CME 2001
Data Structures and Algorithms

Zerrin Işık
zerrin@cs.deu.edu.tr

Sorting in Linear Time

How fast can we sort?

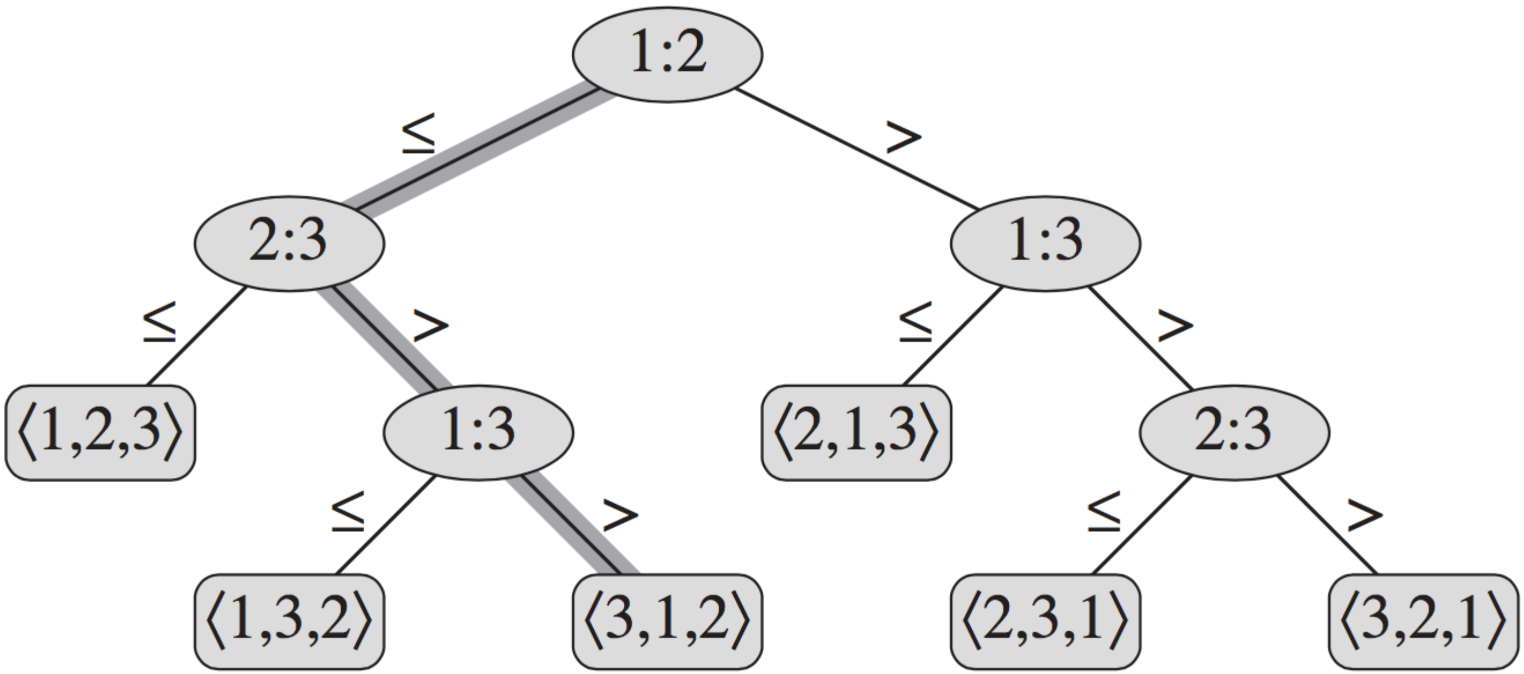
All sorting algorithms we have seen so far are comparison sorts: use comparison to find the relative order of elements

e.g. Insertion sort, Heapsort, Quicksort, Mergesort.

The best "worst-case" running time was $O(n.lgn)$.

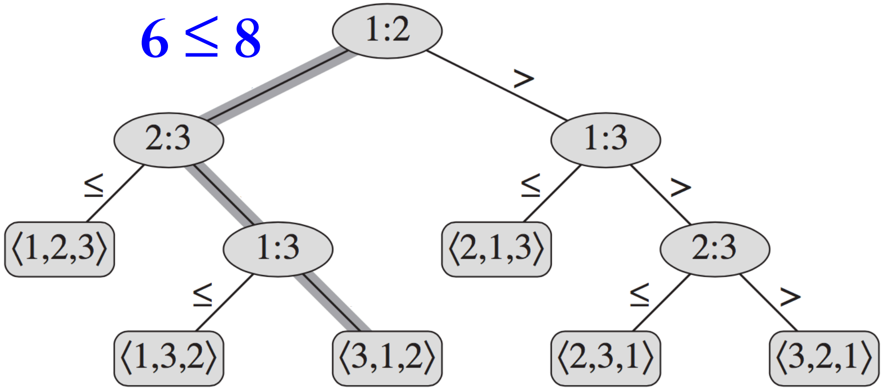
=> *Can we do better than $O(n.lgn)$ running time?*

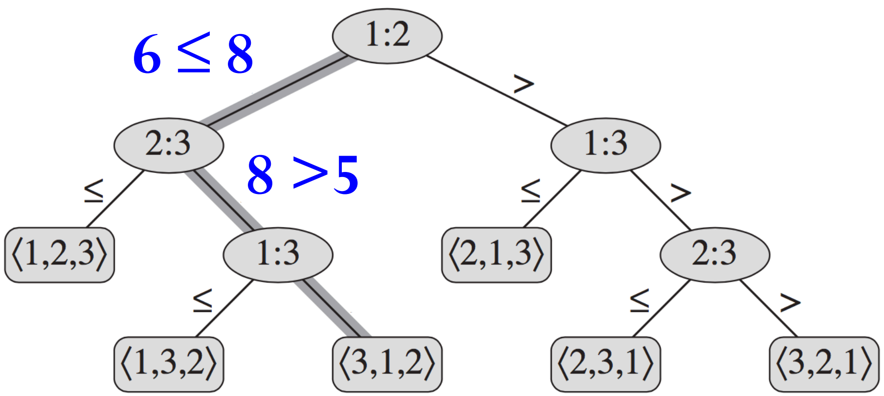
Decision-tree example

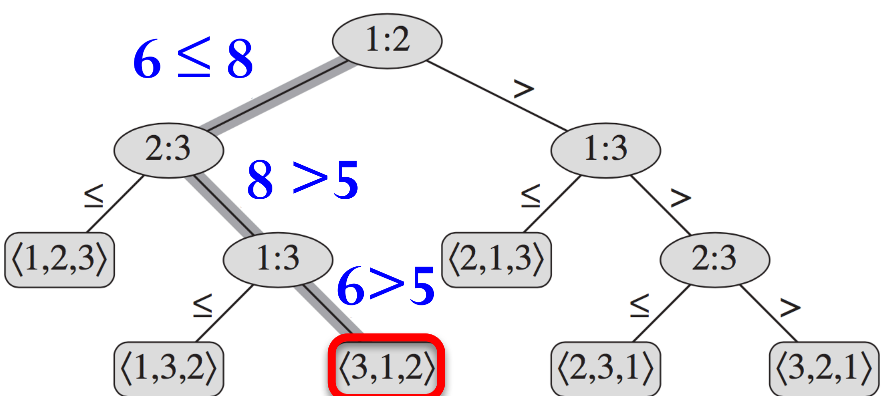


- An internal node is labeled by i:j indices a comparison between a_i and a_j.
- A leaf is labeled by the permutation indicates the orders that the algorithm determines.

Input A = [6,8,5]







Sorted order : **A (3, 1, 2)**

[5,6,8]

Decision-tree model

- It is an abstraction of any comparison sort.
- Tree contains the comparisons along all possible instructions traces e.g., <1,2,3> <3,2,1> <2,1,3> …

- The running time : the length of the path taken.
  - e.g., A = [6,8,5] => path-length=3 => O(3)
- Worst-case : height of the tree

Lower bound for decision-tree sorting

Theorem: Any decision that sorts ***n*** elements should have a height Ω(n lgn).

Lemma

Any binary tree of height h has $\leq 2^h$ leaves.

In other words:

- $l = \#$ of leaves,
- $h =$ height,
- Then $l \leq 2^h$.

Proof

- $l \geq n!$
- By lemma, $n! \leq l \leq 2^h$ or $2^h \geq n!$
- Take logs: $h \geq \lg(n!)$
- Use Stirling's approximation: $n! > (n/e)^n$

$$\begin{aligned} h &\geq \lg(n/e)^n \\ &= n\lg(n/e) \\ &= n\lg n - n\lg e \\ &= \Omega(n\lg n)\ . \qquad \blacksquare \end{aligned}$$

Counting Sort

- Assumes each of the $\boldsymbol{n}$ input elements is an integer in the range $\boldsymbol{0}$ to $\boldsymbol{k}$.
- When $k= O(n)$, the sort runs in $\Theta(n)$ time.
- No comparisons between elements.

Input: $A[1 \ldots n]$, where $A[j]$ is between 0 to k, for $j=1,2,\ldots,n$.
Array A and values n and k are given as parameters.
Output: $B[1 \ldots n]$, sorted. B is assumed to be already allocated and is given as a parameter.

- **Auxiliary storage:** $C[1 \ldots k]$

```
COUNTING-SORT(A, B, n, k)
  let C[0..k] be a new array
  for i = 0 to k
      C[i] = 0
  for j = 1 to n
      C[A[j]] = C[A[j]] + 1
  for i = 1 to k
      C[i] = C[i] + C[i - 1]
  for j = n downto 1
      B[C[A[j]]] = A[j]
      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

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

let $C[0\,..\,k]$ be a new array
for $i = 0$ **to** k
  $C[i] = 0$
for $j = 1$ **to** n
  $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k
  $C[i] = C[i] + C[i-1]$
for $j = n$ **downto** 1
  $B[C[A[j]]] = A[j]$
  $C[A[j]] = C[A[j]] - 1$

	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

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

$\text{COUNTING-SORT}(A, B, n, k)$

let $C[0\,..\,k]$ be a new array
for $i = 0$ **to** k
    $C[i] = 0$
for $j = 1$ **to** n
    $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k
    $C[i] = C[i] + C[i-1]$
for $j = n$ **downto** 1
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

	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	2	4	7	7	8

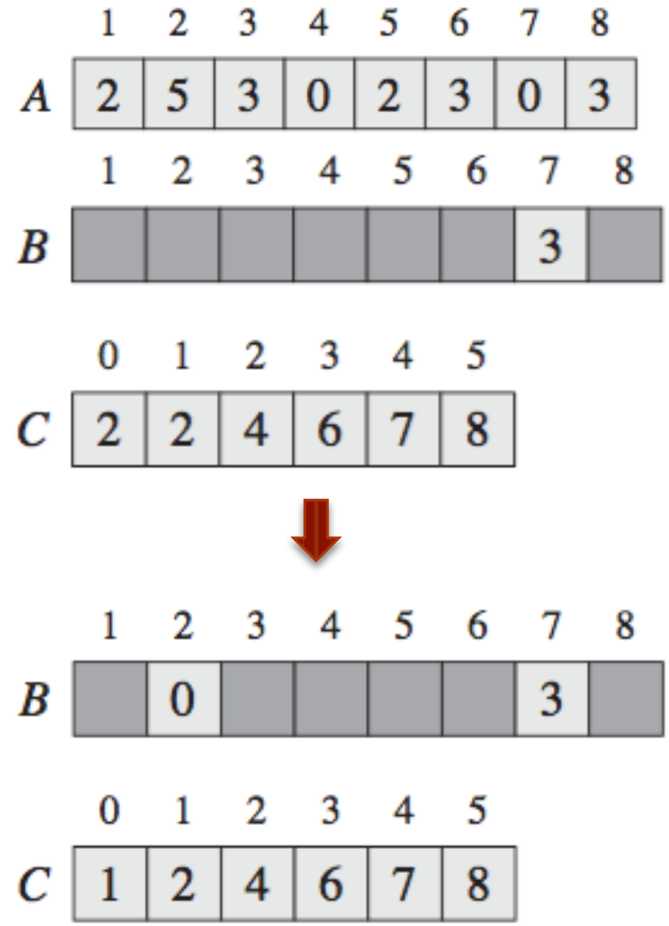
	1	2	3	4	5	6	7	8
B							3	

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

After
1st iteration

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

let $C[0\,..\,k]$ be a new array
for $i = 0$ **to** k
    $C[i] = 0$
for $j = 1$ **to** n
    $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k
    $C[i] = C[i] + C[i-1]$
for $j = n$ **downto** 1
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$



After
2nd iteration

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

let $C[0\,..\,k]$ be a new array
for $i = 0$ **to** k
    $C[i] = 0$
for $j = 1$ **to** n
    $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k
    $C[i] = C[i] + C[i-1]$
for $j = n$ **downto** 1
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

	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					3	

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

After
3rd iteration

Final Sorted Array

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

Counting Sort Analysis

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

let $C[0\,..\,k]$ be a new array

$$\Theta(k)\left\{\begin{array}{l}\textbf{for } i = 0 \textbf{ to } k \\ \qquad C[i] = 0\end{array}\right.$$

$$\Theta(n)\left\{\begin{array}{l}\textbf{for } j = 1 \textbf{ to } n \\ \qquad C[A[j]] = C[A[j]] + 1\end{array}\right.$$

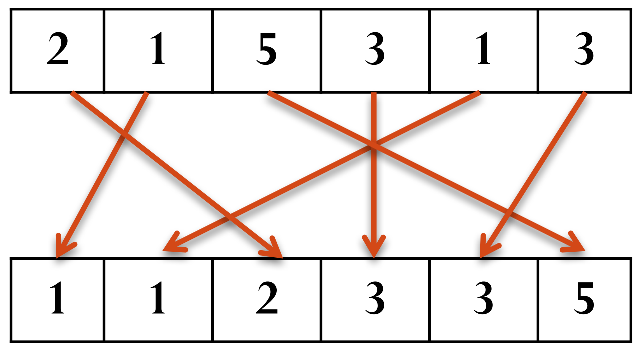
$$\Theta(k)\left\{\begin{array}{l}\textbf{for } i = 1 \textbf{ to } k \\ \qquad C[i] = C[i] + C[i-1]\end{array}\right.$$

$$\Theta(n)\left\{\begin{array}{l}\textbf{for } j = n \textbf{ downto } 1 \\ \qquad B[C[A[j]]] = A[j] \\ \qquad C[A[j]] = C[A[j]] - 1\end{array}\right.$$

$\mathbf{\Theta(n+k)}$

Running time

- Counting sort beats the lower bound of $\Omega(n \lg n)$, because it does not make any comparison.
- It uses actual values of elements to index into an array.
- It is ***stable***: preserves the input order among equal elements.



- But, not so efficient to sort relatively big numbers > 8-bit!
- Stability feature supports the usage of counting sort as a subroutine in radix sort.

Radix Sort

- Treats each data item as a digit (or a character string).
- First sort data items according to their rightmost digit.
- Then, combine these groups.
- Use a ***stable*** sorting algorithm.

RADIX-SORT (A,d)

for i=1 to d

use a stable sorting to sort array ***A*** on digit ***i***

Radix Sort Example

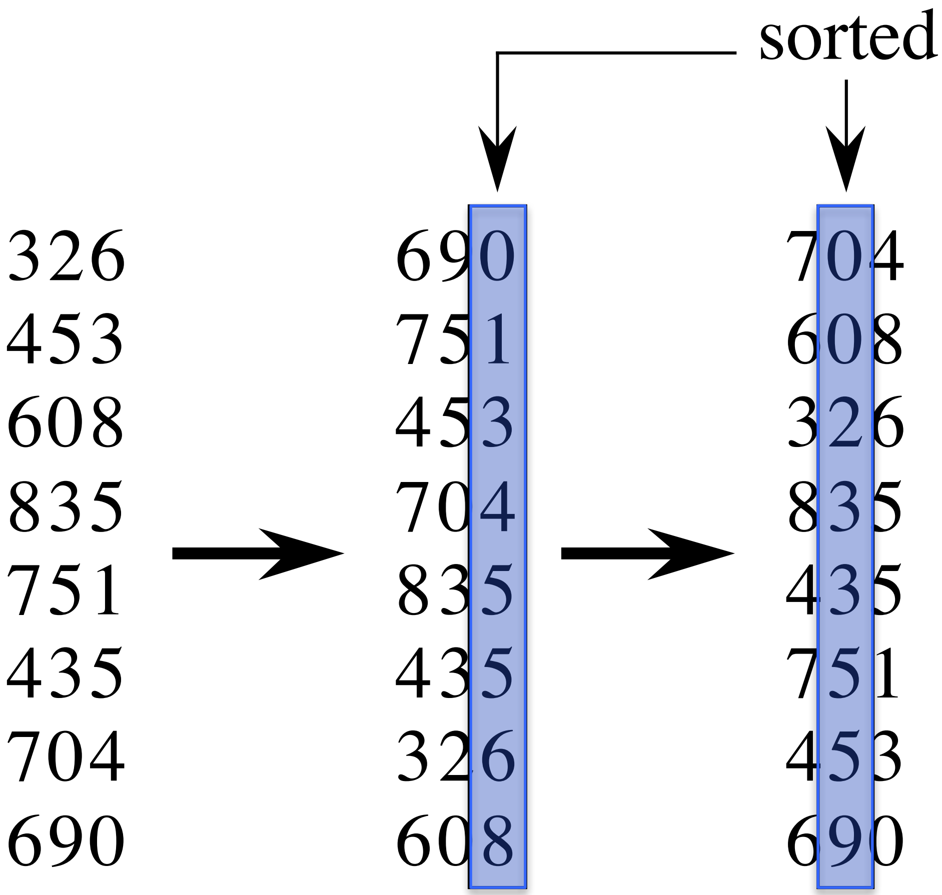
326
453
608
835
751
435
704
690

Radix Sort Example

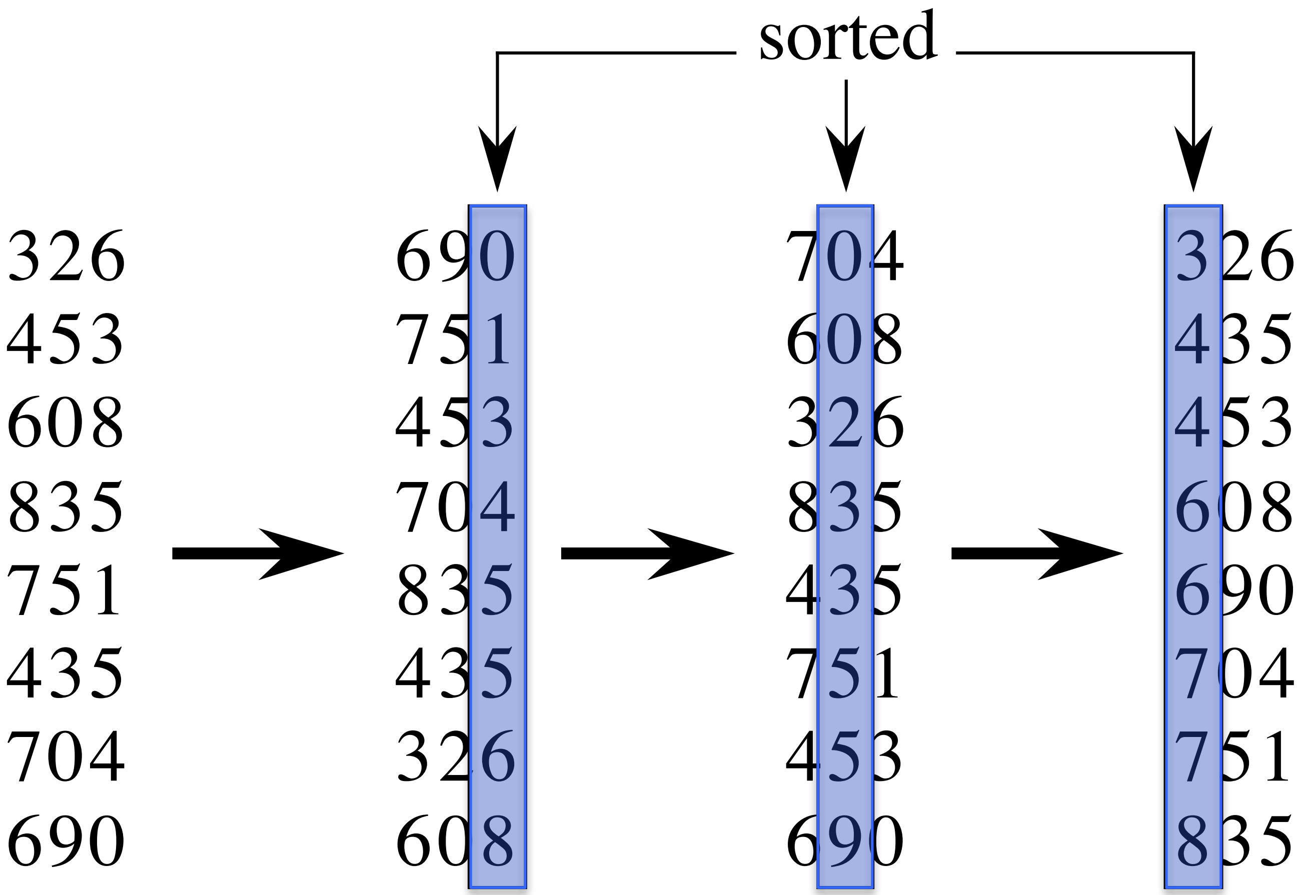
sorted

326	690
453	751
608	453
835	704
751	835
435	435
704	326
690	608

Radix Sort Example



Radix Sort Example



Radix Sort Analysis

Assume that counting sort is used for the intermediate sort.

- Time: $\Theta(n+k)$ per pass (digits in range $0,\ldots,k$) and needs ***d*** passes for ***d*** digits => $\Theta(d(n+k))$.

How to break each key into digits?

- ***n*** words, ***b*** bits per word.
- Break into ***r***-bit digits, will have ***d*** = ***b/r***.
- In counting sort, ***k*** = $2^r - 1$.
  - Example: 32-bit words, 8-bit digits,

    b = 32, r =8 ß d=32/8= 4 k = $2^r - 1$ = 255.
- Time : $\Theta(d(n+k))$ => $\Theta(b/r\ (n+2^r))$.
- Balance ***b/r*** and $n+2^r$: e.g., choosing r ≈ lgn => $\Theta(bn/\lg n)$.

Evaluation

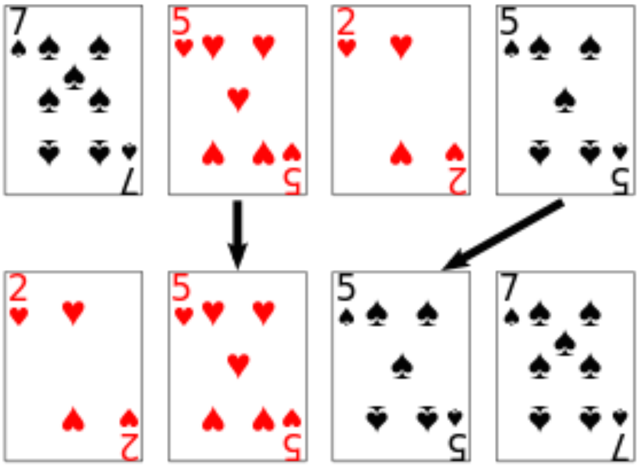
- In practice, radix sort is quite fast for large inputs.

E.g.,

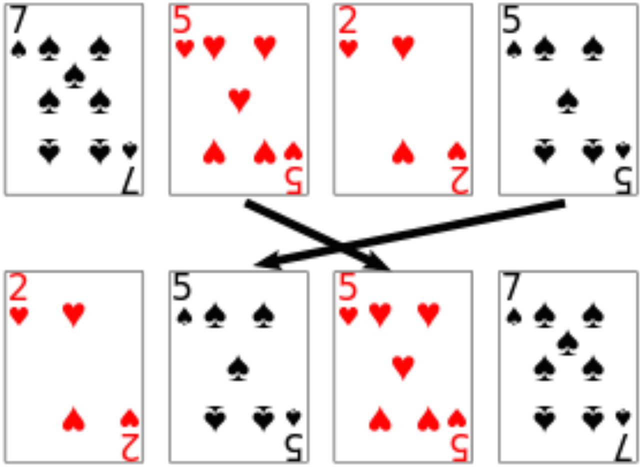
- 1 million (2^{20}) 32-bit integers.
- Radix sort: $d = b/r \;\Rightarrow\; 32/20 \approx 2$ passes.
- Merge sort / Quicksort: $\lg n = 20$ passes.

Stable sorting ?

- To preserve the input order among equal elements.



Stable



No stable

In-place algorithm ?

- If an algorithm transforms (e.g., sorts) the given inputs by using a small extra amount of storage (data structure), then this algorithm is called ***in-place*.**
  - or without using extra storage – much better

- Such algorithms *usually* overwrites the given inputs.

Stable / In-place Algorithms

Algorithm	Stable	In-place
Insertion sort	Yes	Yes
Merge sort	Yes	No
Heapsort	No	Yes
Quicksort	No	Yes
Counting sort	Yes	No
Radix sort	Yes	No