

## CS217 - Data Structures & Algorithm Analysis (DSAA)

### Lecture #7

#### ► Sorting in Linear Time

Prof. Pietro S. Oliveto  
Department of Computer Science and Engineering  
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn  
<https://faculty.sustech.edu.cn/oliveto>

Reading: Chapter 8

#### ► Linear-Time Sorting

- The lower bound of  $\Omega(n \log n)$  is bad news for applications where comparisons are the only source of information.
- However, it suggests a way out: where possible, **use more information** than mere comparisons!
- Elements to be sorted are often **numbers or strings**, which reveal more information.

#### ► Aims of this lecture

- To show how to sort numbers in a **bounded range** in **linear time**.
- Two algorithms use operations other than comparisons so the  $\Omega(n \log n)$  runtime will not apply to them
- **CountingSort**
- **RadixSort**

#### ► CountingSort: Idea Require: boundness!

- Assume that the input elements are integers in  $\{0, \dots, k\}$ .
- For each element  $x$ , CountingSort **counts the number of elements less than  $x$** .
  - For instance, if 17 elements are smaller than  $x$ , then  $x$  belongs in output position 18.
- Caveat: **need to make sure that equal elements are put in different output positions.**
- CountingSort **uses an array  $C[0 \dots k]$  for counting and an array  $B[1 \dots n]$  for writing the output.**

## CountingSort

- Initialise counter array
- Count elements
- Running sum: #elements  $\leq i$
- Write elements to output

COUNTINGSORT( $A, B, k$ )

```

1: let  $C[0 \dots k]$  be a new array
2: for  $i = 0$  to  $k$  do
3:    $C[i] = 0$ 
4: for  $j = 1$  to  $A.length$  do
5:    $C[A[j]] = C[A[j]] + 1$ 
6: for  $i = 1$  to  $k$  do
7:    $C[i] = C[i] + C[i - 1]$ 
8: for  $j = A.length$  downto 1 do
9:    $B[C[A[j]]] = A[j]$ 
10:   $C[A[j]] = C[A[j]] - 1$ 

```

## CountingSort

COUNTINGSORT( $A, B, k$ )

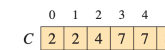
```

1: let  $C[0 \dots k]$  be a new array
2: for  $i = 0$  to  $k$  do
3:    $C[i] = 0$ 
4: for  $j = 1$  to  $A.length$  do
5:    $C[A[j]] = C[A[j]] + 1$ 
6: for  $i = 1$  to  $k$  do
7:    $C[i] = C[i] + C[i - 1]$ 
8: for  $j = A.length$  downto 1 do
9:    $B[C[A[j]]] = A[j]$ 
10:   $C[A[j]] = C[A[j]] - 1$ 

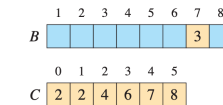
```



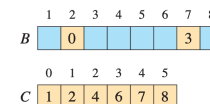
(a)



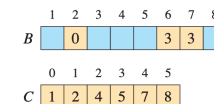
(b)



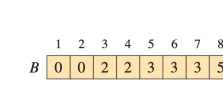
(c)



(d)



(e)



(f)

## CountingSort: Runtime

- Initialise counter array
- Count elements
- Running sum: #elements  $\leq i$
- Write elements to output

COUNTINGSORT( $A, B, k$ )

```

1: let  $C[0 \dots k]$  be a new array
2: for  $i = 0$  to  $k$  do
3:    $C[i] = 0$ 
4: for  $j = 1$  to  $A.length$  do
5:    $C[A[j]] = C[A[j]] + 1$ 
6: for  $i = 1$  to  $k$  do
7:    $C[i] = C[i] + C[i - 1]$ 
8: for  $j = A.length$  downto 1 do
9:    $B[C[A[j]]] = A[j]$ 
10:   $C[A[j]] = C[A[j]] - 1$ 

```

Time

$\Theta(k)$

$\Theta(n)$

$\Theta(k)$

$\Theta(n)$

- Runtime is  $\Theta(n + k)$ 
  - Depends on two input parameters instead of just the problem size  $n$ .
  - This is  $O(n)$  if  $k = O(n)$ .

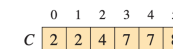
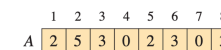
## CountingSort: Correctness

- Loop Invariant:

"At the start of each iteration  $j$  of the last for loop, the elements  $A[j + 1..n]$  are in the right position in  $B$  and the last element in  $A$  that has not yet been copied in  $B$ , with value  $A[j] = i$ , belongs to  $B[C[i]]$ ."

Initialisation:

At the start of the loop  $j = n$  and no elements have been copied. The array  $C$  provides for each element, the number of elements in  $A$  that are smaller or equal to it. So the last element of  $A$ ,  $A[n] = i$ , naturally goes in position  $B[C[i]]$ .



(b)

## ► CountingSort: Correctness

- Loop Invariant:

“At the start of each iteration  $j$  of the last for loop, the elements  $A[j + 1..n]$  are in the right position in  $B$  and the last element in  $A$  that has not yet been copied in  $B$ , with value  $A[j] = i$ , belongs to  $B[C[i]]$ .”

### Maintenance:

At iteration  $j$ , the loop invariant tells us that the element  $A[j] = i$  goes in  $B[C[i]]$  and we copy it in.

Since the next element equal to  $i$  in  $A$  that has not yet been copied in  $B$  should go in position  $B[C[i] - 1]$ , we decrement  $C[i]$  re-establishing the loop invariant (the array  $C$  is updated such that each element  $i$  of  $A$  still to be copied in is indexed correctly again)

---

COUNTINGSORT( $A, B, k$ )

```

1: let  $C[0..k]$  be a new array
2: for  $i = 0$  to  $k$  do
3:    $C[i] = 0$ 
4: for  $j = 1$  to  $A.length$  do
5:    $C[A[j]] = C[A[j]] + 1$ 
6: for  $i = 1$  to  $k$  do
7:    $C[i] = C[i] + C[i - 1]$ 
8: for  $j = A.length$  downto 1 do
9:    $B[C[A[j]]] = A[j]$ 
10:   $C[A[j]] = C[A[j]] - 1$ 

```

---

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B	0						3	

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

(d)

10

CSE217: Data Structures & Algorithm Analysis

## ► CountingSort: Correctness

- Loop Invariant:

“At the start of each iteration  $j$  of the last for loop, the elements  $A[j + 1..n]$  are in the right position in  $B$  and the last element in  $A$  that has not yet been copied in  $B$ , with value  $A[j] = i$ , belongs to  $B[C[i]]$ .”

### Termination:

When the loop terminates  $j = 0$ . The loop invariant tells us that all the elements of  $A[1..n]$  are in the right position in  $B$  thus there are no more elements to be copied.

---

COUNTINGSORT( $A, B, k$ )

```

1: let  $C[0..k]$  be a new array
2: for  $i = 0$  to  $k$  do
3:    $C[i] = 0$ 
4: for  $j = 1$  to  $A.length$  do
5:    $C[A[j]] = C[A[j]] + 1$ 
6: for  $i = 1$  to  $k$  do
7:    $C[i] = C[i] + C[i - 1]$ 
8: for  $j = A.length$  downto 1 do
9:    $B[C[A[j]]] = A[j]$ 
10:   $C[A[j]] = C[A[j]] - 1$ 

```

---

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

11

CSE217: Data Structures & Algorithm Analysis

## ► Stability in Sorting

- CountingSort is stable: numbers with the same value appear in the output in the same order as they do in the input array.
  - The order of equal elements is preserved.
  - This property is relevant when satellite data (e.g. Java objects) is attached to keys being sorted.
  - We may think of the original order being used to break ties between elements with equal keys.
  - Works well for sorting emails according to (1) read/unread and (2) date.

- How do we prove stability of CountingSort?
- Can I be faster if I don't care about stability?

## ► Counting Sort: advantages & disadvantages

- Sorts in linear time  $n$  integers in the range  $\{0..k\}$  if  $k=O(n)$
- Is stable (preserves original ordering for breaking ties)
- Does not sort in place
- What if  $k = \omega(n)$  (or  $k \gg n$ )? (eg., I have to sort  $n=100$  numbers between 0 and 1 billion)
- Is there a way of limiting the size of  $k$ ?

## ► Radix Sort

- How many different integers can appear in a digit in a number of  $x$  digits?
- How many different letters can appear in a word written using a latin (eg., English) alphabet?
- Can we **sort digit by digit (or letter by letter)**?
- Stability helps for sorting numbers digit by digit (or English words letter by letter).

## ► Radix Sort

- Assume that each array element has  $d$  digits (from lowest significance to highest significance)

```
RADIXSORT( $A, d$ )
1: for  $i = 1$  to  $d$  do
2:   use a stable sort to sort array  $A$  on digit  $i$ 
```

这边我们只能从右边的digit向左边的 digit 依次做 stable sort，否则会出问题！  
 从左向右：个位排序完成后，十位排序时，遇到相同的十位，按照先前的个位顺序排序，正确！  
 从右向左：十位排序完后，个位排序时，遇到个位相同的数，按照十位顺序排序。那这样有可能排出来这样的情况：94 < 45 < 55 < 65 < 92 < 82 < 72 < 42。

## ► Radix Sort: Example

```
RADIXSORT( $A, d$ )
1: for  $i = 1$  to  $d$  do
2:   use a stable sort to sort array  $A$  on digit  $i$ 
```

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

## ► Radix Sort: Correctness

```
RADIXSORT( $A, d$ )
1: for  $i = 1$  to  $d$  do
2:   use a stable sort to sort array  $A$  on digit  $i$ 
```

**Correctness follows from stability and induction on columns.**

**Loop Invariant:** "At each iteration of the **for** loop, the array is sorted on the last  $i-1$  digits"

**Initialisation:** The array is trivially sorted on the empty set of digits for  $i=0$

329
457
657
839
436
720
355

## ► Radix Sort: Correctness

---

RADIXSORT( $A, d$ )

---

```

1: for  $i = 1$  to  $d$  do
2:   use a stable sort to sort array  $A$  on digit  $i$ 

```

---

Correctness follows from stability and induction on columns.

**Loop Invariant:** “At each iteration of the **for** loop, the array is sorted on the last  $i-1$  digits”

**Maintenance:** The invariant tells us that the array is sorted on the last  $i-1$  digits. Now we sort the  $i$ -th digit re-establishing the loop invariant, since our stable sort ensures that elements with same  $i$ -th digit remain in the same order as before sorting.

720  
329  
436  
839  
355  
457  
657

## ► Radix Sort: Correctness

---

RADIXSORT( $A, d$ )

---

```

1: for  $i = 1$  to  $d$  do
2:   use a stable sort to sort array  $A$  on digit  $i$ 

```

---

Correctness follows from stability and induction on columns.

**Loop Invariant:** “At each iteration of the **for** loop, the array is sorted on the last  $i-1$  digits”

**Termination:** The loop terminates when  $i=d+1$ . Then the loop invariant states that the array is completely sorted.

329  
355  
436  
457  
657  
720  
839

## ► Radix Sort: Runtime

---

RADIXSORT( $A, d$ )

---

```

1: for  $i = 1$  to  $d$  do
2:   use a stable sort to sort array  $A$  on digit  $i$ 

```

---

- Given  $n$   $d$ -digit numbers in which each digit can take up to  $k$  possible values, RadixSort using CountingSort sorts these numbers in time  $\Theta(d(n + k))$ 
  - This is just the runtime of running CountingSort  $d$  times.
- Advantage to CountingSort?
- The support array has only size  $[0..9]$  for numbers,  $[A..Z]$  for words with latin letters ( $k$  is not too large)

## ► Radix Sort: Application

Task: Sort  $n$  integers in the range 0 to  $n^3 - 1$

- Runtime of a ComparisonSort algorithm?
- Runtime of CountingSort?
- Runtime of RadixSort?

A number  $n^3-1$  requires  $\lceil \log_{10} n^3 \rceil = \lceil 3 \log_{10} n \rceil = O(\log n)$  digits

Eg.  $n = 10$ , then  $n^3 - 1 = 999$ , and  $\lceil 3 \log_{10} n \rceil = 3$

Eg.  $n = 20$ , then  $n^3 - 1 = 7999$ , and  $\lceil 3 \log_{10} n \rceil = 4$

So RadixSort has runtime  $\Theta(d(n + k)) = \Theta(\log n(n + 10)) = \Theta(n \log n)$

Turn the numbers to base  $n$  (eg.,  $n=20$  range:  $[0..JJJ]$ )

$\Rightarrow \lceil 3 \log_n n \rceil = O(1)$ , and  $T(n) = O(n + k) = O(n + k) = O(n)$

**Caveat?** I have to make sure I can convert bases and back in time  $O(1)$

## ► Summary

- CountingSort sorts numbers in a bounded range  $\{0, \dots, k\}$  in time  $\Theta(n + k)$ .
- RadixSort uses a **stable sorting algorithm** to sort digit by digit.
  - **Stability** preserves the order of equal elements.
  - The time for sorting  $d$ -digit numbers is  $\Theta(d(n + k))$ .
  - This is  $\Theta(n)$  when  $d = O(1)$  and  $k = O(n)$ .