Counting Sort

Ju-Won Seo
2019.07.31

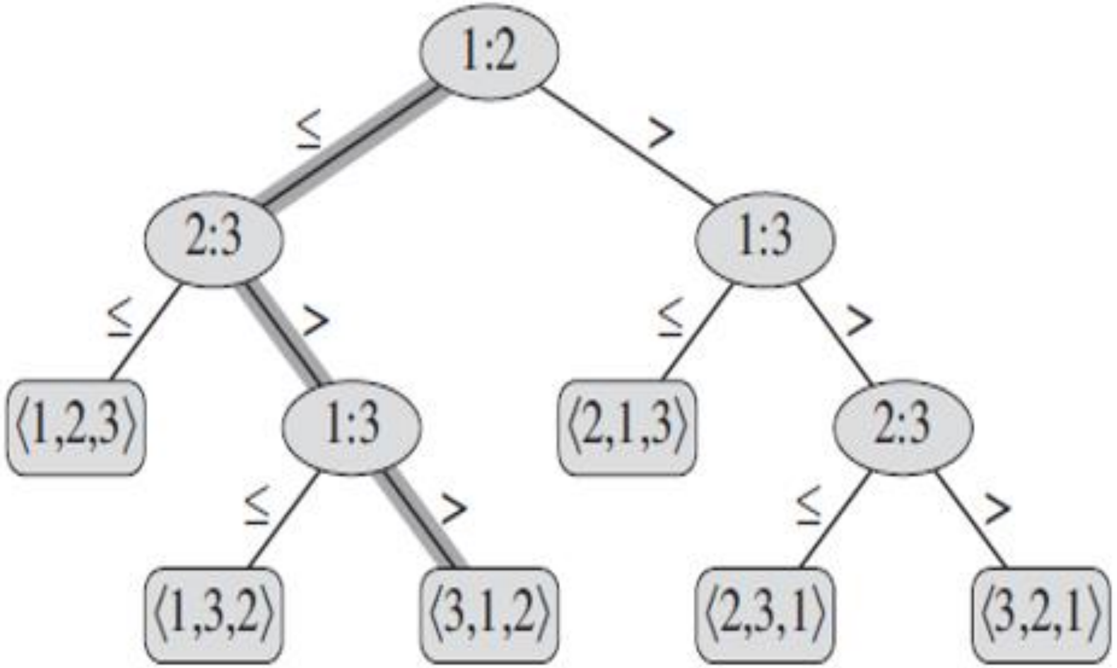
Table of Contents

1. Introduction

Different types of sorting

Algorithm	Average Case	Worst Case
Bubble Sort	O($\boldsymbol{n}^2$)	O($\boldsymbol{n}^2$)
Merge Sort	$\boldsymbol{O(n\log n)}$	$\boldsymbol{O(n\log n)}$
Quick Sort	$\boldsymbol{\theta(n\log n)}$	O($\boldsymbol{n}^2$)

Decision Tree

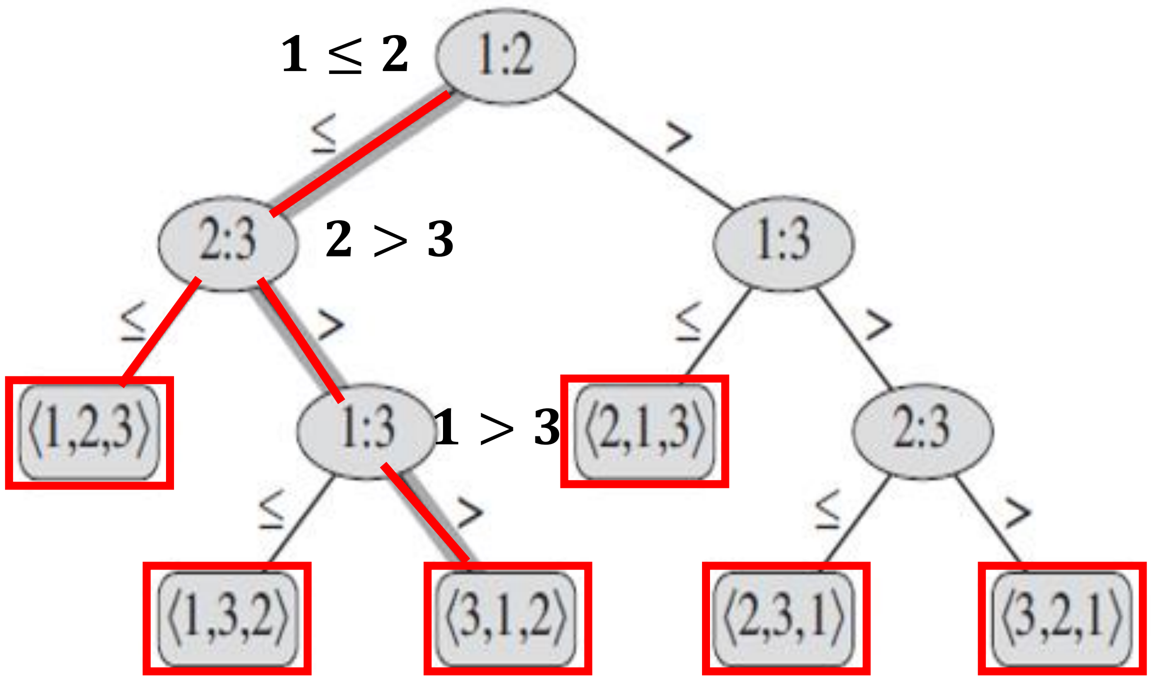


Assumption

1. **All the input elements are distinct.**
2. **All comparisons have the form $a_i \leq a_j$**

about an input sequence <a_1, a_2, a_3, …. a_n>

Lower bound for the worst Case



Lemma

1. Each of $n!$ Permutations on n elements must appear as one of the leaf nodes.

2. Decision tree has this trait that $l \leq 2^h$ $(l = \text{leaf node},\ h = \textit{height of binary tree})$

3. The worst-case number of comparisons equals the height of its decision tree.

Lower bound for the worst Case

Lemma

1. Each of $n!$ Permutations on n elements must appear as one of the leaf nodes.

2. Binary tree has this trait that $l \leq 2^h$
   ($l = \text{leaf node},\ h = height\ of\ binary\ tree$)

3. The worst-case number of comparisons equals the height of its decision tree.

4. $\log n! = \Omega(nlogn)$ by Stirling approximation

$n! \leq l \leq 2^h$ (using lemma 1 and 2)

$n! \leq 2^h$

$\log_2 n! \leq h$

$h \geq \log n!$
$= \Omega(nlogn)$ (using lemma 3 and 4)

∴ Comparison sort algorithm requires $\Omega(nlogn)$ comparisons in the worst case.

2. Counting Sort

- Counting Sort

- **A method of sort by counting how many elements each appears**

- **No comparison between elements.**

- **Assumption**

1. Each of the n input elements is an integer in the range 0 to k (k is positive integer)

2. We know k integer

Algorithm

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Step

1. Initialize Array C[0..k]

2. Counting the number of each element from input array A and store them in array C

3. For each element x in array C, calculate the number of elements less than or equal to x

4. Use Step 3, place element x directly into its position in the output array B.

Algorithm - we need

Array A -> Input array[1...n]

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

Array B -> Sorted array[1...n]

	1	2	3	4	5	6	7	8
B								

Array C -> Counting array[0...k]

	0	1	2	3	4	5
C						

K = 5 (range is 0 to 5)

Algorithm - step 1

```
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
```

Initialize Array C

Input array

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

Counting array

	0	1	2	3	4	5
C	**0**	**0**	**0**	**0**	**0**	**0**

Algorithm - step 2

for $j = 1$ **to** $A.length$
    $C[A[j]] = C[A[j]] + 1$

Counting the number of each element from input array A

Input array

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

Counting array

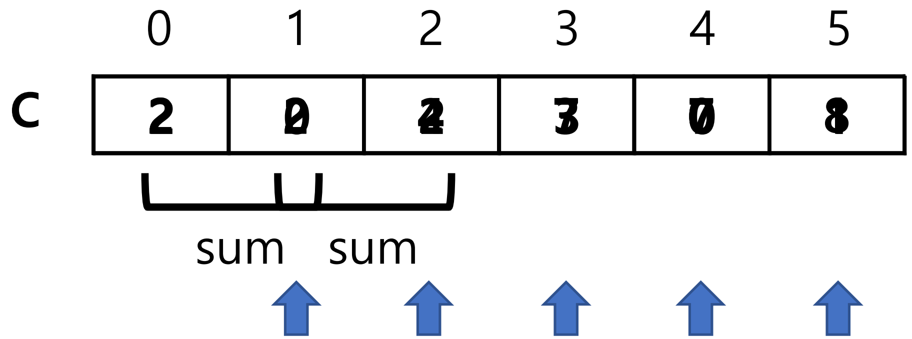
	0	1	2	3	4	5
C	**2**	**0**	**2**	**3**	**0**	**1**

Algorithm - step 3

```
for i = 1 to k
    C[i] = C[i] + C[i - 1]
```

Calculate the number of elements less than or equal to C[i]

Counting array



Algorithm - step 4

```
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

Place element A[j] directly into its position in array B

Input array

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

Counting array

	0	1	2	3	4	5
C	**0**	**2**	**2**	**4**	**7**	**7**

Sorted array

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

Analysis

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
        C[i] = 0
```

$\rightarrow \theta(k)$

```
for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
```

$\rightarrow \theta(n)$

```
for i = 1 to k
        C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
```

$\rightarrow \theta(k)$

```
for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

$\rightarrow \theta(n)$

$$\therefore \theta(n+k)$$

Property of counting sort

1. Trough step 3, when we sorted, We can see where each element should be located.

2. If $k \leq n$, we can use only $\theta(n)$ time for sorting.

3. If $k \gg n$, it can consume memory too much. (ex. 1, 10000000, 3, 2)

4. Counting sort is stable sort algorithm.

What is stable?

Numbers with the same value appear in the output array in the same order as they do in the input array.

	1	2	3	4	5	6	7	8
A	**22a**	**55a**	**33a**	**00a**	**22b**	**33b**	**00b**	**33c**

	1	2	3	4	5	6	7	8
B	**0.a**	**0.b**	**2.a**	**2.b**	**3.a**	**3.b**	**3.c**	**5.a**

Stability of counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
        C[i] = 0
for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
        C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

	1	2	3	4	5
A	**1**	**1**	**0**	**2**	**1**

Stability of counting sort

	1	2	3	4	5
A	1.a	1.b	0.a	2.a	1.c

	0	1	2
C	0 1	2 4	4 5

	1	2	3	4	5
B	0.a	1.a	1.b	1.c	2.a

Stability of counting sort

1. By using this property that stability, we can make radix sort.

3. Radix Sort

- Radix Sort

- **No comparison between elements.**
- **Use a stable sort like counting sort.**
- **Assumption**

1. Each element in the n-element array A has d digits(d is the highest-order digit).
2. All elements are integers not negative and decimal.

Algorithm

The highest digit

Counting Sort

RADIX-SORT(A, d)

1 **for** $i = 1$ **to** d

2 use a $\boxed{\text{stable sort}}$ to sort array A on digit i

Algorithm

```
void radixSort() {

    // For get the highest digit
    for (int i = 1; i <= n; i++) {
        mx = max(mx, input[i]);
    }

    for (int i = 1; mx / i > 0; i = i * 10) {
        countingSort(i);
    }
}
```

```
void countingSort(int digit) {

    for (int i = 1; i <= n; i++) {
        c[(input[i] / digit) % 10]++;
    }

    for (int i = 1; i < 10; i++) {
        c[i] = c[i] + c[i - 1];
    }

    for (int i = n; i >= 1; i--) {
        tmp[c[(input[i] / digit) % 10]] = input[i];
        c[(input[i] / digit) % 10]--;
    }

    for (int i = 1; i <= n; i++) {
        input[i] = tmp[i];
    }
}
```

Algorithm

Input array[1..n]

Input

1	2	3	4	5	6	7
329	**457**	**657**	**839**	**436**	**720**	**355**

Temporary array[1..n]

Tmp

1	2	3	4	5	6	7
0	**0**	**0**	**0**	**0**	**0**	**0**

Counting array[10]

C

0	1	2	3	4	5	6	7	8	9
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Algorithm

```
void radixSort() {

    // For get the highest digit
    for (int i = 1; i <= n; i++) {
        mx = max(mx, input[i]);
    }

    for (int i = 1; mx / i > 0; i = i * 10) {
        countingSort(i);
    }
}
```

	1	2	3	4	5	6	7
Input	**329**	**457**	**657**	**839**	**436**	**720**	**355**

MAX : 839

The highest digit : 3

Algorithm

now digit is 1

```
void countingSort(int digit) {

    for (int i = 1; i <= n; i++) {
        c[(input[i] / digit) % 10]++;
    }

    for (int i = 1; i < 10; i++) {
        c[i] = c[i] + c[i - 1];
    }

    for (int i = n; i >= 1; i--) {
        tmp[c[(input[i] / digit) % 10]] = input[i];
        c[(input[i] / digit) % 10]--;
    }

    for (int i = 1; i <= n; i++) {
        input[i] = tmp[i];
    }
}
```

	1	2	3	4	5	6	7
Input	**329**	**457**	**657**	**839**	**436**	**720**	**355**

	0	1	2	3	4	5	6	7	8	9
C	**0**	**0**	**0**	**0**	**0**	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Algorithm

now digit is 1

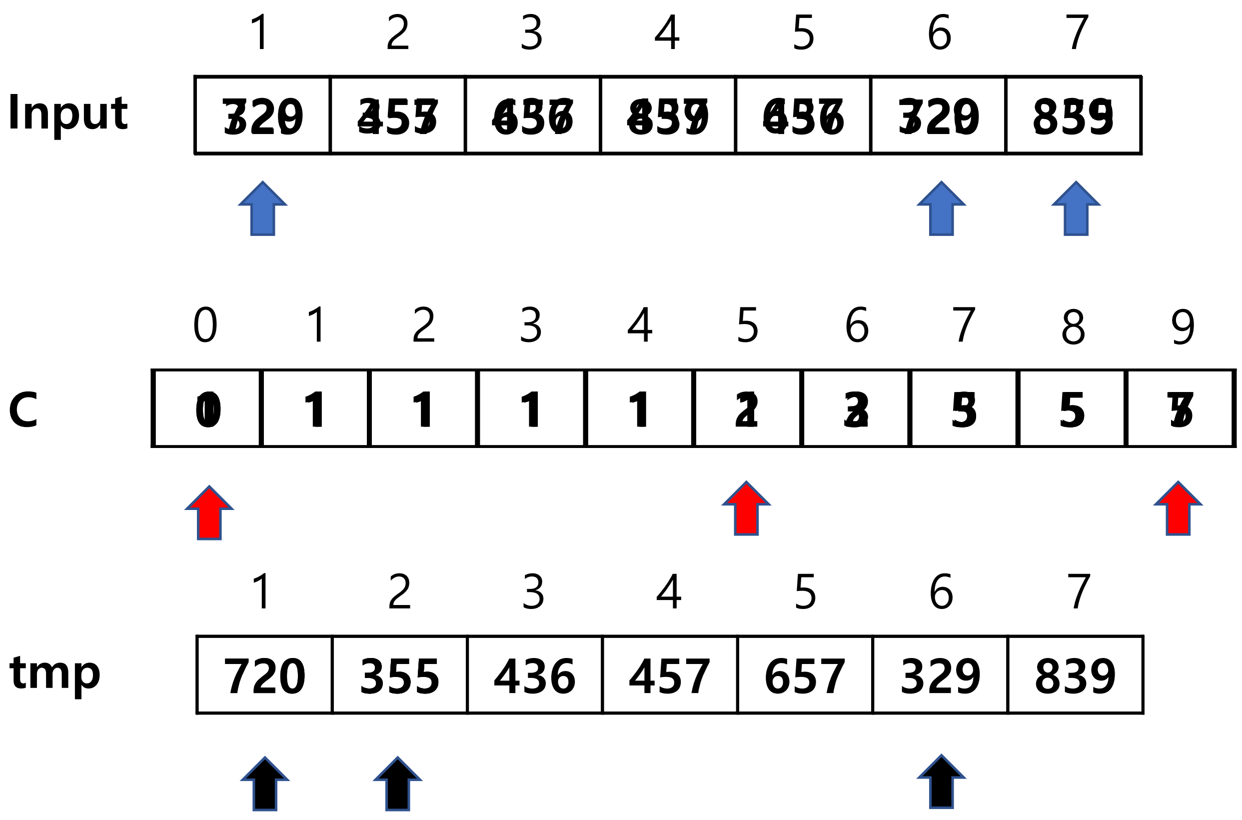
```
void countingSort(int digit) {

    for (int i = 1; i <= n; i++) {
        c[(input[i] / digit) % 10]++;
    }

    for (int i = 1; i < 10; i++) {
        c[i] = c[i] + c[i - 1];
    }

    for (int i = n; i >= 1; i--) {
        tmp[c[(input[i] / digit) % 10]] = input[i];
        c[(input[i] / digit) % 10]--;
    }

    for (int i = 1; i <= n; i++) {
        input[i] = tmp[i];
    }
}
```



Algorithm

	1	2	3	4	5	6	7
Input	**329**	**457**	**657**	**839**	**436**	**720**	**355**

↓

	1	2	3	4	5	6	7
Sorted digit 1	**720**	**355**	**436**	**457**	**657**	**329**	**839**

↓

	1	2	3	4	5	6	7
Sorted digit 2	**720**	**329**	**436**	**839**	**355**	**457**	**657**

↓

	1	2	3	4	5	6	7
Sorted digit 3	**329**	**355**	**436**	**457**	**657**	**720**	**839**

Analysis

```
void radixSort() {

    // For get the highest digit
    for (int i = 1; i <= n; i++) {
        mx = max(mx, input[i]);
    }

    for (int i = 1; mx / i > 0; i = i * 10) {
        countingSort(i);
    }
}
```

$\rightarrow \theta(n)$

$\rightarrow \theta(d(n+k))$

if d is constant and k = $O(n)$

$\therefore O(n)$

4. Conclusion

Summary

1. Comparison sort algorithm requires $\Omega(nlogn)$ time.

2. Counting sort is not comparison sort and requires $\theta(n + k)$ time.

3. If k(number of range)is bigger than n, be careful about overspending memory

4. Radix sort requires $O(n)$ time.

5. Radix sort uses counting sort's property such as stability.

Q & A

Thank you!