondered why we don't just use the first `POS` list to build a new list by simply going through `POS` and writing that many of each value. The reason is that we are typically thinking of the number in our list as keys which are associated with lots of data. This means we are not simply sorting numbers but rather we are moving around troves of information associated to those numbers. Consequently our creation of `ANEW` should be treated as though it is not just writing individual numbers but copying over that associated information.

Here are the first two steps:

`A[12]==0` (the value) and `POS[0]==1` (the position, one more than the index, to put it) so assign `ANEW[1-1]=0`...

`A`	3	10	4	3	4	5	4	3	5	6	1	3	0

index	0	1	2	3	4	5	6	7	8	9	10
`POS`	1	2	2	6	9	11	12	12	12	12	13

index	0	1	2	3	4	5	6	7	8	9	10	11	12
`ANEW`	0												

...and update `POS[0]=0` (ready for the next `0`, which there isn't one).

`A[11]==3` (the value) and `POS[3]==6` (the position, one more than the index, to put it) so assign `ANEW[6-1]=3`...

`A`	3	10	4	3	4	5	4	3	5	6	1	3	0

index	0	1	2	3	4	5	6	7	8	9	10
`POS`	1	2	2	6	9	11	12	12	12	12	13

index	0	1	2	3	4	5	6	7	8	9	10	11	12
`ANEW`	0					3							

...and update `POS[3]=5` (ready for the next `3`, which there is one).

And so on until we have filled up `ANEW`. Typically we then copy `ANEW` back to `A`.

The reason for going through `A` in reverse order is that it results in CountingSort being stable.

3 Pseudocode and Time Complexity

Here is the pseudocode:

```
\\ PRE: A is a list of length n with minimum 0 and maximum k
POS = list of zeros of length k+1
ANEW = list of zeros of length n
for i = 0 to n-1
    POS[A[i]] = POS[A[i]] + 1
end
for i = 1 to k
    POS[i] = POS[i] + POS[i-1]
end
for i = n-1 down to 0
    ANEW[POS[A[i]]-1] = A[i]
    POS[A[i]] = POS[A[i]] - 1
end
for i = 0 to n-1
    A[i] = ANEW[i]
end
\\ POST: A is sorted.
```

Note 3.0.1. If `A` is given but `k` is not we can simply add a simple loop to calculate it first since it's the maximum of the list. This process is $\Theta(n)$.

Best-, Worst-, and Average-Cases:

In most languages to allocate an empty list of length n it takes $\Theta(n)$ time, so our initial assignments of `POS` and `ANEW` take $\Theta(k)$ and $\Theta(n)$ respectively. Note that this is the first time that something other than n is showing in our time complexity.

We then iterate the four `for` loops which take time $\Theta(n)$, $\Theta(k)$, $\Theta(n)$, and $\Theta(n)$ respectively.

All together this is then $\Theta(n + k)$. This is the same for best-case, worst-case, and average-case.

If this is confusing, think of $n + k$ as a single variable, so what this means is that there are $B, C > 0$ and some index m_0 such that:

$$\text{If } n + k \geq m_0 \text{ then } B(n + k) \leq T(n, k) \leq C(n + k)$$

Note 3.0.2. If k is a fixed constant and we let n vary then the time complexity is $\Theta(n)$.

Note 3.0.3. If k is not fixed but we can guarantee that $k \leq n$ then the time complexity is $\Theta(n)$.

4 Auxiliary Space

CountingSort uses $\Theta(n+k)$ auxiliary space since it is required to create the `POS` list as well as the `ANEW` list.

5 Stability

Our particular pseudocode implementation of CountingSort is stable.

6 In-Place

CountingSort is not in-place.

7 Usage Note

CountingSort can be modified to manage other types of data. See the exercises for examples. It is often used as an auxiliary method inside other methods like RadixSort.

8 Thoughts, Problems, Ideas

1. Suppose $k_1, k_2 \in \mathbb{Z}$ with $k_1 < k_2$. Suppose every x in your list is an integer satisfying $k_1 \leq x \leq k_2$. Modify CountingSort so that it can manage this list. What would the Θ time complexity be?

2. Suppose $k_1, k_2 \in \mathbb{R}$ with $k_1 < k_2$ and both having at most one digit after the decimal point. Suppose every x in your list is a real number satisfying $k_1 \leq x \leq k_2$ with x having at most one digit after the decimal point. Modify CountingSort so that it can manage this list. What would the Θ time complexity be?

3. Suppose $d \in \mathbb{Z}$ with $d \geq 0$, $k_1, k_2 \in \mathbb{R}$ with $k_1 < k_2$ and both having at most d digits after the decimal point. Suppose every x in your list is a real number satisfying $k_1 \leq x \leq k_2$ with x having at most d digits after the decimal point. Modify CountingSort so that it can manage this list. What would the Θ time complexity be?

9 Python Test

Code:

```
import random
A = []
k = 5
n = 10
for i in range(0,n):
    A.append(random.randint(0,k))
print(A)
ANEW = [0] * n
LOC = [0] * (k+1)
for i in range(0,n):
    LOC[A[i]] = LOC[A[i]] + 1
print('Count array: '+str(LOC))
for i in range(1,k+1):
    LOC[i] = LOC[i] + LOC[i-1]
print('Cumulative count array: '+str(LOC))
for i in range(n-1,-1,-1):
    ANEW[LOC[A[i]]-1] = A[i]
    LOC[A[i]] = LOC[A[i]] - 1
    print('Positioning A['+str(i)+'] in position '+str(LOC[A[i]]-1))
    print('Result: '+str(ANEW))
for i in range(0,n):
    A[i] = ANEW[i]
print(A)
```

Output:

```
[3, 2, 0, 5, 2, 5, 0, 1, 2, 1]
Count list: [2, 2, 3, 1, 0, 2]
Cumulative count list: [2, 4, 7, 8, 8, 10]
Positioning A[9] in position 2
Result: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
Positioning A[8] in position 5
Result: [0, 0, 0, 1, 0, 0, 2, 0, 0, 0]
Positioning A[7] in position 1
Result: [0, 0, 1, 1, 0, 0, 2, 0, 0, 0]
Positioning A[6] in position 0
Result: [0, 0, 1, 1, 0, 0, 2, 0, 0, 0]
Positioning A[5] in position 8
Result: [0, 0, 1, 1, 0, 0, 2, 0, 0, 5]
Positioning A[4] in position 4
Result: [0, 0, 1, 1, 0, 2, 2, 0, 0, 5]
Positioning A[3] in position 7
Result: [0, 0, 1, 1, 0, 2, 2, 0, 5, 5]
Positioning A[2] in position -1
Result: [0, 0, 1, 1, 0, 2, 2, 0, 5, 5]
Positioning A[1] in position 3
Result: [0, 0, 1, 1, 2, 2, 2, 0, 5, 5]
Positioning A[0] in position 6
Result: [0, 0, 1, 1, 2, 2, 2, 3, 5, 5]
[0, 0, 1, 1, 2, 2, 2, 3, 5, 5]
```