COUNTING SORT

* Assume each input element is an integer.
* Range of input number 0 to k
* When $k = O(n)$ then Time complexity = $\Theta(n)$
* Main Idea
  - For each input element X, the number of elements less than X are determined.
  - It uses this information to place element X directly into its position.
  - Eg If 20 elements are less than X, then X belongs in o/p array at 21 position

Modify the scheme Handle Duplicate elements

A

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

→

2	0	2	3	0	1
0	1	2	3	4	5

1) Let C[0, ... k] be a new Array
2) For i=0 to k
3) C[i] ← 0

0	0	0	0	0	0
0	1	2	3	4	5

4) For j ← 1 to A.length
5) C[A[j]] ← C[A[j]] + 1

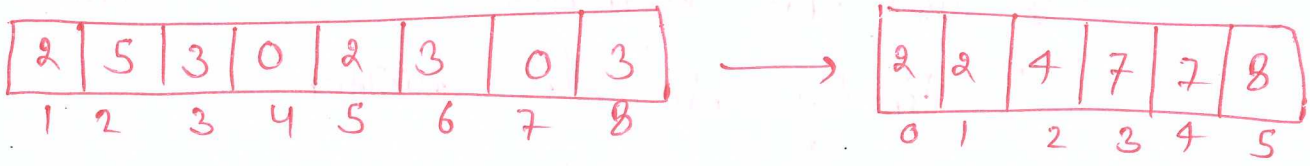
2	0	2	3	0	1
0	1	2	3	4	5

C[j] Now contains the number of elements equal to j.

7) For i = 1 to K
8) C[i] ← C[i] + C[i-1]

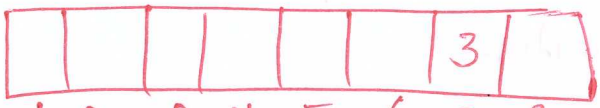
* C[i] now contains the number of elements less than or Equal to i.

10) For j ← A.length to 1
11) B[C[A[j]]] ← A[j]
12) C[A[j]] ← C[A[j]] - 1

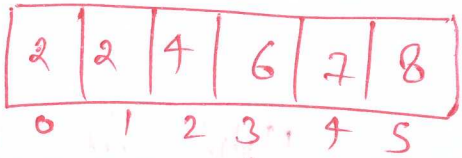


j = 8 A[j] = 3 C[A[j]] = 7 B[7] ← 3

j = 7

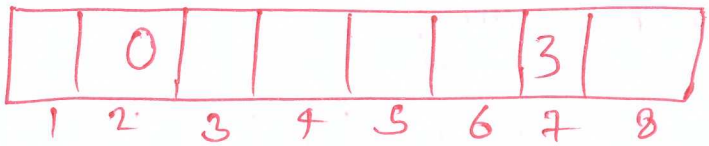


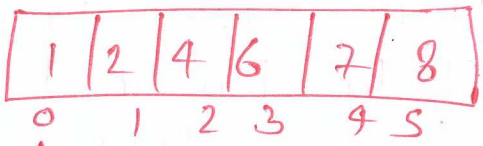
B



C[3] ← C[3] - 1

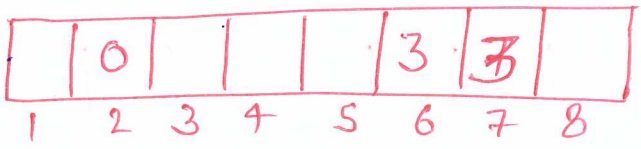
A[j] = 0 C[A[j]] = 2 B[2] ← 0

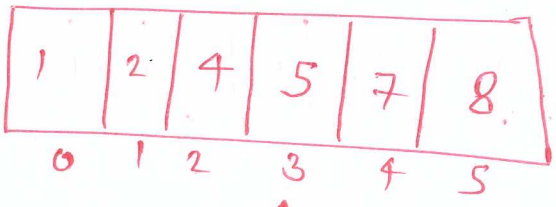




C[0] ← C[0] - 1

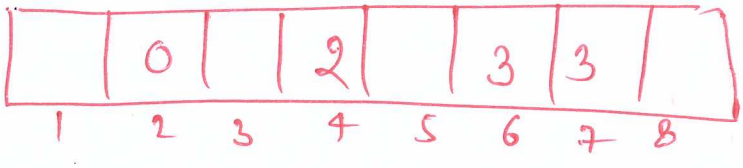
j=6 A[j] = 3 C[A[j]] = 6 B[6] ← 3

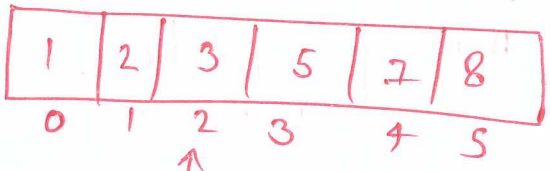




C[3] ← C[3] - 1

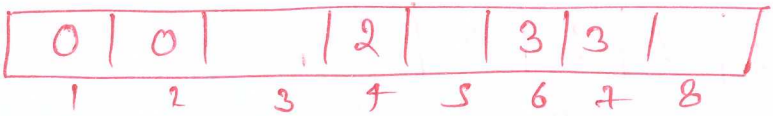
j=5 A[j] = 2 C[A[j]] = 4 B[4] ← 2

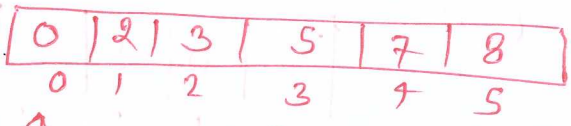




C[2] ← C[2] - 1

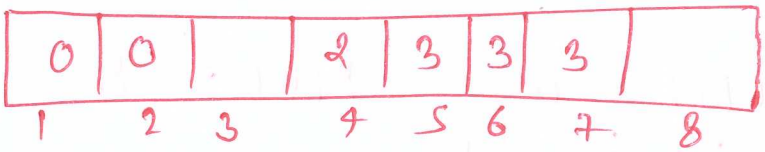
j=4 A[j] = 0 C[A[j]] = 1 B[1] ← 0

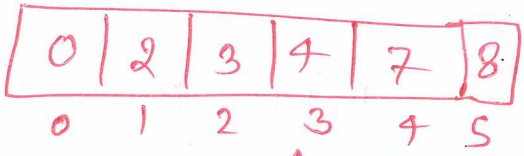




C[0] ← C[0] - 1

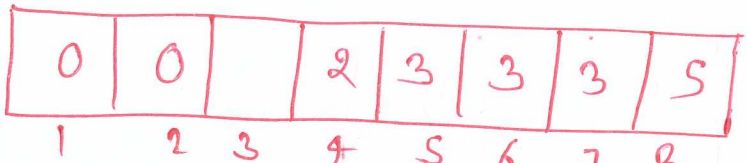
j=3 A[j] = 3 C[A[j]] = 5 B[5] ← 3

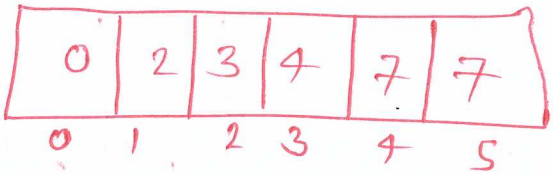




C[3] ← C[3] - 1

j=2 A[j] = 5 C[A[j]] = 8 B[8] ← 5





C[5] ← C[5] - 1

j=1

A[j] = 2 C[A[j]] = 3 B[3] ← 2

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

0	2	2	4	7	7
0	1	2	3	4	5

C[2] ← C[2] − 1

Let A =

2	4	6	3	5	8
1	2	3	4	5	6

C =

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

C =

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

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

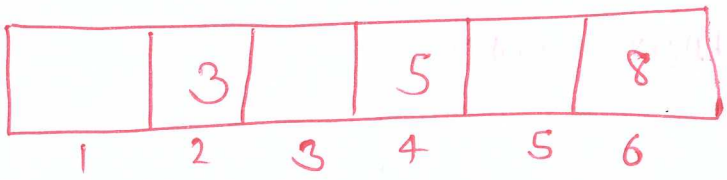
j=6 A[j] = 8 C[A[j]] = 6 B[6] ← 8

					8
1	2	3	4	5	6

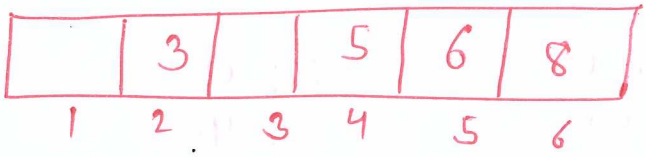
j=5 A[5] = 5 C[A[j]] = ~~4~~ 4 B[4] ← 5

			5		8
1	2	3	4	5	6

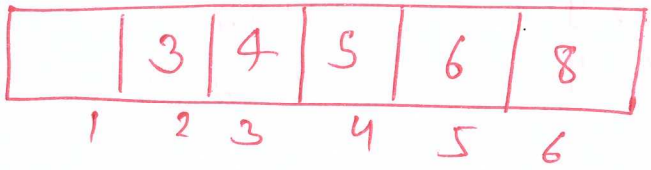
j=4 A[j] = 3 C[A[j]] = 2 B[2] ← 3



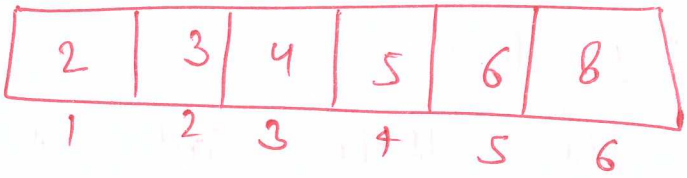
j=3 A[j] = 6 C[A[j]] = 5 B[5] ← 6



j=2 A[j] = 4 C[A[j]] = 3 B[3] ← 4



j=1 A[j] = 2 C[A[j]] = 1 B[1] ← 2



* Counting Sort is STABLE

* Used for RADIX SORT

RADIX SORT

3 2 9		7 2 0		7 2 0		3 2 9
4 5 7		3 5 5		3 2 9		3 5 5
6 5 7		4 3 6		4 3 6		4 3 6
8 3 9		4 5 7		8 3 9		4 5 7
4 3 6	→	6 5 7	→	3 5 5	→	6 5 7
7 2 0		3 2 9		4 5 7		7 2 0
3 5 5		8 3 9		6 5 7		8 3 9

```
RADIX_SORT (A, d)
    For i = 1 to d
        Use STABLE Sort on A using digit i
```

Time Complexity = $\Theta(d(n+k))$

Each digit can take upto k possible values.

$= \Theta(d(n+b))$

where b is the base as ##no. will be in range 0 to b-1

0 to 9 in case of decimal

Integer with base b
Max. value = K
No of digit (d) = $\lfloor \log_b K \rfloor + 1$

Let K = 9999999999

$d = \log_{10} 9999999999 \approx \log_{10} 10^{10} = 10$

Time Complexity

$O(n+b)$ Using counting sort

In case of decimal max. no. can be 9

Total Time = $O((n+b)d)$

$= O((n+b)\log_b K)$

digit in b/w 0 to n-1

Time complexity will be

Minimum when $b = \Theta(n)$ then $K = n^c$

$= O(n \log_n K)$

~~If $K = n^c$ then Time Comp = $\Theta(n)$~~

n = 394

$d_3 d_2 d_1$

First digit $d_1 = n\%10 = 4$ (394/10)

$d_2 = (n/10)\%10 = 39\%10 = 9$

$d_3 = ((n/10)/10)\%10 = 3$

$(n/b^{d-1})\%10$

In General

$(n/b^{d-1})\%b$

$d_1 = n \% 10$

$d_2 = (n/10) \% 10$

$d_3 = (n/100) \% 10$

$\downarrow$

$d_i = (n / b^{i-1}) \% 10$

$n = d_n d_q \ldots d_i \ldots d_3 d_2 d_1$

Parvees

> algorithm stable. How much additional time and space does your scheme entail?

Insertion sort is stable. When inserting $A[j]$ into the sorted sequence $A[1..j-1]$, we do it the following way: compare $A[j]$ to $A[i]$, starting with $i = j - 1$ and going down to $i = 1$. Continue at long as $A[j] < A[i]$.

Merge sort as defined is stable, because when two elements compared are equal, the tie is broken by taking the element from array L which keeps them in the original order.

Heapsort and quicksort are not stable.

One scheme that makes a sorting algorithm stable is to store the index of each element (the element's place in the original ordering) with the element. When comparing two elements, compare them by their values and break ties by their indices.

Additional space requirements: For n elements, their indices are $1 \ldots n$. Each can be written in $\lg n$ bits, so together they take $O(n \lg n)$ additional space.

Additional time requirements: The worst case is when all elements are equal. The asymptotic time does not change because we add a constant amount of work to each comparison.

8.3-3

> Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

Basis: If $d = 1$, there's only one digit, so sorting on that digit sorts the array.

Inductive step: Assuming that radix sort works for $d - 1$ digits, we'll show that it works for d digits.

Radix sort sorts separately on each digit, starting from digit 1. Thus, radix sort of d digits, which sorts on digits $1, \ldots, d$ is equivalent to radix sort of the low-order $d - 1$ digits followed by a sort on digit d. By our induction hypothesis, the sort of

the low-order $d-1$ digits works, so just before the sort on digit d, the elements are in order according to their low-order $d-1$ digits.

The sort on digit d will order the elements by their dth digit. Consider two elements, a and b, with dth digits a_d and b_d respectively.

- If $a_d < b_d$, the sort will put a before b, which is correct, since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will put a after b, which is correct, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order they were in, because it is stable. But that order is already correct, since the correct order of a and b is determined by the low-order $d-1$ digits when their dth digits are equal, and the elements are already sorted by their low-order $d-1$ digits.

If the intermediate sort were not stable, it might rearrange elements whose dth digits were equal—elements that *were* in the right order after the sort on their lower-order digits.

8.3-4

> Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

Treat the numbers as 3-digit numbers in radix n. Each digit ranges from 0 to $n-1$. Sort these 3-digit numbers with radix sort.

There are 3 calls to counting sort, each taking $\Theta(n+n) = \Theta(n)$ time, so that the total time is $\Theta(n)$.

8.3-5 ⋆

> In the first card-sorting algorithm in this section, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards would an operator need to keep track of in the worst case?

- Since a pass consists of one iteration of the loop on line 1–2, only d passes are needed.