Sorting in Linear Time

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Theory of Algorithms (CS 433)

We know that quick sort, merge sort, and heap sort all take $\Theta(n \log n)$ time to sort an array of n numbers. The question at this point is whether we can break the $n \log n$ barrier. Unfortunately, in the comparison model (where the only operation on two numbers allowed is comparison), the $\Theta(n \log n)$ time is optimal.[1] However, we know that the RAM model offers much more than simple comparisons between numbers. We shall see two non-comparison sorting algorithms that break the $n \log n$ barrier (under certain reasonable assumptions). However, we have the following restriction: **the data comprises of non-negative integers.**

1 Counting Sort

The first algorithm is the simpler of the two. We simply create an array $C[\,]$ of size $(1 + M)$, where $M = max_{0 \le i < n} A[i]$ is the maximum in the input array $A[0, n-1]$ that is to be sorted. At $C[A[i]]$, we store the count of $A[i]$, i.e., $C[A[i]]$ equals the number of times $A[i]$ appears in the array A. Finally, scan $C[\,]$ to report the numbers in A in sorted order. Clearly this algorithm has complexity $\Theta(n + M)$, which is $\Theta(n)$ as long as $M = O(n)$.

Here, we shall present a slightly different version of the algorithm, which is **stable**. Stability in a sorting algorithm guarantees that if $A[i] = A[j]$ for some $i < j$, then in the sorted order $A[i]$ will appear before $A[j]$.[2] Stability is not a property which the above straightforward implementation of Counting Sort guarantees. However, we will need stability for Radix Sort.

The main idea is to convert the $C[\,]$ array into storing cumulative frequencies, i.e., $C[i]$ will store the sum of the number of occurrences of $0, 1, \ldots, i$. Note that $C[i]$ essentially stores the frequency of numbers that are less than or equal to i.

For any index i, note that in the sorted order, $C[A[i]] - 1$ (not $C[A[i]]$ due to base-0 indexing) is the index of the rightmost occurrence of $A[i]$ in $A[\,]$. So, we simply scan $A[\,]$ from the right, and place $A[i]$ into the index $C[A[i]] - 1$ of the sorted array, following which we decrease $C[A[i]]$ by one (so as to correctly place the next occurrence of $A[i]$ in the appropriate index). Clearly this algorithm also has complexity $\Theta(n + M)$, which is $\Theta(n)$ as long as $M = O(n)$.

See next page for the algorithm and an illustration.

2 Radix Sort

The main problem with counting sort is that it begins to suffer as M grows in comparison to n. For example, when $n = 10^5$ and $M = 10^{15}$. What radix sort does is that it takes advantage of

[1] Check CLRS for the proof.

[2] Examples of stable sorting are MergeSort and InsertionSort. Unstable sorting algorithms are Quicksort and Heapsort; however, both can be made stable with some extra work (and extra space in case of heap sort).

Algorithm 1 A Stable Version of Counting Sort

```
function CountSort(int A[ ], int n)
    int M = maximum value in array $A$;
    create an array $C[\,]$ of length $(1 + M)$, and initialize all its cells to 0
    create an array $T[\,]$ of length $n$
    for (int i = 0; i < n; i++) do C[A[i]] = C[A[i]] + 1;
    for (int i = 1; i ≤ M; i++) do C[i] = C[i−1] + C[i];
    for (int i = n−1; i ≥ 0; i−−) do
        T[C[A[i]] − 1] = A[i];
        C[A[i]] = C[A[i]] − 1;
    use a loop to copy $T$ into $A$;
```

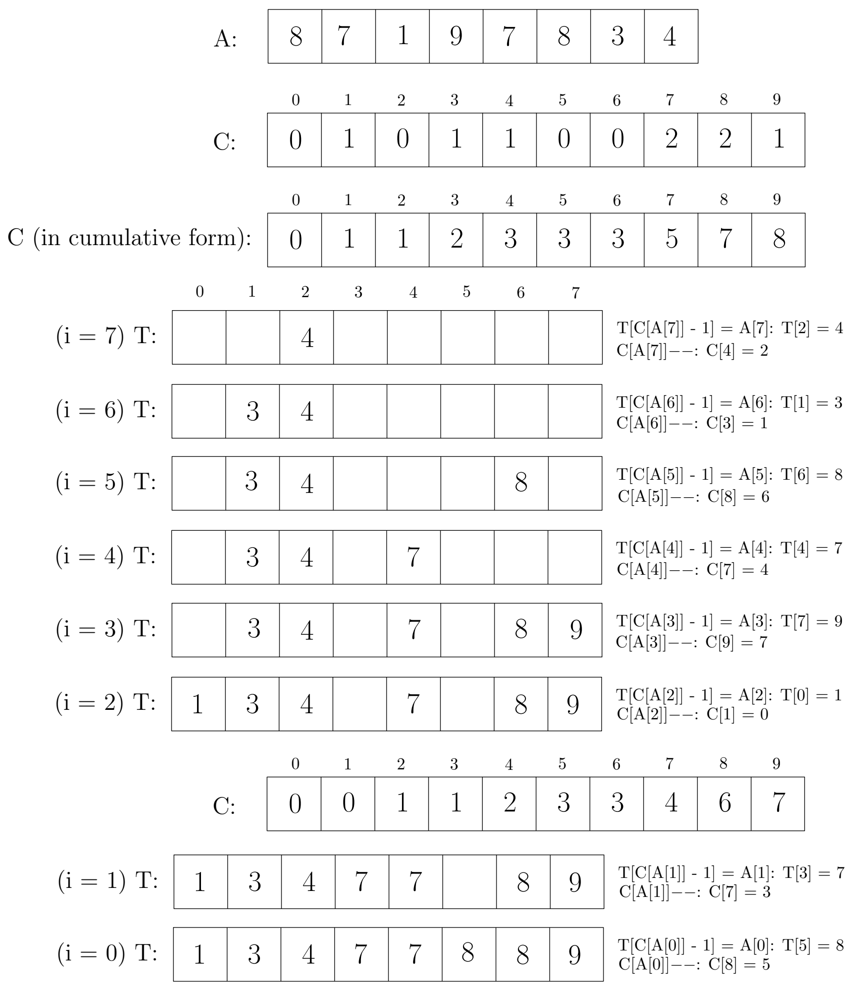

A: 8 7 1 9 7 8 3 4
C: 0 1 0 1 1 0 0 2 2 1
C (in cumulative form): 0 1 1 2 3 3 3 5 7 8
(i = 7) T:
T[C[A[7]] - 1] = A[7]: T[2] = 4
C[A[7]]−−: C[4] = 2
(i = 6) T:
T[C[A[6]] - 1] = A[6]: T[1] = 3
C[A[6]]−−: C[3] = 1
(i = 5) T:
T[C[A[5]] - 1] = A[5]: T[6] = 8
C[A[5]]−−: C[8] = 6
(i = 4) T:
T[C[A[4]] - 1] = A[4]: T[4] = 7
C[A[4]]−−: C[7] = 4
(i = 3) T:
T[C[A[3]] - 1] = A[3]: T[7] = 9
C[A[3]]−−: C[9] = 7
(i = 2) T:
T[C[A[2]] - 1] = A[2]: T[0] = 1
C[A[2]]−−: C[1] = 0
C: 0 0 1 1 2 3 3 4 6 7
(i = 1) T:
T[C[A[1]] - 1] = A[1]: T[3] = 7
C[A[1]]−−: C[7] = 3
(i = 0) T: 1 3 4 7 7 8 8 9
T[C[A[0]] - 1] = A[0]: T[5] = 8
C[A[0]]−−: C[8] = 5

Figure 1: Illustration of the stable Counting Sort algorithm.

the fact that even though M might be very large, the maximum number of digits in a number is still possibly not too high; for example, when $M = 10^{15}$, the maximum number of digits is 16. Radix Sort uses this idea and runs Counting Sort on each digit of the number starting from the least significant one. In other words, sort the numbers via their rightmost digit, then again sort the numbers via their last but rightmost digit, and so on, until you sort the numbers via their leftmost digit; here, to sort the digits, we use *Stable Counting Sort*. See next page for illustration.[3]

Proof of Correctness. We show that when radix sort has completed d' rounds of count sort, the numbers are sorted based on their last d' digits. We prove this by induction.

The base-case of the induction is when $d' = 1$. Obviously the numbers are sorted by their last digit at this point. By induction hypothesis, say when $d' = x$, the numbers are sorted by their last x digits. Consider the $(x+1)^{th}$ digit from right. For two numbers, if their $(x+1)^{th}$ digit is different, then count sort will place the number with the smaller of the two digits earlier, else if the digits are same, then since count sort is stable, the original order (based on the last x digits) will be maintained. Hence, the numbers will be sorted by their last $(x+1)$ digits, when $d' = (x+1)$.

The correctness follows by setting $d' = d$, i.e., the number of digits in the maximum number.

Complexity Analysis. A counting sort round takes $\Theta(n)$ time, as any digit is at most 9. If the maximum number has d digits, we need d rounds of counting sort, resulting in a $\Theta(d*n)$ algorithm.[4]

Pseudo-code. See the algorithm below.

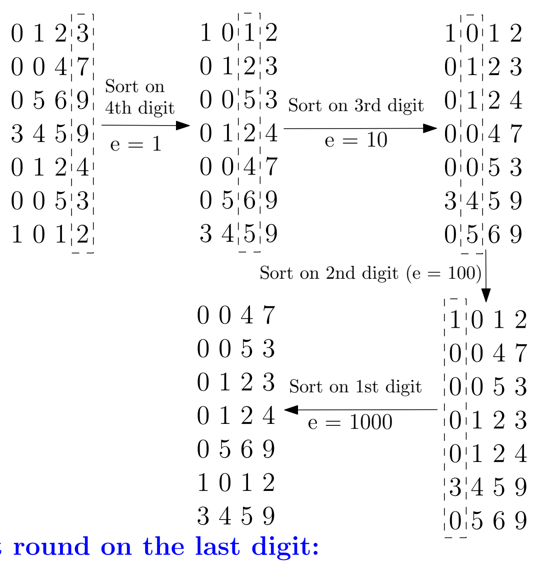
Algorithm 2 Base-10 Radix Sort Algorithm

```
function CountSortOnDigits(int A[ ], int n, int digits[ ])
    create an array $C[\,]$ of length 10, and initialize all its cells to 0
    create an array $T[\,]$ of length $n$
    for (i = 0 to $n-1$) do increment $C[digits[i]]$;
    for ($i = 1$ to 9) do $C[i] = C[i-1] + C[i]$;
    for ($i = n-1$ to 0) do
        $T[C[digits[i]] - 1] = A[i]$;
        decrement $C[digits[i]]$;
    use a loop to copy $T$ into $A$;

function RadixSortNonNegative(int A[ ], int n)
    int M = maximum value in array $A$
    create an array $digits[\,]$ of length $n$
    $e = 1$;
    while ($M/e > 0$) do
        for ($i = 0$ to $n-1$) do
            $digits[i] = (A[i]/e)\%10$;
        CountSortOnDigits(A, n, digits);
        e = e*10;
```

[3] Note the importance of stability. If counting sort was not stable, then in the last round, we could have switched 47 and 124 leading to an incorrect order.

[4] If the numbers are written in base b, then the complexity is $\Theta\big(d*(n+b)\big)$.



Radix Sort round on the last digit:

A:	123	47	569	3459	124	53	1012
digits:	3	7	9	9	4	3	2

	0	1	2	3	4	5	6	7	8	9
C:	0	0	1	2	1	0	0	1	0	2

	0	1	2	3	4	5	6	7	8	9
C (in cumulative form):	0	0	1	3	4	4	4	5	5	7

T:	1012							C[2]−−: C[2] = 0
T:	1012		53					C[3]−−: C[3] = 2
T:	1012		53	124				C[4]−−: C[4] = 3
T:	1012		53	124			3459	C[9]−−: C[9] = 6
T:	1012		53	124		569	3459	C[9]−−: C[9] = 5
T:	1012		53	124	47	569	3459	C[7]−−: C[7] = 4
T:	1012	123	53	124	47	569	3459	C[3]−−: C[3] = 1

Figure 2: Illustration of Radix Sort algorithm.

3 Dealing with Negative Integers

We show two approaches to deal with negative integers. The first approach is arguably simpler to implement, but has a drawback. The second approach, which is slightly trickier, can be used to alleviate this problem. In either case, the complexity is the same as that of radix sort.

3.1 Approach 1

The idea is to simply shift the scale of numbers. Specifically:

- We first find the minimum number in the array.
- If the minimum is non-negative, then simply radix sort the array and you are done.
- Otherwise, let min be the minimum value.
- You subtract min from every number in the array, thereby increasing every number by $|min|$; note that this does not change the relative order of numbers.

  Also, this ensures that all numbers are now non-negative.
- Now radix sort the modified array.
- Finally, you add min to every number in the array, thereby decreasing every number by $|min|$ (and reverting to original values).

3.2 Approach 2

The problem with the Approach 1 is that if you increase each number by the magnitude of the minimum value, it can lead to integer overflow (and thus positive numbers may change to negative numbers), and you are no better off. The following approach gets over this hurdle (at the cost of extra space, which can be avoided with a slightly cleverer implementation.)

- We create two separate arrays – $NEG[\,]$ containing the negative numbers and $NONNEG[\,]$ containing the non-negative numbers.[5]
- For the $NONNEG$ array, just use radix sort.
- For the NEG array, first negate the numbers (so that they become positive) and then use radix sort on the resultant array. Now, negate the numbers of NEG array again (so that they again become negative). Right now NEG array is sorted but based on absolute values, which implies that it is reverse sorted based on actual (negative) values.
- Clearly, we have the original array split into two arrays – $NEG[\,]$ containing the negative numbers in reverse sorted order and $NONNEG[\,]$ containing the non-negative numbers in sorted array. Sort the original array by first copying NEG from the end and then copying $NONNEG$ from the beginning.

[5] Scan through the original array once to count the number of negative numbers, which gives you the count of non-negative numbers as well. Let the respective counts be x and y. Now, create $NEG[\,]$ and $NONNEG[\,]$ having lengths x and y respectively. Scan through the original array once again, and copy to $NEG[\,]$ or $NONNEG[\,]$, depending on whether the number is negative or non-negative.