Non comparison based sorting

Comparison Based sorting algorithm

- Selection Sort, Bubble Sort, Insertion Sort: $O(n^2)$
- Heap Sort, Merge sort: O(nlogn)
- Quick sort: : O(nlogn) average case

What is common to all these algorithms?

- Make **comparisons** between input elements

  $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$

why non- comparison based algorithm?

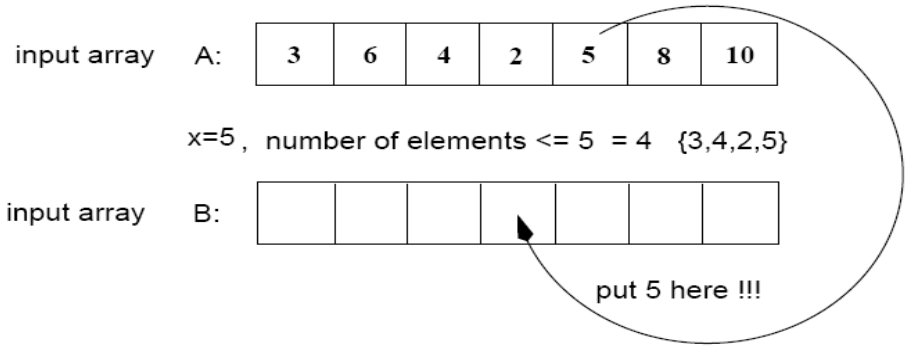
The performance of comparison based algorithm you will realize that bubble, selection and insertion sort take O(n) time to sort n items . While heap sort, quick sort and merge sort take around O(nlog n) and it can be proved that any comparison based sorting algorithm will take at least O(nlogn) operations to sort n elements hence we need a non comparison based algorithm which allows sort elements in linear time.

Linear sorting algorithms **(Non comparison based sorting)**

- Counting Sort
- Radix Sort
- Bucket sort

Counting Sort: In this sorting , no comparisons between input elements occur anywhere in this sorting .Counting sort is stable.

- Assumptions:
  - n integers which are in the range [0 … r]
  - r is in the order of n, that is, r=O(n)
- Idea:
  - For each element x, find the number of elements x
  - Place x into its correct position in the output array



Step 1: Find the number of times A[i] appears in A

Array A

3	6	4	1	3	4	1	4

Allocate C [1--- r]with 0

1	2	3	4	5	6
0	0	0	0	0	0

For 1 <= I <= n do ++C[A[i]]

i=1, A[i]=3 C[A[i]] =C[3] =1

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

i=2, A[i]=6 C[A[i]] =C[6] =1

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

i=3, A[i]=4 C[A[i]] =C[4] =1

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

i=4, A[i]=1 C[A[i]] =C[1] =1

1	2	3	4	5	6
1	0	1	1	0	1

i=5, A[i]=3 C[A[i]] =C[3] =2

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

i=6, A[i]=4 C[A[i]] =C[4] =2

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

i=7, A[i]=1 C[A[i]] =C[1] =2

1	2	3	4	5	6
2	0	2	2	0	1

i=8, A[i]=4 C[A[i]] =C[4] =3

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

C[i] contain number of time i appears in Array A

Step 2: Find the number of elements <= A[i]

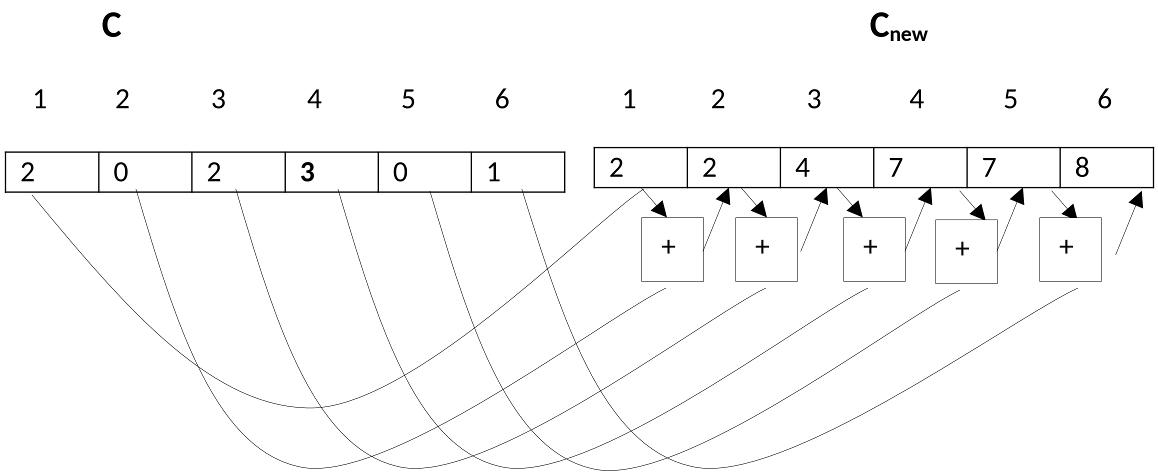
C array contain frequency of each element for range 1 to r

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

C_{new} contains number of elements less then equal to A[i]

$C_{new}[0] = C[0]$

$C_{new}[i] = C_{new}[i-1]+C[i]$



Algorithm:

- Start from the last element of A
- Place A[i] at its correct place in the output array
- Decrease C_{new}[A[i]] by one

Array A

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

C_{new}

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

Start with **last index** of array which **8** with element 4 place at its correct place in the output array B which is 7th palce with help of C_{new} array and decrease C_{new}[A[i]] by one

Array B

						4	
1	2	3	4	5	6	7	8

C_{new}

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

Index: 7

Array B

	1					4	
1	2	3	4	5	6	7	8

C_{new}

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

Index: 6

Array B

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

C_{new}

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

Index: 5

Array B

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

C_{new}

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

Index: 4

Array B

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

C_{new}

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

Index: 3

Array B

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

C_{new}

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

Index: 2

Array B

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

C_{new}

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

Index: 1

Array B

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

C_{new}

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

Algorithm

```
COUNTING-SORT(A, B, n, k)

for i ← 0 to r
    do C[ i ] ← 0        --------------------O(r)
for j ← 1 to n
    do C[A[ j ]] ← C[A[ j ]] + 1  ----------- O(n)
C[i] contains the number of elements equal to i
for i ← 1 to r
    do C[ i ] ← C[ i ] + C[i -1]  -----------------O(r)
 C[i] contains the number of elements ≤ i
for j ← n down to 1
    do B[C[A[ j ]]] ← A[ j ]          ------- O(n)
       C[A[ j ]] ← C[A[ j ]] - 1
```

Analysis of algorithm:

- Overall time: O(n + r)
- In practice we use Counting sort when r = O(n)

⇒ running time is O(n)