Lecture 14

Sorting in Linear Time: Lower Bounds for Sorting, Counting Sort, and Radix Sort

Fundamental Question

- Suppose that we just designed an algorithm that takes time f(n) to solve some problem, where n:= Problem Size
- Is it the best solution?
- Desirable result:
  - Every algorithm for solving the problem must take time at least $\Omega(f(n)$ … *Problem Lower Bound*
- Time taken by an algorithm=Problem Lower bound
  - We have the BEST possible algorithm

Prior Knowledge: Lower Bounds on Algorithm

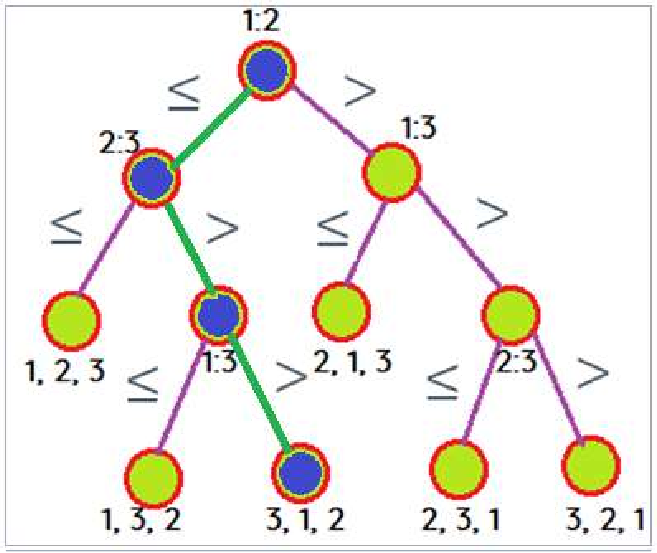
- An Algorithm A for a Problem P is said to have a lower bound f_A if worst-case max_i time for Algorithm A on instance i of size $n \geq f_A(n)$.
- **How to Prove:** Construct instance i taking large time, $\forall\, n$.
- Problem Lower Bound:
  - A Problem P is said to have a lower bound if every possible algorithm has lower bound f.
  - More algebraically, the algorithm that takes the minimum worst-case time (min_A) must have time bigger (max_i) than f(n).
    - i.e. $min_A\, max_i$ time for algorithm A on instance i of size n ≥ f(n).
  - Clearly, Every algorithm will have time bigger than f(n) for a problem P.
  - It means that f to be problem lower bound for this problem P.
- How to Prove: we need to create BAD instances i must be constructed $\forall\, n$, $\forall\, A$

Trivial & Non-Trivial Problem Lower Bound

- Trivial
  - Most problems have $\Omega(n)$ lower bound, n=instance size
  - n time is required to at least read all the input – No matter what algorithm is used.
    - In few problem uninteresting problems, this may not apply.
- Non-Trivial
  - Very difficult on RAM (Random Access Machines)
  - Space of all possible algorithms is huge, tricky to analyze
    - RAM has many instructions
    - Many Control flow patterns, looping, recursion
- Space of algorithms is easier to analyze on simpler computational models.

The Decision Tree Model

- Input
  - A sequence of numbers $x_1, x_2, \ldots, x_n$. Already read.
- Program: Labelled Tree
  - Non-leaf node labels: $i\,, j$
  - Edge Labels: $<, >, \neq, \leq, \geq$
  - Leaf node Labels: value to be output
- Program tree for sorting 3 numbers
- Execution: Begins at the root, At node i:j, x_i is compared with x_j. Execution follows branch with the appropriate label. Leaf node shows the output
  - For example: x_1=20, x_2=30, x_3=10



- Time taken by the program: Number of comparisons performed. Worst-case time=length of longest path in the program tree, Average case time=Average root-leaf path length

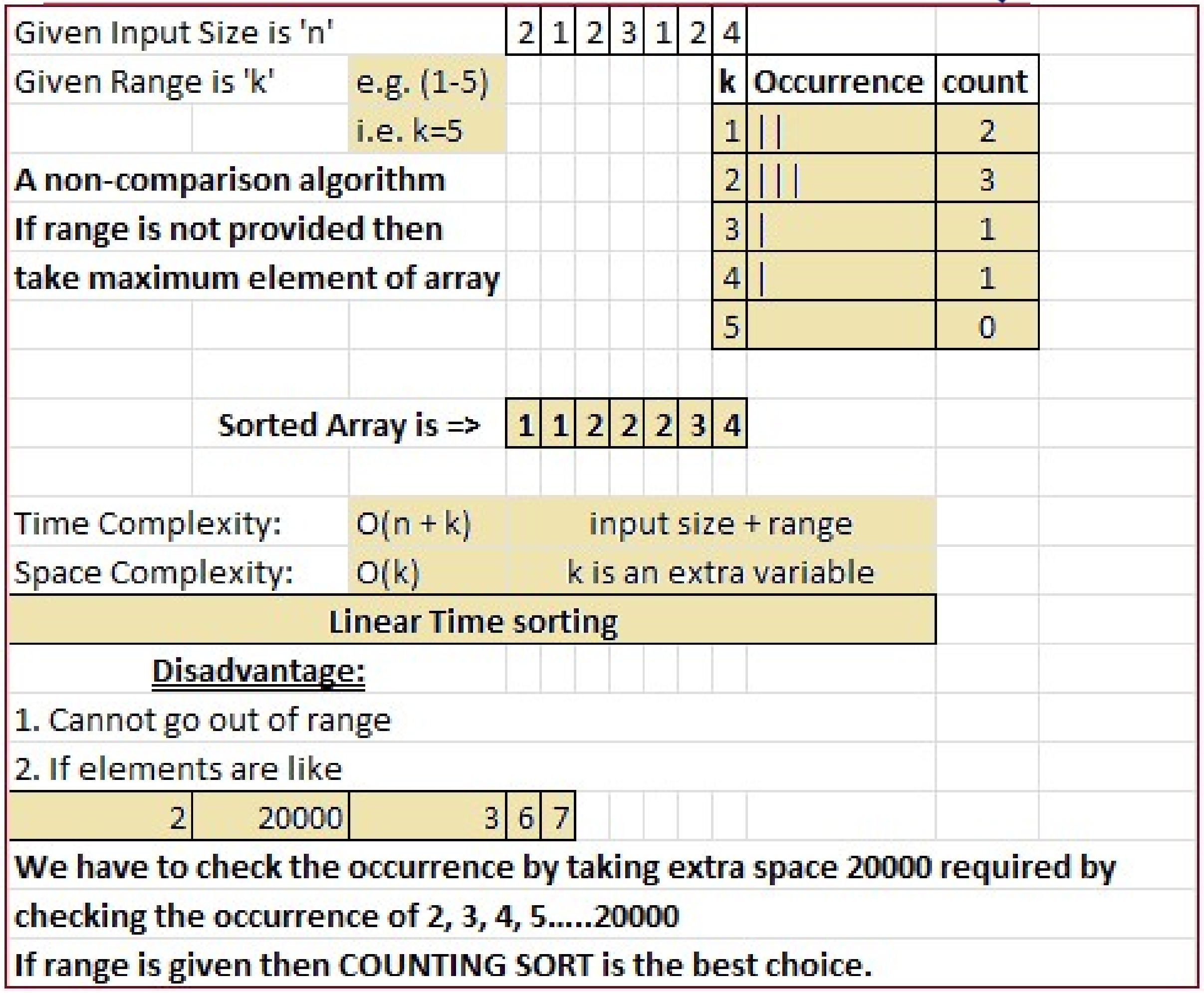
Our Claim: Sorting takes time Ω(n log n) in the decision tree model?

- Proof:
  - Suppose that Input $x = \pi(1, 2, 3, \ldots, n)$ *where* π *is permutation*.
  - Thus, $(1, 2, 3, \ldots, n) = \pi^{-1}(x)$.
    Clearly, the algorithm must output the answer π^{-1} *for this input*
  - Thus, π^{-1} *must appear as the label of at least one leaf*. This holds for all permutations ➔ tree at least has n! leaves.
  - Each tree node can only have at most 3 outgoing edges: with labels <, = or >. Thus, height is ≥ $\log_3 n!$.
  - Only 2 edges will be used if input is permutation of $(1, 2, 3, \ldots, n)$. So, height ≥ $\log_2 n!$ ≥ $\log(n/2)^{n/2}$

Can we do better?

- *Linear sorting algorithms*
  - ***Counting Sort***
  - ***Radix Sort***
  - *Bucket sort – Self learning*
- *Make certain assumptions about the data*
- *Linear sorts are NOT "comparison sorts"*

Counting Sort Algorithm



Given Input Size is 'n'

2	1	2	3	1	2	4

Given Range is 'k' e.g. (1-5)
i.e. k=5

k	Occurrence	count
1	||	2
2	|||	3
3	|	1
4	|	1
5		0

A non-comparison algorithm
If range is not provided then
take maximum element of array

Sorted Array is =>

1	1	2	2	2	3	4

Time Complexity:	O(n + k)	input size + range
Space Complexity:	O(k)	k is an extra variable

Linear Time sorting

Disadvantage:

1. Cannot go out of range
2. If elements are like

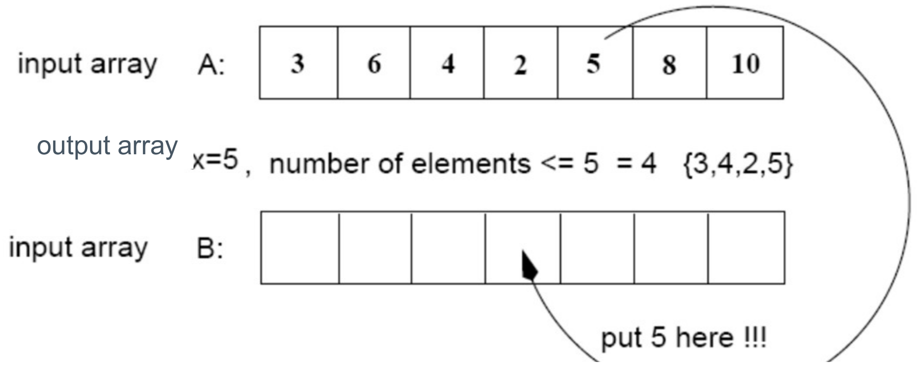
2	20000	3	6	7

We have to check the occurrence by taking extra space 20000 required by checking the occurrence of 2, 3, 4, 5.....20000

If range is given then COUNTING SORT is the best choice.

Counting Sort

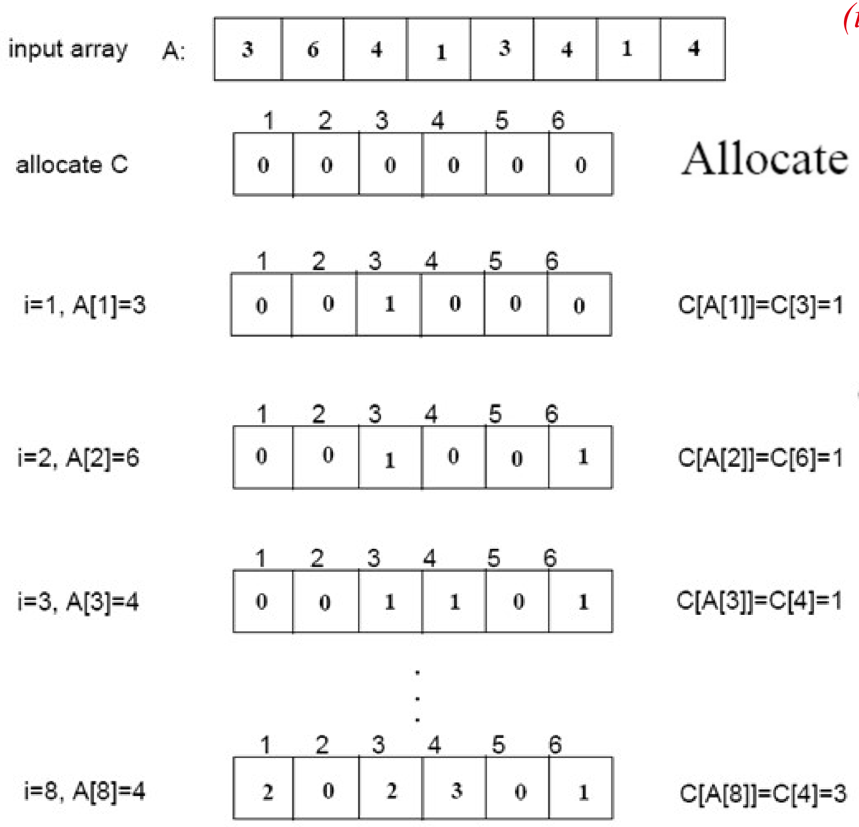
- *Assumptions:*
  - *n integers which are in the range [0 ... r]*
  - *r is in the order of n, that is, r=O(n)*
- *Idea:*
  - *For each element x, find the number of elements ≤ x*
  - *Place x into its correct position in the output array*



Step 1

Find the number of times $A[i]$ appears in A

(i.e., frequencies)



input array A:

3	6	4	1	3	4	1	4

allocate C

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

Allocate $C[1..r]$ (r=6)

i=1, A[1]=3

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

C[A[1]]=C[3]=1

For $1 \leq i \leq n$, $++C[A[i]];$

C[i] = number of times element i appears in A

i=2, A[2]=6

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

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

i=3, A[3]=4

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

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

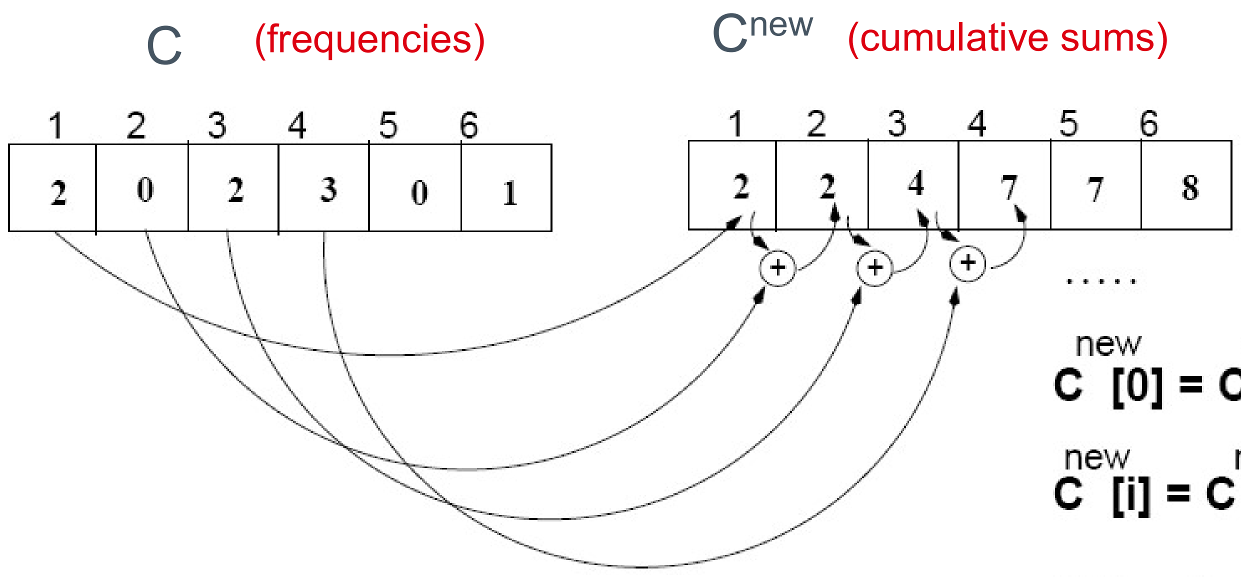
⋮

i=8, A[8]=4

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

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

Step 2 Find the number of elements $\leq A[i]$,



$C^{new}[0] = C^{old}[0]$

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

C[i] = # elements <= i

Algorithm

- *Start from the last element of A*
- *Place A[i] at its correct place in the output array*
- *Decrease C[A[i]] by one*

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

	0	1	2	3	4	5
C^{new}	2	2	4	7	7	8

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^{new}	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C^{new}	2	2	4	6	7	8

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C^{new}	1	2	4	6	7	8

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

	0	1	2	3	4	5
C^{new}	1	2	4	5	7	8

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

	0	1	2	3	4	5
C^{new}	1	2	3	5	7	8

Example (Cont !!!)

	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	0		2		3	3	

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

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

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

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

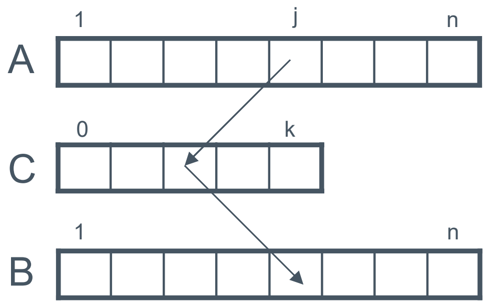
	0	1	2	3	4	5
C	0	2	3	4	7	8

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

COUNTING-SORT

Alg.: COUNTING-SORT(A, B, n, k)

```
1.  for i ← 0 to r
2.      do C[ i ] ← 0
3.  for j ← 1 to n
4.      do C[A[ j ]] ← C[A[ j ]] + 1
5.  ▷ C[i] contains the number of elements equal to i
6.  for i ← 1 to r
7.      do C[ i ] ← C[ i ] + C[i -1]
8.  ▷ C[i] contains the number of elements ≤ i
9.  for j ← n downto 1
10.     do B[C[A[ j ]]] ← A[ j ]
11.         C[A[ j ]] ← C[A[ j ]] - 1
```



Analysis of Counting Sort

Alg.: COUNTING-SORT(A, B, n, k)

```
1.    for i ← 0 to r                                    } O(r)
2.        do C[ i ] ← 0
3.    for j ← 1 to n                                    } O(n)
4. ▷      do C[A[ j ]] ← C[A[ j ]] + 1
5.    C[i] contains the number of elements equal to i
6.    for i ← 1 to r
7. ▷      do C[ i ] ← C[ i ] + C[i -1]                  } O(r)
8.    C[i] contains the number of elements ≤ i
9.    for j ← n downto 1                                } O(n)
10.       do B[C[A[ j ]]] ← A[ j ]
11.               C[A[ j ]] ← C[A[ j ]] - 1
```

Overall time: $O(n + r)$

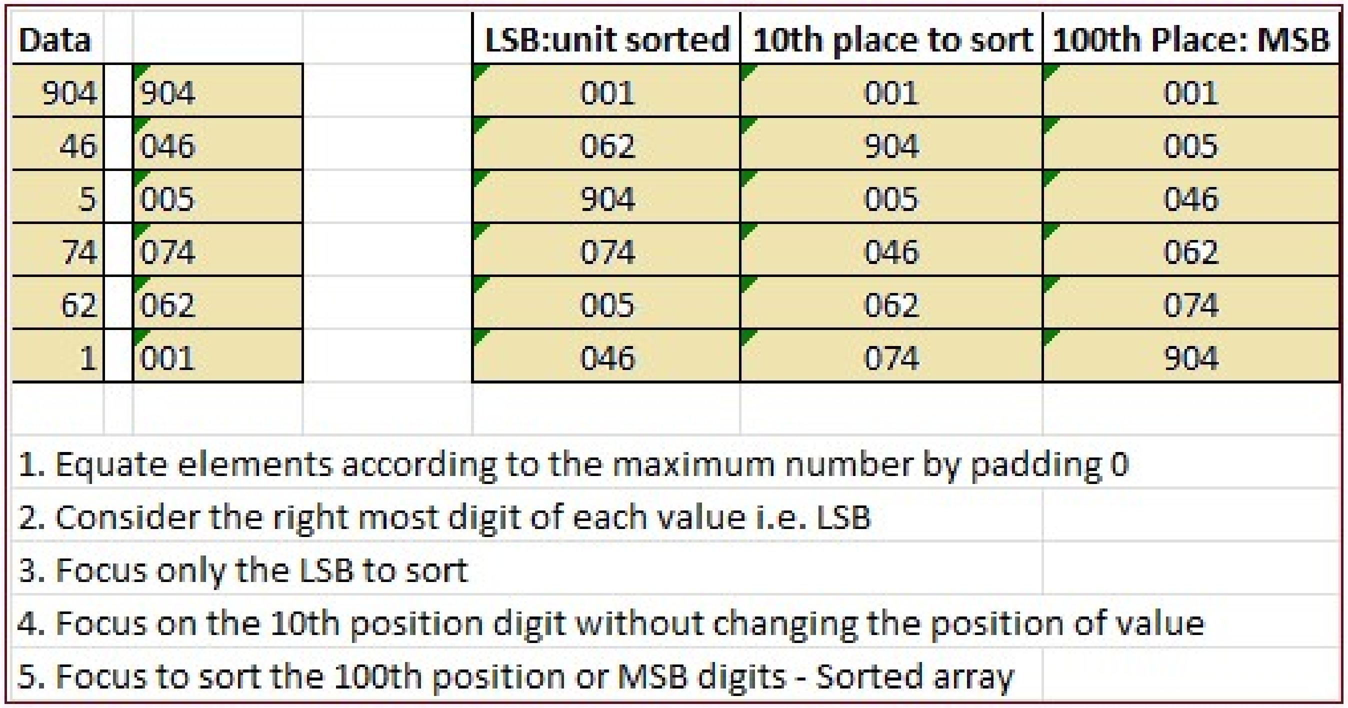
Analysis of Counting Sort

- *Overall time: $O(n + r)$*

- *In practice we use COUNTING sort when $r = O(n)$*

  $\Rightarrow$ *running time is $O(n)$*

Radix Sort: Radix means the base



Data				LSB:unit sorted	10th place to sort	100th Place: MSB
904		904		001	001	001
46		046		062	904	005
5		005		904	005	046
74		074		074	046	062
62		062		005	062	074
1		001		046	074	904

1. Equate elements according to the maximum number by padding 0
2. Consider the right most digit of each value i.e. LSB
3. Focus only the LSB to sort
4. Focus on the 10th position digit without changing the position of value
5. Focus to sort the 100th position or MSB digits - Sorted array

Time Complexity: O(dn)

Space Complexity: $O(n + 2^d) = O(n + k)$

Radix Sort

- Represents keys as *d*-digit numbers in some base-*k*

$$key = x_1x_2\ldots x_d \quad \text{where } 0 \le x_i \le k-1$$

- Example: key=15

$$key_{10} = 15,\ d=2,\ k=10 \quad \text{where } 0 \le x_i \le 9$$

$$key_2 = 1111,\ d=4,\ k=2 \quad \text{where } 0 \le x_i \le 1$$

Radix Sort

- *Assumptions*

  d=O(1)   and k =O(n)

- *Sorting looks at one column at a time*
  - *For a d digit number, sort the least significant digit first*
  - *Continue sorting on the next least significant digit,*
    - *until all digits have been sorted*
  - *Requires only d passes through the list*

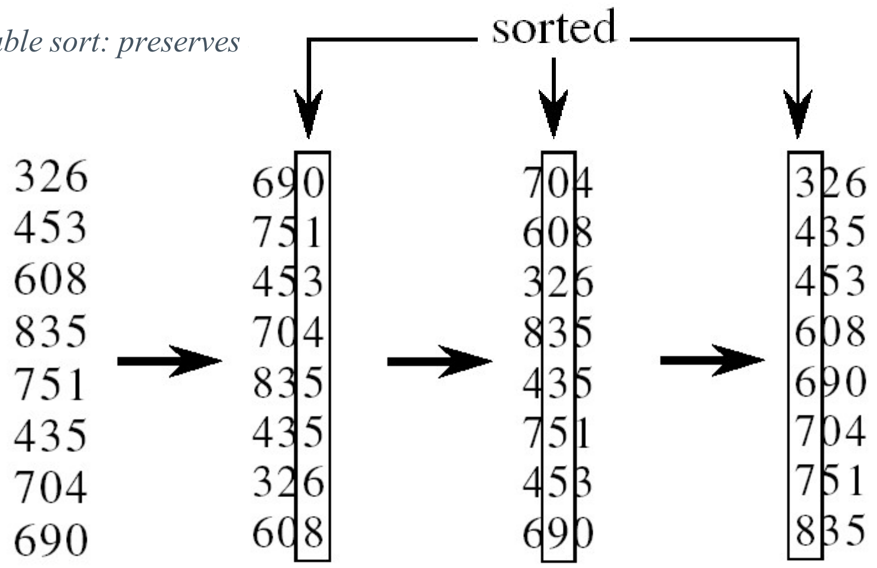
326
453
608
835
751
435
704
690

RADIX-SORT

Alg.: RADIX-SORT(A, d)

```
for i ← 1 to d
    do use a stable sort to sort array A on digit i
```

(stable sort: preserves



Analysis of Radix Sort

- *Given n numbers of d digits each, where each digit may take up to k possible values, RADIX-SORT correctly sorts the numbers in O(d(n+k))*
  - *One pass of sorting per digit takes O(n+k) assuming that we use* ***counting sort***
  - *There are d passes (for each digit)*

Analysis of Radix Sort

- *Given n numbers of d digits each, where each digit may take up to k possible values, RADIX-SORT correctly sorts the numbers in $O(d(n+k))$*
  - *Assuming $d=O(1)$ and $k=O(n)$, running time is $O(n)$*

Thank You!!!

Have a good day