Counting and Radix Sort

Sorting So Far

- **Insertion sort**
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average

Sorting So Far

- **Merge sort**
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case

Sorting So Far

- **Quick sort**
  - Divide-and-conquer
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed
  - O(n lg n) average case
  - $O(n^2)$ worst case
  - Fast in practice

Counting sort

- Counting sort assumes that each of the n input elements is an integer in the range 0 to k.
- **n is the number of elements** and **k is the highest value element.**
- Consider the input set : 4, 1, 3, 4, 3 ➔n=5 and k=4
- Counting sort determines for each input element x, **the number of elements less than x**. And it uses this information to place element **x directly into its position in the output array.**
- For example if there exits 20 elements less that x then x is placed into the 21st position into the output array.
- The algorithm uses three array:

  Input Array: A[1..*n*] store input data where $A[j] \in \{1, 2, 3, \ldots, k\}$

  Output Array: B[1..*n*] finally store the sorted data

  Temporary Array: C[1..*k*]

Counting Sort

1. Counting-Sort(A, B, k)
2. Let C[0.....k] be a new array
3. for i=0 to k
4. C[i]= 0;
5. for j=1 to A.length (or n)
6. C[A[j]] = C[A[j]] + 1;
7. for i=1 to k
8. C[i] = C[i] + C[i-1];
9. for j=n or A.length down to 1
10. B[C[A[j]]] = A[j];
11. C[A[j]] = C[A[j]] - 1;

Counting Sort

1. Counting-Sort(A, B, k)
2. Let C[0.....k] be a new array
3. for i=0 to k **[Loop 1]**
4. C[i]= 0;
5. for j=1 to A.length (or n) **[Loop 2]**
6. C[A[j]] = C[A[j]] + 1;
7. for i=1 to k **[Loop 3]**
8. C[i] = C[i] + C[i-1];
9. for j=n or A.length down to 1 **[Loop 4]**
10. B[C[A[j]]] = A[j];
11. C[A[j]] = C[A[j]] - 1;

Counting-sort example

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

	0	1	2	3	4	5
C:						

	1	2	3	4	5	6	7	8
B:								

Executing Loop 1

3. for i=0 to k
4. C[i]= 0;

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

	0	1	2	3	4	5
C:	0	0	0	0	0	0

	1	2	3	4	5	6	7	8
B:								

Executing Loop 2

5. for j=1 to A.length or n
6. C[A[j]] = C[A[j]] + 1;

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

	0	1	2	3	4	5
C:	0	0	1	0	0	0

	1	2	3	4	5	6	7	8
B:								

Executing Loop 2

5. for j=1 to A.length or n
6. C[A[j]] = C[A[j]] + 1;

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

	0	1	2	3	4	5
C:	0	0	1	0	0	1

	1	2	3	4	5	6	7	8
B:								

Executing Loop 2

5. for j=1 to A.length or n
6. C[A[j]] = C[A[j]] + 1;

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

	0	1	2	3	4	5
C:	0	0	1	1	0	1

	1	2	3	4	5	6	7	8
B:								

Executing Loop 2

5. for j=1 to A.length or n
6. C[A[j]] = C[A[j]] + 1;

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

	0	1	2	3	4	5
C:	1	0	1	1	0	1

	1	2	3	4	5	6	7	8
B:								

Executing Loop 2

5. for j=1 to A.length or n
6. C[A[j]] = C[A[j]] + 1;

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

	0	1	2	3	4	5
C:	1	0	2	1	0	1

	1	2	3	4	5	6	7	8
B:								

Executing Loop 2

5. for j=1 to A.length or n
6. C[A[j]] = C[A[j]] + 1;

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

	0	1	2	3	4	5
C:	1	0	2	2	0	1

	1	2	3	4	5	6	7	8
B:								

Executing Loop 2

5. for j=1 to A.length or n
6. C[A[j]] = C[A[j]] + 1;

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

	0	1	2	3	4	5
C:	2	0	2	2	0	1

	1	2	3	4	5	6	7	8
B:								

Executing Loop 2

5. for j=1 to A.length or n
6. C[A[j]] = C[A[j]] + 1;

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

	0	1	2	3	4	5
C:	2	0	2	3	0	1

	1	2	3	4	5	6	7	8
B:								

End of Loop 2

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

	0	1	2	3	4	5
C:	2	0	2	3	0	1

	1	2	3	4	5	6	7	8
B:								

Executing Loop 3

```
7.  for i=1 to k
8.      C[i] = C[i] + C[i-1];
```

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

	0	1	2	3	4	5
C:	2	0	2	3	0	1
C:	2	2	2	3	0	1

	1	2	3	4	5	6	7	8
B:								

Executing Loop 3

```
7.  for i=1 to k
8.      C[i] = C[i] + C[i-1];
```

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

	0	1	2	3	4	5
C:	2	2	2	3	0	1
C:	2	2	4	3	0	1

	1	2	3	4	5	6	7	8
B:								

Executing Loop 3

```
7.  for i=1 to k
8.      C[i] = C[i] + C[i-1];
```

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

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

	1	2	3	4	5	6	7	8
B:								

Executing Loop 3

```
7.  for i=1 to k
8.      C[i] = C[i] + C[i-1];
```

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

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

	1	2	3	4	5	6	7	8
B:								

Executing Loop 3

```
7.  for i=1 to k
8.      C[i] = C[i] + C[i-1];
```

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

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

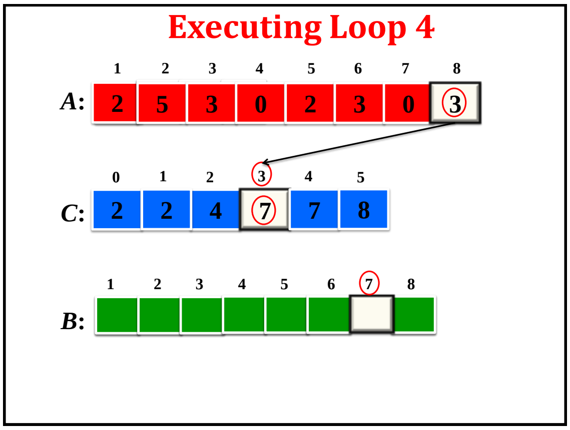
	1	2	3	4	5	6	7	8
B:								

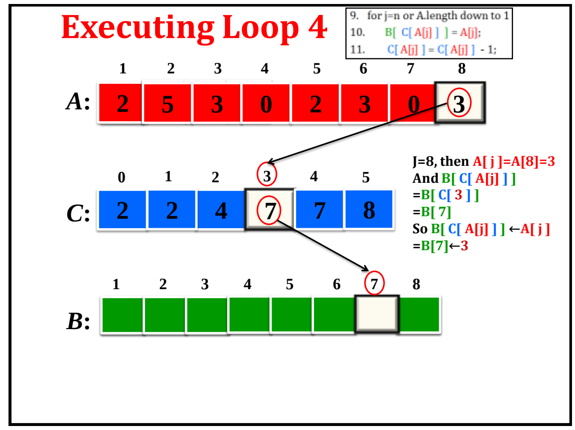
End of Loop 3

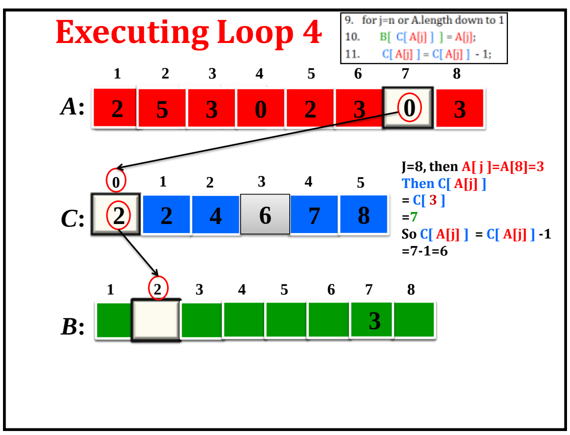
	1	2	3	4	5	6	7	8
A:	2	5	3	0	2	3	0	3

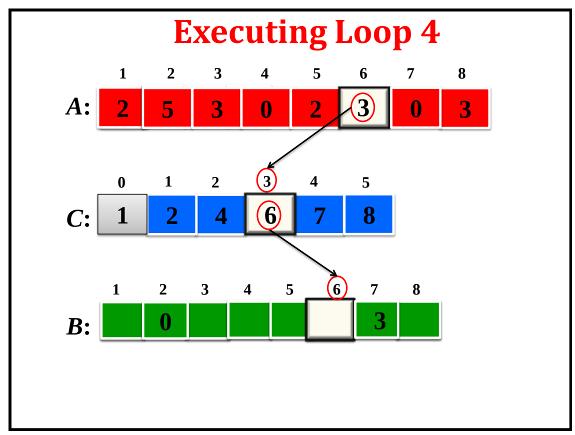
	0	1	2	3	4	5
C:	2	2	4	7	7	8

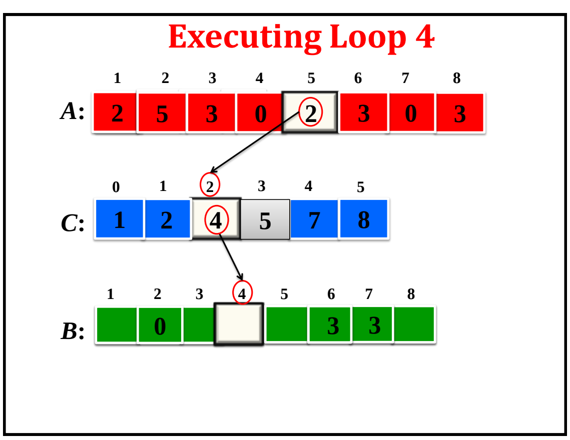
	1	2	3	4	5	6	7	8
B:								

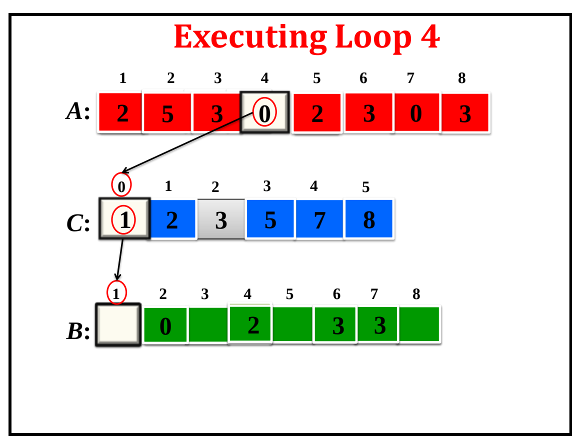

Executing Loop 4
A: 2 5 3 0 2 3 0 3
C: 2 2 4 7 7 8
B:


Executing Loop 4
9. for j=n or A.length down to 1
10. B[C[A[j]]] = A[j];
11. C[A[j]] = C[A[j]] - 1;
A: 2 5 3 0 2 3 0 3
C: 2 2 4 7 7 8
B:
J=8, then A[j]=A[8]=3
And B[C[A[j]]]
=B[C[3]]
=B[7]
So B[C[A[j]]] ←A[j]
=B[7]←3

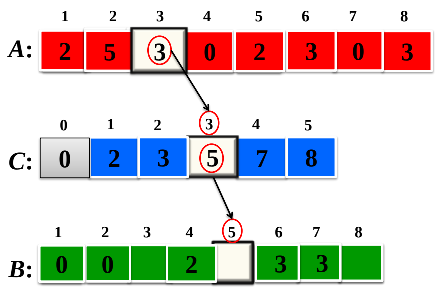

Executing Loop 4
9. for j=n or A.length down to 1
10. B[C[A[j]]] = A[j];
11. C[A[j]] = C[A[j]] - 1;
A: 2 5 3 0 2 3 0 3
C: 2 2 4 6 7 8
B: 3
J=8, then A[j]=A[8]=3
Then C[A[j]]
= C[3]
=7
So C[A[j]] = C[A[j]] -1
=7-1=6


Executing Loop 4
A: 2 5 3 0 2 3 0 3
C: 1 2 4 6 7 8
B: 0 3

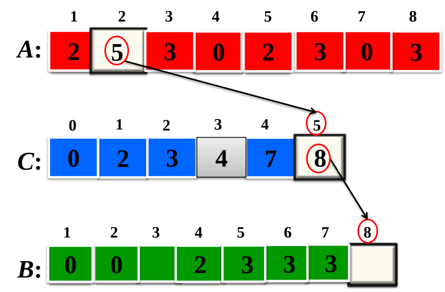

Executing Loop 4
A: 2 5 3 0 2 3 0 3
C: 1 2 4 5 7 8
B: 0 3 3


Executing Loop 4
A: 2 5 3 0 2 3 0 3
C: 1 2 3 5 7 8
B: 0 2 3 3

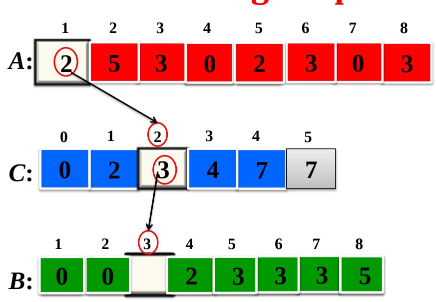
Executing Loop 4



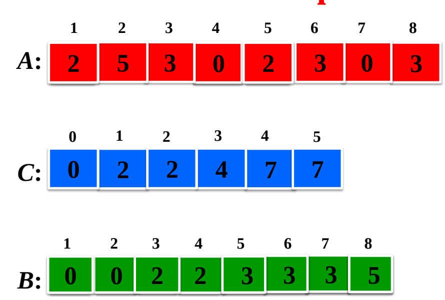
Executing Loop 4



Executing Loop 4



End of Loop 4



Sorted data in Array B

Time Complexity Analysis

1. Counting-Sort(A, B, k)
2. Let C[0.....k] be a new array
3. for i=0 to k **[Loop 1]**
4. C[i]= 0;
5. for j=1 to A.length or n **[Loop 2]**
6. C[A[j]] = C[A[j]] + 1;
7. for i=1 to k **[Loop 3]**
8. C[i] = C[i] + C[i-1];
9. for j=n or A.length down to 1 **[Loop 4]**
10. B[C[A[j]]] = A[j];
11. C[A[j]] = C[A[j]] - 1;

Loop 1 and 3 takes O(k) time

Loop 2 and 4 takes O(n) time

```
void CountingSort (int A[], int n, int B[], int K) {
    int C[K], i, j;
    //Complexity: O(K)
    for (i =0 ; i<K; i++)
        C[i] = 0;
    //Complexity: O(n)
    for (j =0 ; j<n; j++)
        C[A[j]] = C[A[j]] + 1;
    //C[i] now contains the number of elements equal to i
    //Complexity: O(K)
    for (i =1 ; i<K; i++)
        C[i] = C[i] + C[i-1];
    // C[i] now contains the number of elements ≤ i
    //Complexity: O(n)
    for (j = n-1; j>=0; j--) {
        B[C[A[j]]] = A[j];
        C[A[j]] = C[A[j]] - 1;
    }
}
```

Time Complexity Analysis

- **So the counting sort takes a total time of:**
- =O(k) + O(n) + O(k) + O(n) = O(n + k)
- Usually, k = O(n)
  - Thus counting sort runs in O(n) time
- Counting sort is called stable sort.

Radix Sort

- Radix sort is non comparative sorting method
- Two classifications of radix sorts are **least significant digit (LSD)** radix sorts and most significant digit (**MSD**) radix sorts.
- **LSD radix sorts** process the integer representations starting from the least digit and move towards the most significant digit.

39

Radix Sort

In input array A, each element is a number of d digit.

Radix - Sort(A, d)

for $i \leftarrow 1$ to d

do "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

- What sort will we use to sort on digits?
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from 1..k
  - Time: O(n + k)
- Each pass over n numbers with **d digits takes** time O(n+k), so total time **O(dn+dk)**
  - When d is constant and k=O(n), takes O(n) time
- **How many bits in a computer word?**

Median Finding Algorithm

Problem Definition

- **Given** a set of "n" unordered numbers we want to find the "k" smallest number. (k is an integer between 1 and n).

A Simple Solution

- A simple sorting algorithm (like heapsort) will take Order of O(nlg$_2$n) time.

Step	Running Time
Sort n elements (using heapsort)	O(nlog$_2$n)
Return the kth smallest element	O(1)
Total running time	O(nlog$_2$n)

Linear Time selection algorithm

- Also called **Median Finding Algorithm**.
- Find k th smallest element in O (n) time in worst case.
- Uses Divide and Conquer strategy.
- Uses elimination in order to cut down the running time substantially.

Steps to solve the problem

- Step 1: If n is small, for example n<6, just sort and return the kth smallest number in constant time i.e; O(1) time.
- Step 2: Group the given number in subsets of 5 in O(n) time.
- Step3: Sort each of the group in O (n) time. Find median of each group.

Algorithm

- Given array A of size n and integer k≤n,
  1. Group the array into n/ 5 groups of size 5 and find the median of each group. (For simplicity, we will ignore integrality issues.)
  2. Recursively, find the true median of the medians. Call this p
  3. Use p as a pivot to split the array into subarrays LESS and GREATER.
  4. Recursive on the appropriate piece.
- O(n) comparisons to find the k th smallest in an array of size n

Theorem 4.2 *DeterministicSelect makes $O(n)$ comparisons to find the kth smallest in an array of size n.*

Proof: Let $T(n,k)$ denote the worst-case time to find the kth smallest out of n, and $T(n) = \max_k T(n,k)$ as before.

Step 1 takes time $O(n)$, since it takes just constant time to find the median of 5 elements. Step 2 takes time at most $T(n/5)$. Step 3 again takes time $O(n)$. Now, we claim that at least $3/10$ of the array is $\leq p$, and at least $3/10$ of the array is $\geq p$. Assuming for the moment that this claim is true, Step 4 takes time at most $T(7n/10)$, and we have the recurrence:

$$T(n) \leq cn + T(n/5) + T(7n/10), \quad (4.1)$$

Description of the Algorithm step

- If n is small, for example n<6, just sort and return the k the smallest number.(Bound time- 7)
- If n>5, then partition the numbers into groups of 5.(Bound time n/5)
- Sort the numbers within each group. Select the middle elements (the medians). (Bound time- 7n/5)
- Call your "Selection" routine recursively to find the median of n/5 medians and call it m. (Bound time-$T_{n/5}$)
- Compare all n-1 elements with the median of medians m and determine the sets L and R, where L contains all elements <m, and R contains all elements >m. (Bound time- n)

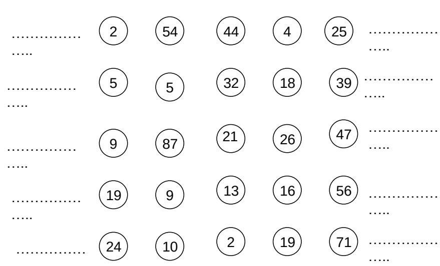
Recursive formula

- T (n)=O (n) + T (n/5) +T (7n/10)
- T(n) <= c7n/10+ cn/5 + O(n)
- T (n) = Θ (n)

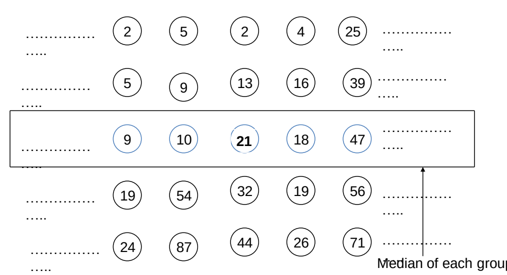
Example

- Given a set (........2,5,9,19,24,54,5,87,9,10,44,32,21,13,24,18,26,16,19,25,39,47,56,71,91,61,44,28.........) having n elements.

Arrange the numbers in groups of five



Find median of N/5 groups



Find the Median of each group