

# **SORTING ALGORITHMS**

**Prof. Zheng Zhang**

**Harbin Institute of Technology, Shenzhen**

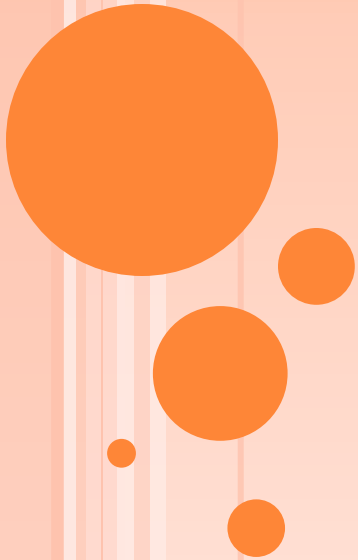


# OUTLINE

- Why We Do Sorting?
- Review of Learned Sorting Algorithms
- How Fast can We Sort so far?
- Linear-Time Sorting Algorithms
  - ✧ Counting Sort
  - ✧ Radix Sort
  - ✧ Bucket Sort



# WHY WE DO SORTING



# WHY WE DO SORTING?

- Commonly encountered programming task in computing.
- Examples of sorting:
  - ✧ List containing exam scores sorted from Lowest to Highest or from Highest to Lowest
  - ✧ List Rank of Milk Powder Producer Brands in China from the Star to Notorious
  - ✧ List Search Results of a User Query by Relevance from the WWW.



# WHY WE DO SORTING?

- Searching for an element in an array will be more efficient. (example: looking up for information like phone number).
- It's always nice to see data in a sorted display. (example: spreadsheet or database application).
- Computers sort things much faster.



# SORTING PROBLEM

## Description:

- **Input:** sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$ .
- **Output:** A permutation  $\langle a'_1, a'_2, \dots, a'_n \rangle$  such that
$$a'_1 \leq a'_2 \leq \dots \leq a'_n.$$

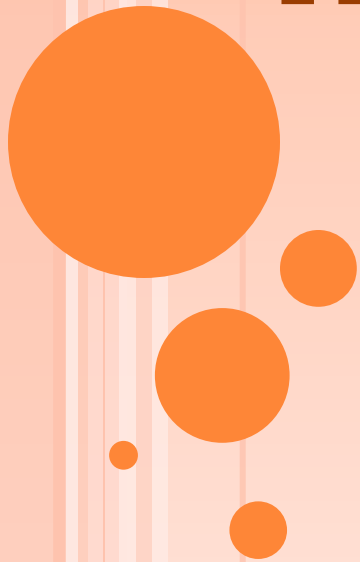
## Example:

***Input:*** 8 2 4 9 3 6

***Output:*** 2 3 4 6 8 9



# LEARNED SORTING ALGORITHMS



# LEARNED SORTING ALGORITHMS

- ⌘ Selection Sort, Heap Sort
- ⌘ Insertion Sort, Binary Insertion Sort,  
Shell Sort
- ⌘ Exchange Sort, Bubble Sort, Quick Sort
- ⌘ Merge Sort
- ⌘ Counting Sort, Radix Sort, Bucket Sort





# Selection Sort

Initial:	265	301	751	129	937	863	742	694	<u>076</u>	438
1st step:	076	301	751	<u>129</u>	937	863	742	694	265	438
2cd step:	076	129	751	301	937	863	742	694	<u>265</u>	438
3rd step:	076	129	265	<u>301</u>	937	863	742	694	751	438
4th step:	076	129	265	301	937	863	742	694	751	<u>438</u>
5th step:	076	129	265	301	438	863	742	<u>694</u>	751	937
6th step:	076	129	265	301	438	694	<u>742</u>	863	751	937
7th step:	076	129	265	301	438	694	742	863	<u>751</u>	937
8th step:	076	129	265	301	438	694	742	751	<u>863</u>	937
9th step:	076	129	265	301	438	694	742	751	863	<u>937</u>

# Insertion Sort

Initial: (265) 301 751 129 937 863 742 694 076 438

1st step: (265 301) 751 129 937 863 742 694 076 438

2cd step: (265 301 751) 129 937 863 742 694 076 438

3rd step: (129 265 301 751) 937 863 742 694 076 438

4th step: (129 265 301 751 937) 863 742 694 076 438

5th step: (129 265 301 751 863 937) 742 694 076 438

6th step: (129 265 301 742 751 863 937) 694 076 438

7th step: (129 265 301 694 742 751 863 937) 076 438

8th step: (076 129 265 301 694 742 751 863 937) 438

9th step: (076 129 265 301 438 694 742 751 863 937)

The diagram illustrates the Insertion Sort algorithm through 10 steps. Each step shows the current state of the array, with elements being compared and shifted. Red dashed arrows indicate the insertion movement of elements. The final sorted array is underlined.

Step	Array
Initial	(265) 301 751 129 937 863 742 694 076 438
1st step	(265 301) 751 129 937 863 742 694 076 438
2cd step	(265 301 751) 129 937 863 742 694 076 438
3rd step	(129 265 301 751) 937 863 742 694 076 438
4th step	(129 265 301 751 937) 863 742 694 076 438
5th step	(129 265 301 751 863 937) 742 694 076 438
6th step	(129 265 301 742 751 863 937) 694 076 438
7th step	(129 265 301 694 742 751 863 937) 076 438
8th step	(076 129 265 301 694 742 751 863 937) 438
9th step	<u>(076 129 265 301 438 694 742 751 863 937)</u>

# SHELL SORT

- Shell sort
  - ✧ An algorithm that first beats the  $O(n^2)$  barrier
  - ✧ Suitable performance for general use
- Very popular
  - ✧ It is the basis of **the default R sort() function**
- Tunable algorithm
  - ✧ Can use different orderings for comparisons



# SHELL SORT

- Donald L. Shell (1959)

- ⌘ A High-Speed Sorting Procedure

- Communications of the Association for Computing Machinery 2:30-32

- ⌘ Systems Analyst working at GE

- ⌘ Back then, most computers read punch-cards

- Also called:

- ⌘ Diminishing increment sort

- ⌘ “Comb” sort

- ⌘ “Gap” sort



# INTUITION/MOTIVATION

- Insertion sort is effective
  - ✧ For small datasets
  - ✧ For data that is nearly sorted
- Insertion sort is inefficient when
  - ✧ Elements must move far in array

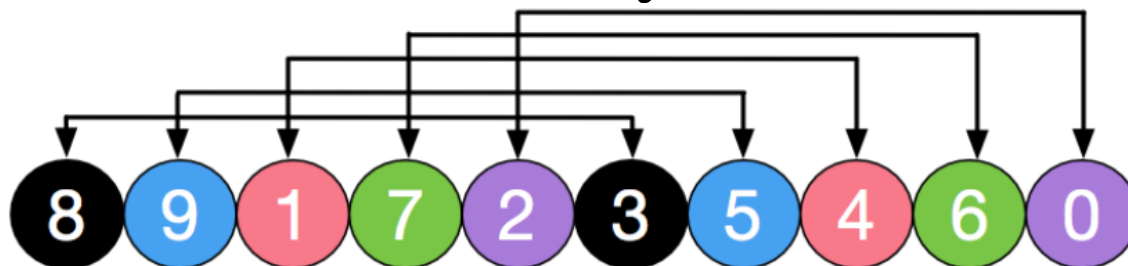


# THE IDEA ...

- Allow elements to move **large steps**
- Bring elements close to final location
  - ✧ First, ensure array is nearly sorted ...
  - ✧ then, run insertion sort ...

How?

Sort interleaved arrays first



# SHELL SORT RECIPE

- Decreasing sequence of **step sizes  $h$** 
  - ⌘ Every sequence must end at 1
  - ⌘ ... , 8, 4, 2, 1
- For each  $h$ , sort sub-arrays that start at arbitrary element and include every  $h$ -th element
  - ⌘ If  $h = 4$ 
    - ⌘ Sub-array with elements 1, 5, 9, 13 ...
    - ⌘ Sub-array with elements 2, 6, 10, 14 ...
    - ⌘ Sub-array with elements 3, 7, 11, 15 ...
    - ⌘ Sub-array with elements 4, 8, 12, 16 ...



# SHELL SORT NOTES

- Any decreasing sequence that ends at  $1$  will do...
  - ✧ The final pass ensures array is sorted
- Different sequences can dramatically increase (or decrease) performance
- Code is similar to insertion sort





# SHELL SORT NOTES

Initial: 265 301 751 129 937 863 742 694 076 438

1st step: Step size  $d_1=5$ :

1st loop: 265 301 694 076 438 863 742 751 129 937

2cd step: Step size  $d_2=d_1/2=2$ :

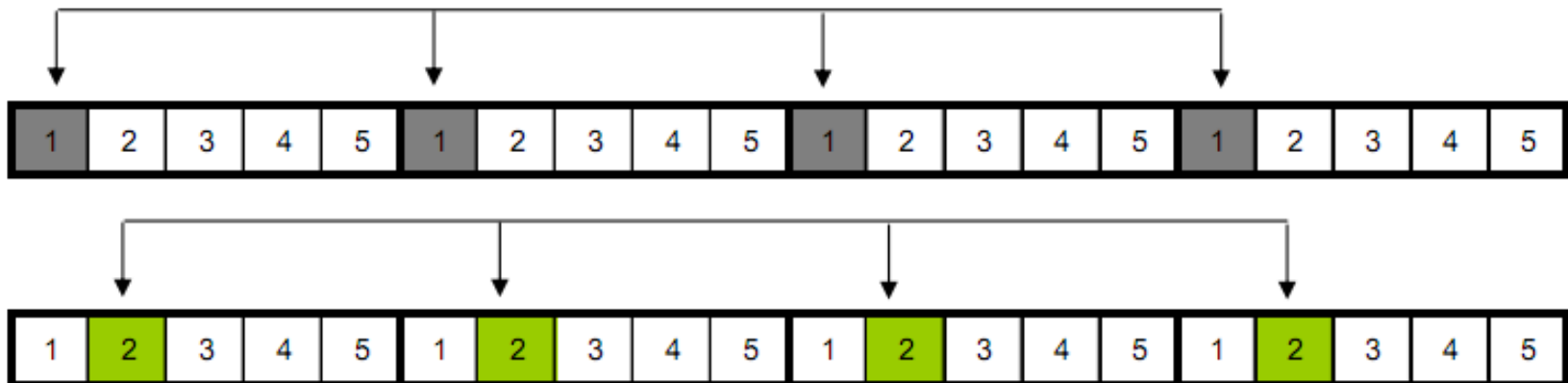
2cd loop: 129 076 265 301 438 751 694 863 742 937

3rd step: Step size:  $d_3=d_2/2=1$ :

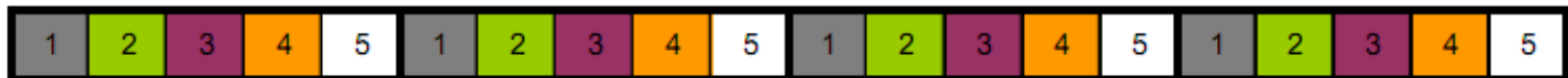
3rd loop: 076 129 265 301 438 694 742 751 863 937

# SUB-ARRAYS WHEN INCREMENT IS 5

5-sorting an array



Elements in each subarray color coded



# C Code: Shellsort

```
void sort(Item a[], int sequence[], int start, int stop)
{
    int step, i;

    for (int step = 0; sequence[step] >= 1; step++)
    {
        int inc = sequence[step];

        for (i = start + inc; i <= stop; i++)
        {
            int j = i;
            Item val = a[i];

            while ((j >= start + inc) && val < a[j - inc])
            {
                a[j] = a[j - inc];
                j -= inc;
            }

            a[j] = val;
        }
    }
}
```



```
#include "stdlib.h"
#include "stdio.h"

#define Item int

void sort(Item a[], int sequence[], int start, int stop);

int main(int argc, char * argv[])
{
    printf("This program uses shell sort to sort a random array\n\n");
    printf("  Parameters: [array-size]\n\n");

    int size = 100;
    if (argc > 1) size = atoi(argv[1]);

    int sequence[] = { 364, 121, 40, 13, 4, 1, 0};
    int * array = (int *) malloc(sizeof(int) * size);

    srand(123456);
    printf("Generating %d random elements ...\n", size);
    for (int i = 0; i < size; i++)
        array[i] = rand();

    printf("Sorting elements ...\n", size);
    sort(array, sequence, 0, size - 1);

    printf("The sorted array is ...\n");
    for (int i = 0; i < size; i++)
        printf("%d ", array[i]);
    printf("\n");
    free(array);
}
```



## Shell 增量序列

Shell 增量序列的递推公式为:  $h_t = \lfloor \frac{N}{2} \rfloor, h_k = \lfloor \frac{h_{k+1}}{2} \rfloor$

## Hibbard 增量序列

Hibbard 增量序列的通项公式为:  $h_i = 2^i - 1$   $h_1 = 1, h_i = 2 * h_{i-1} + 1$

## Knuth 增量序列

Knuth 增量序列的通项公式为:  $h_i = \frac{1}{2}(3^i - 1)$   $h_1 = 1, h_i = 3 * h_{i-1} + 1$

## Gonnet 增量序列

Gonnet 增量序列的递推公式为:  $h_t = \lfloor \frac{N}{2.2} \rfloor, h_k = \lfloor \frac{h_{k+1}}{2.2} \rfloor$  (若  $h_2 = 2$  则  $h_1 = 1$ )

## Sedgewick 增量序列

Sedgewick 增量序列的通项公式为:  $h_i = \max(9 * 4^j - 9 * 2^j + 1, 4^k - 3 * 2^k + 1)$

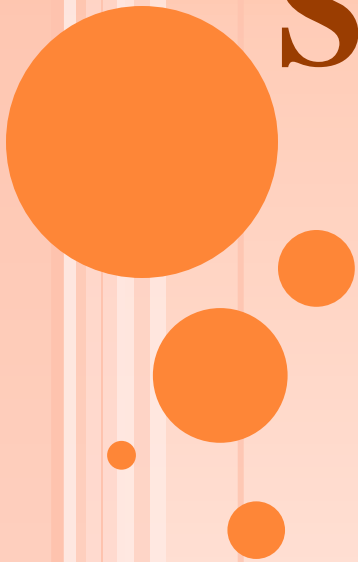


# ANALYSIS OF SHELL SORT

- The shell sort is still significantly slower than the merge, heap, and quick sorts, but its relatively simple algorithm makes it a good choice for sorting lists of less than 5000 items unless speed is important. It's also an excellent choice for repetitive sorting of smaller lists.
- For good increment sequences, requires time proportional to
  - ∞  $O(n (\log n)^2)$  ( $n \rightarrow \infty$ )
  - ∞  $O(n^{1.25})$
- 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.



**HOW FAST CAN WE  
SORT BY NOW**



# HOW FAST CAN WE SORT?

All the sorting algorithms we have seen so far are *comparison sorts*: only use comparisons to determine the relative order of elements.

- E.g., insertion sort, merge sort, quicksort, heapsort.

The best worst-case running time that we've seen for comparison sorting is  $O(n \lg n)$ .

*Q: Is  $O(n \lg n)$  the best we can do?*

*A: Yes, as long as we use comparison sorts*

**TODAY:** Prove any comparison sort has  $\Omega(n \lg n)$  worst case running time





# THE TIME COMPLEXITY OF A PROBLEM

The minimum time needed by an algorithm to solve it.

Upper Bound:

Problem **P** is solvable in time  $T_{\text{upper}}(n)$   
if there is an algorithm **A** which

- outputs the correct answer
- in this much time

$$\exists A, \forall I, A(I)=P(I) \text{ and } \text{Time}(A,I) \leq T_{\text{upper}}(|I|)$$

Lower Bound:

Time  $T_{\text{lower}}(n)$  is a lower bound for problem  
if no algorithm solves the problem faster.

$$\forall A, \exists I, A(I) \neq P(I) \text{ or } \text{Time}(A,I) \geq T_{\text{lower}}(|I|)$$



# TODAY:

PROVED A LOWER BOUND FOR  
*ANY COMPARISON BASED ALGORITHM*  
FOR THE SORTING PROBLEM

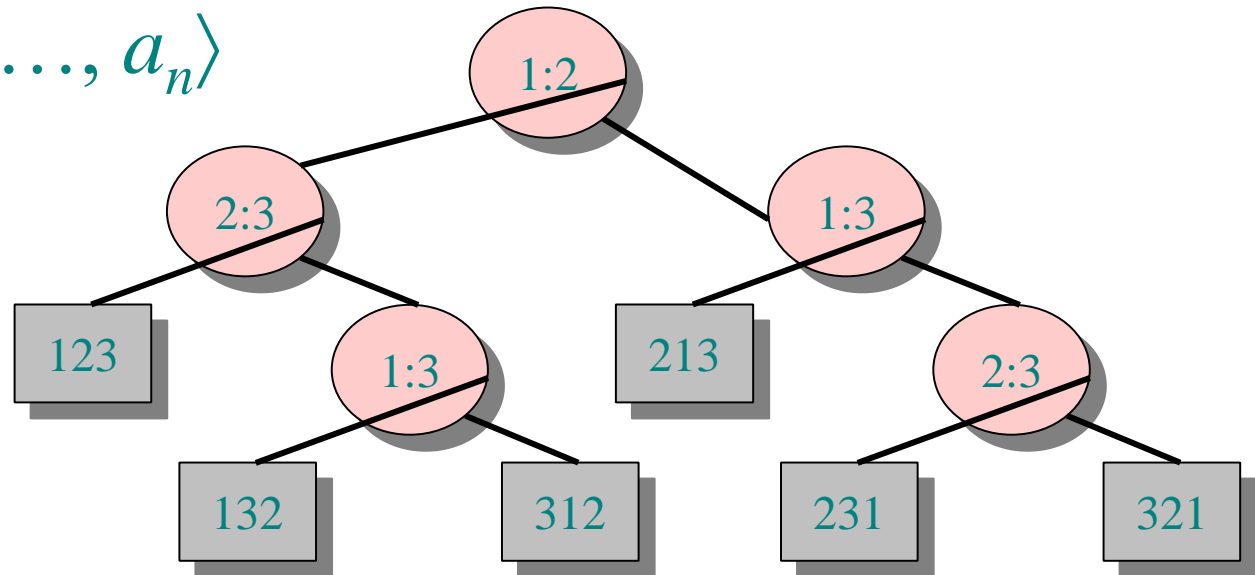
*How?*

*Decision trees* help us.



# DECISION-TREE EXAMPLE

Sort  $\langle a_1, a_2, \dots, a_n \rangle$



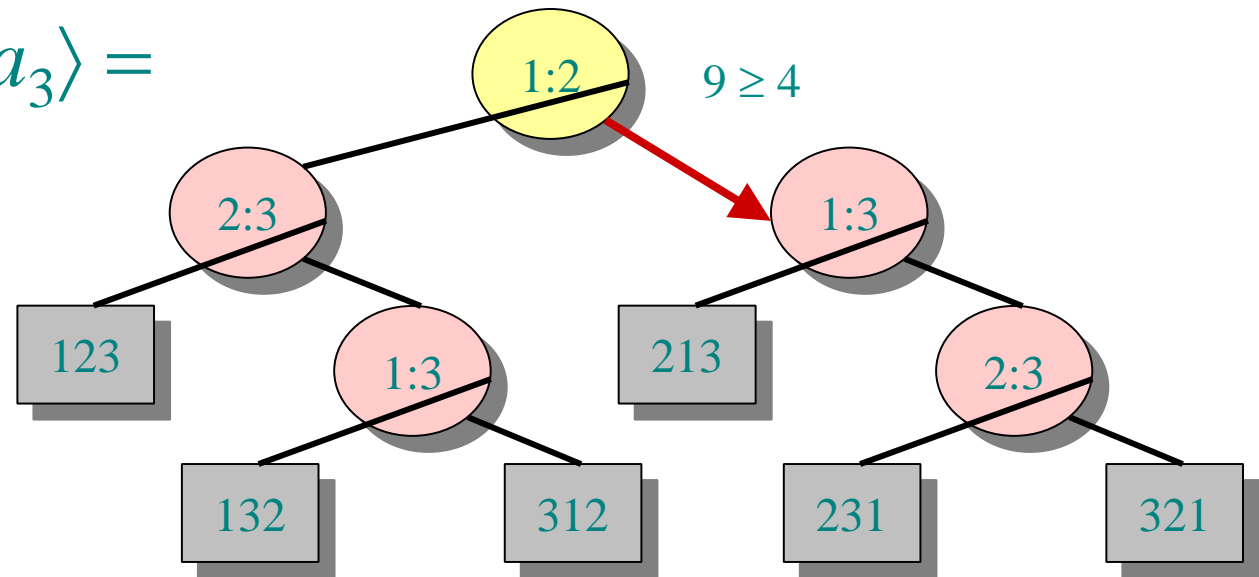
Each internal node is labeled  $i:j$  for  $i, j \in \{1, 2, \dots, n\}$ .

- The left subtree shows subsequent comparisons if  $a_i \leq a_j$ .
- The right subtree shows subsequent comparisons if  $a_i \geq a_j$ .



# DECISION-TREE EXAMPLE

Sort  $\langle a_1, a_2, a_3 \rangle =$   
 $\langle 9, 4, 6 \rangle$ :



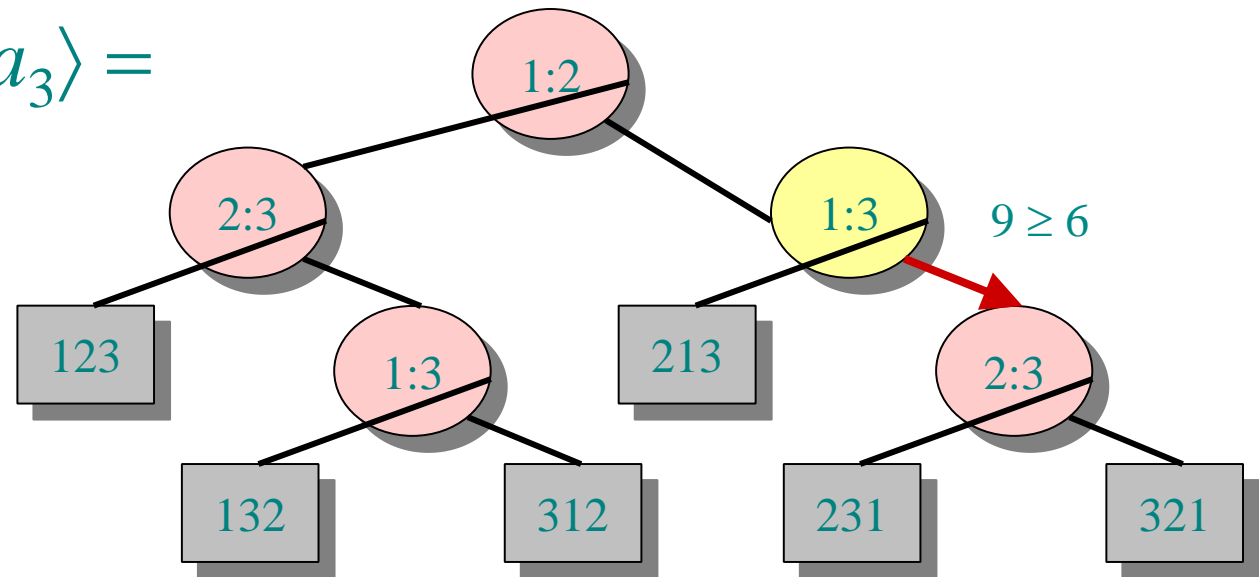
Each internal node is labeled  $i:j$  for  $i, j \in \{1, 2, \dots, n\}$ .

- The left subtree shows subsequent comparisons if  $a_i \leq a_j$ .
- The right subtree shows subsequent comparisons if  $a_i \geq a_j$ .



# DECISION-TREE EXAMPLE

Sort  $\langle a_1, a_2, a_3 \rangle =$   
 $\langle 9, 4, 6 \rangle$ :



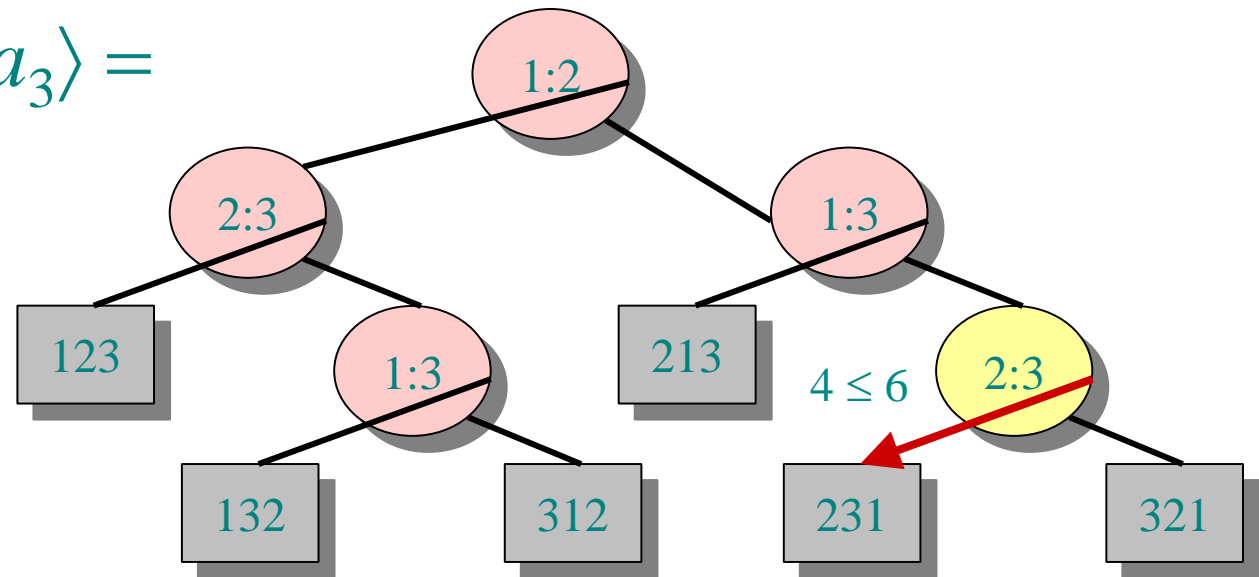
Each internal node is labeled  $i:j$  for  $i, j \in \{1, 2, \dots, n\}$ .

- The left subtree shows subsequent comparisons if  $a_i \leq a_j$ .
- The right subtree shows subsequent comparisons if  $a_i \geq a_j$ .



# DECISION-TREE EXAMPLE

Sort  $\langle a_1, a_2, a_3 \rangle =$   
 $\langle 9, 4, 6 \rangle$ :



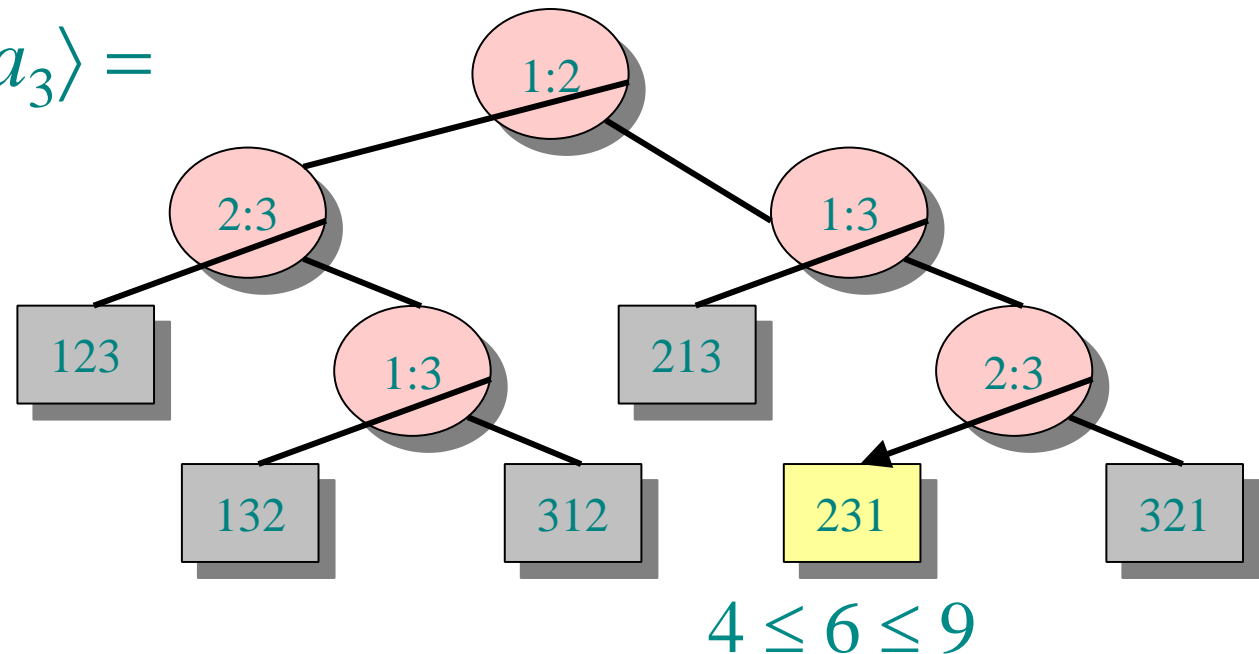
Each internal node is labeled  $i:j$  for  $i, j \in \{1, 2, \dots, n\}$ .

- The left subtree shows subsequent comparisons if  $a_i \leq a_j$ .
- The right subtree shows subsequent comparisons if  $a_i \geq a_j$ .



# DECISION-TREE EXAMPLE

Sort  $\langle a_1, a_2, a_3 \rangle =$   
 $\langle 9, 4, 6 \rangle$ :



Each leaf contains a permutation  $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$  to indicate that the ordering  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$  has been established.

# DECISION-TREE MODEL

*A decision tree can model the execution of any comparison sort:*

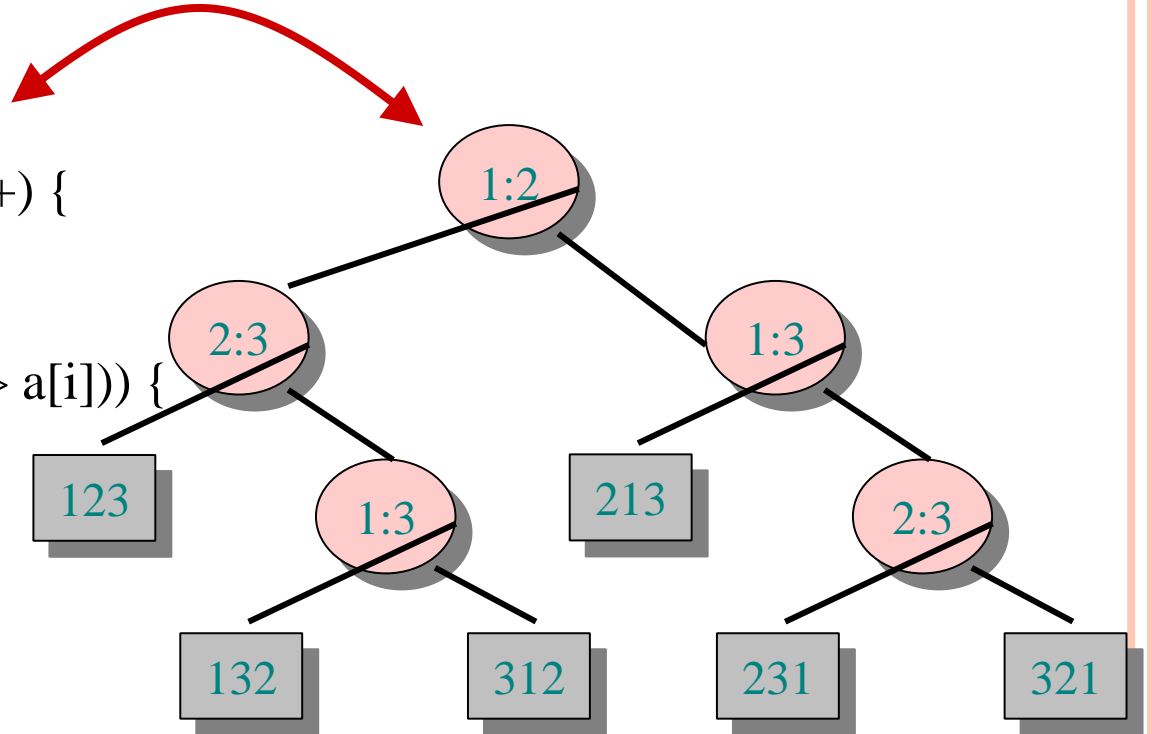
- One tree for each input size  $n$ .
- View the algorithm as **splitting** whenever it compares two elements.
- The tree contains the **comparisons** along all possible instruction traces.
- The running **time** of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.





# Any comparison sort Can be turned into a Decision tree

```
class InsertionSortAlgorithm {  
    for (int i = 1; i < a.length; i++) {  
        int j = i;  
        while ((j > 0) && (a[j-1] > a[i])) {  
            a[j] = a[j-1];  
            j--;  
        }  
        a[j] = B;    }  
}
```



*What do the leaves represent?*  
*How many leaves must there be?*

A permutation  
 $n!$



# LOWER BOUND FOR DECISION-TREE SORTING

**Theorem.** Any decision tree that can sort  $n$  elements must have height  $\Omega(n \lg n)$ .

- *What's the minimum # of leaves?*
- *What's the maximum # of leaves of a binary tree of height  $h$ ?*

Clearly the minimum # of leaves is less than or equal to the maximum # of leaves

**Proof.** The tree must contain  $\geq n!$  leaves, since there are  $n!$  possible permutations. A height- $h$  binary tree has  $\leq 2^h$  leaves. Thus,  $n! \leq 2^h$ .



# LOWER BOUND FOR DECISION-TREE SORTING

- So we have...

$$n! \leq 2^h$$

- Taking logarithms:

$$\lg(n!) \leq h$$

- Stirling's approximation tells us:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \theta\left(\frac{1}{n}\right)\right) > \left(\frac{n}{e}\right)^n$$

- Thus:  $h \geq \lg\left(\frac{n}{e}\right)^n$

$$= n \lg n - n \lg e$$

$$= \Omega(n \lg n)$$



# LOWER BOUND FOR COMPARISON SORTING

**Corollary.** Heapsort and merge sort are asymptotically optimal comparison sorting algorithms. 



# Sorting Lower Bound

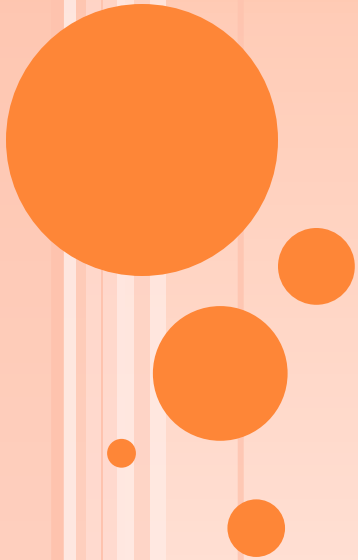
Is there a faster algorithm?

If different models of computation?

```
class InsertionSortAlgorithm {  
    for (int i = 1; i < a.length; i++) {  
        int j = i;  
        while ((j > 0) && (a[j-1] > a[i])) {  
            a[j] = a[j-1];  
            j--; }  
        a[j] = B; } }
```



# **LINEAR TIME SORTING ALGORITHMS**



# SORTING IN LINEAR TIME

**Counting sort:** No comparisons between elements.

- **Input:**  $A[1 \dots n]$ , where  $A[j] \in \{1, 2, \dots, k\}$ .
- **Output:**  $B[1 \dots n]$ , sorted.
- **Auxiliary storage:**  $C[1 \dots k]$ .



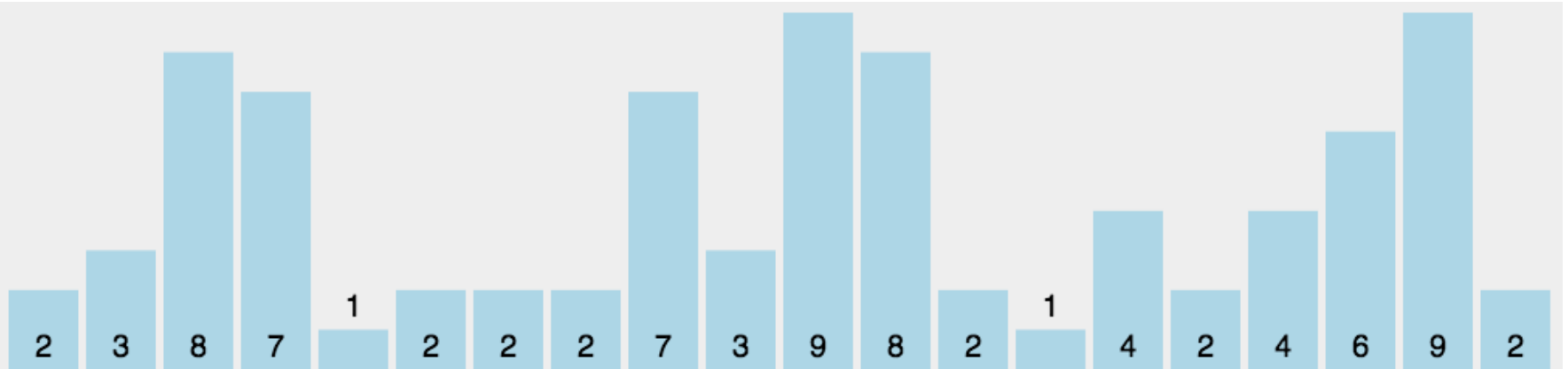
# COUNTING SORT

```
for  $i \leftarrow 1$  to  $k$ 
  do  $C[i] \leftarrow 0$ 
for  $j \leftarrow 1$  to  $n$ 
  do  $C[A[j]] \leftarrow C[A[j]] + 1 \triangleright C[i] = |\{\text{key} = i\}|$ 
for  $i \leftarrow 2$  to  $k$ 
  do  $C[i] \leftarrow C[i] + C[i-1] \triangleright C[i] = |\{\text{key} \leq i\}|$ 
for  $j \leftarrow n$  down to 1
  do  $B[C[A[j]]] \leftarrow A[j]$ 
    $C[A[j]] \leftarrow C[A[j]] - 1$ 
```

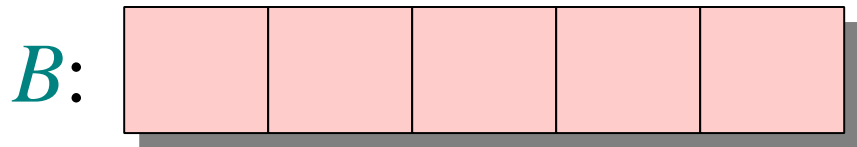
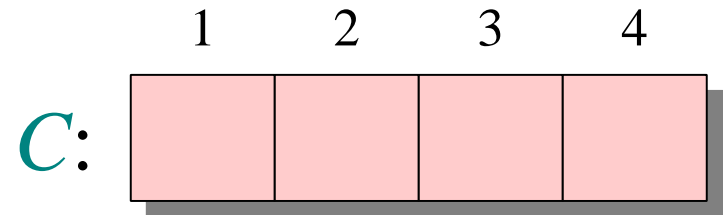
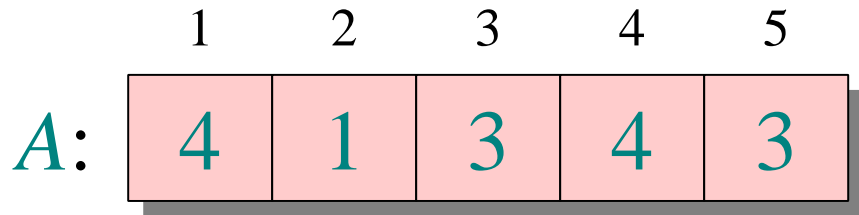




# COUNTING SORT



# COUNTING-SORT EXAMPLE



# LOOP 1

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

	1	2	3	4
<i>C</i> :	0	0	0	0

<i>B</i> :					
------------	--	--	--	--	--

**for**  $i \leftarrow 1$  **to**  $k$   
    **do**  $C[i] \leftarrow 0$



# LOOP 2

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

	1	2	3	4
<i>C</i> :	0	0	0	1

<i>B</i> :					
------------	--	--	--	--	--

**for**  $j \leftarrow 1$  **to**  $n$

**do**  $C[A[j]] \leftarrow C[A[j]] + 1$   $\triangleright C[i] = |\{\text{key} = i\}|$



# LOOP 2

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

	1	2	3	4
<i>C</i> :	1	0	0	1

<i>B</i> :					
------------	--	--	--	--	--

**for**  $j \leftarrow 1$  **to**  $n$

**do**  $C[A[j]] \leftarrow C[A[j]] + 1$   $\triangleright C[i] = |\{\text{key} = i\}|$



# LOOP 2

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

	1	2	3	4
<i>C</i> :	1	0	1	1

<i>B</i> :					
------------	--	--	--	--	--

**for**  $j \leftarrow 1$  **to**  $n$

**do**  $C[A[j]] \leftarrow C[A[j]] + 1$   $\triangleright C[i] = |\{\text{key} = i\}|$



# LOOP 2

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

	1	2	3	4
<i>C</i> :	1	0	1	2

<i>B</i> :					
------------	--	--	--	--	--

**for**  $j \leftarrow 1$  **to**  $n$

**do**  $C[A[j]] \leftarrow C[A[j]] + 1$   $\triangleright C[i] = |\{\text{key} = i\}|$



# LOOP 2

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

	1	2	3	4
<i>C</i> :	1	0	2	2

<i>B</i> :					
------------	--	--	--	--	--

**for**  $j \leftarrow 1$  **to**  $n$

**do**  $C[A[j]] \leftarrow C[A[j]] + 1$   $\triangleright C[i] = |\{\text{key} = i\}|$





# LOOP 3

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

	1	2	3	4
<i>C</i> :	1	0	2	2

<i>B</i> :					
------------	--	--	--	--	--

<i>C'</i> :	1	1	2	2
-------------	---	---	---	---

**for**  $i \leftarrow 2$  **to**  $k$

**do**  $C[i] \leftarrow C[i] + C[i-1]$   $\triangleright C[i] = |\{\text{key} \leq i\}|$



# LOOP 3

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

	1	2	3	4
<i>C</i> :	1	0	2	2

<i>B</i> :					
------------	--	--	--	--	--

<i>C'</i> :	1	1	3	2
-------------	---	---	---	---

**for**  $i \leftarrow 2$  **to**  $k$

**do**  $C[i] \leftarrow C[i] + C[i-1]$   $\triangleright C[i] = |\{\text{key} \leq i\}|$



# LOOP 3

	1	2	3	4	5
<i>A</i> :	4	1	3	4	3

	1	2	3	4
<i>C</i> :	1	0	2	2

<i>B</i> :					
------------	--	--	--	--	--

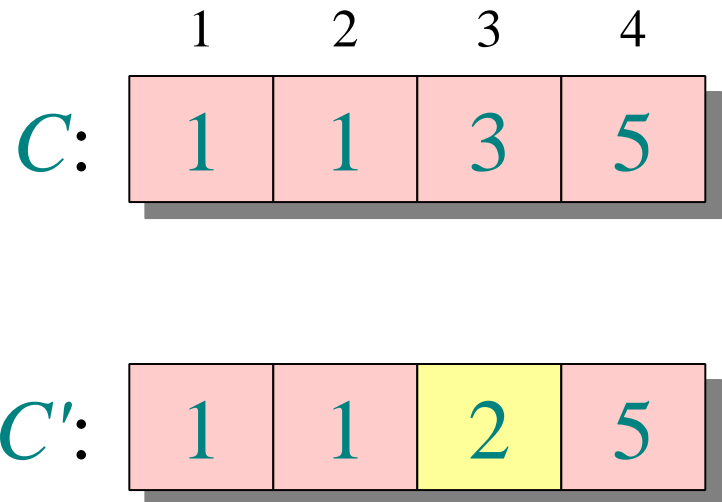
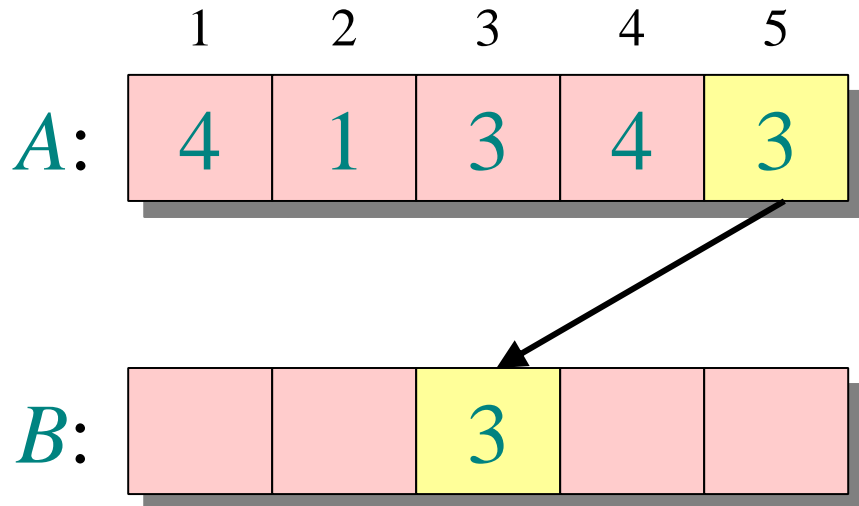
<i>C'</i> :	1	1	3	5
-------------	---	---	---	---

**for**  $i \leftarrow 2$  **to**  $k$

**do**  $C[i] \leftarrow C[i] + C[i-1]$   $\triangleright C[i] = |\{\text{key} \leq i\}|$



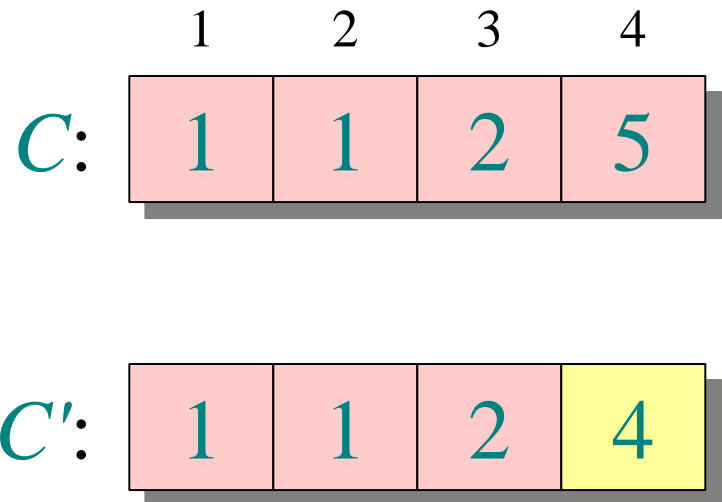
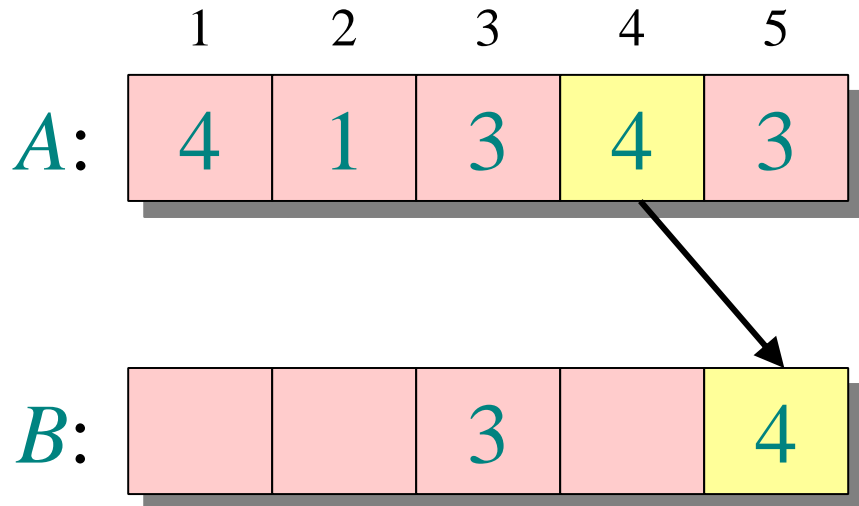
# LOOP 4



```
for  $j \leftarrow n$  down to 1  
  do  $B[C[A[j]]] \leftarrow A[j]$   
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



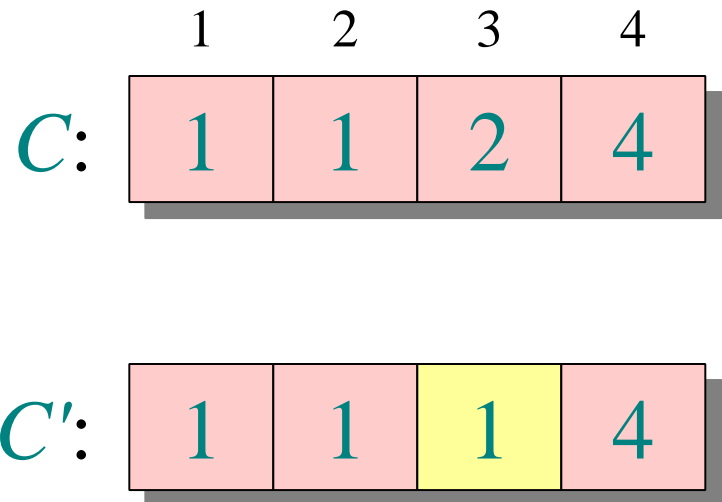
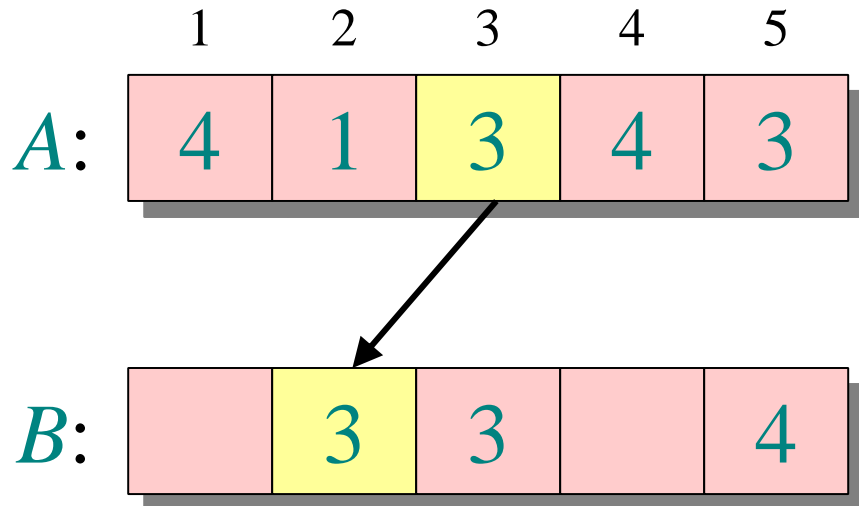
# LOOP 4



```
for  $j \leftarrow n$  down to 1  
  do  $B[C[A[j]]] \leftarrow A[j]$   
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



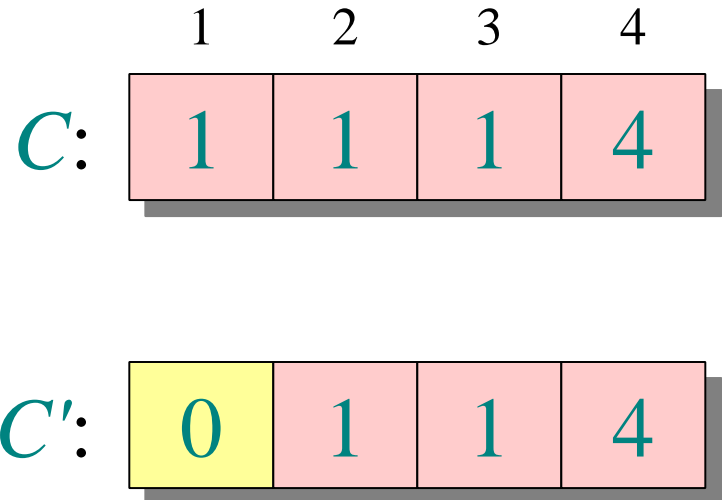
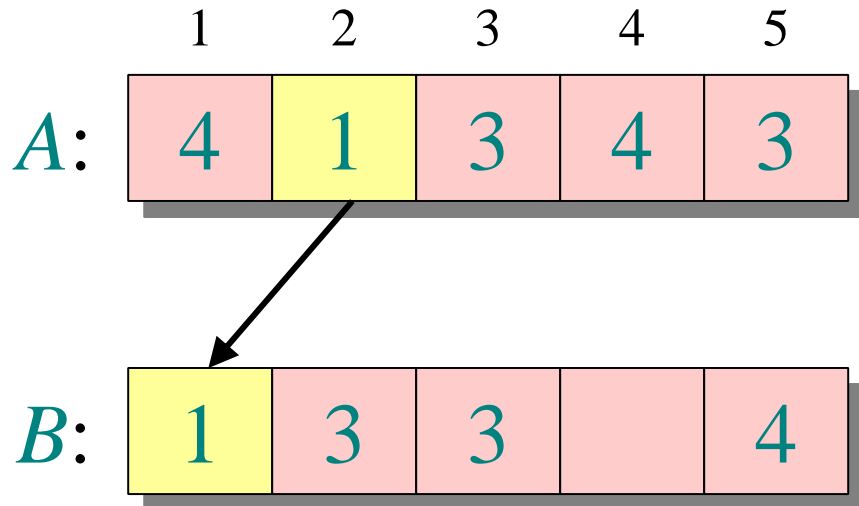
# LOOP 4



```
for  $j \leftarrow n$  down to 1  
  do  $B[C[A[j]]] \leftarrow A[j]$   
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



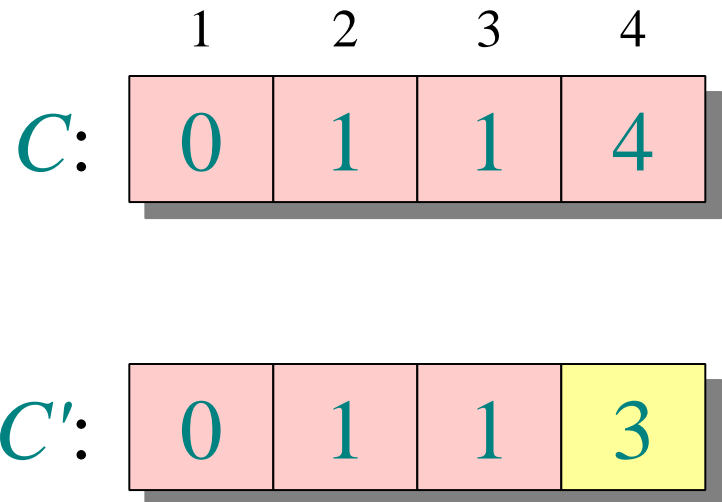
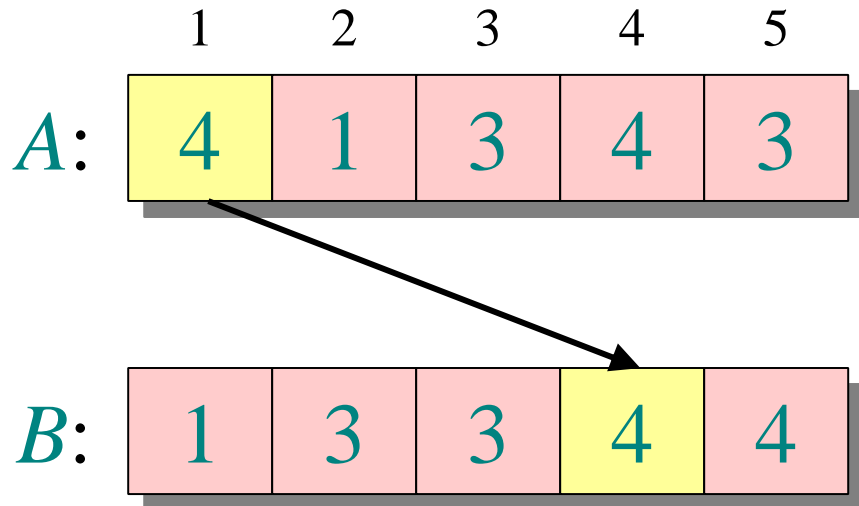
# LOOP 4



```
for  $j \leftarrow n$  down to 1  
  do  $B[C[A[j]]] \leftarrow A[j]$   
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



# LOOP 4



```
for  $j \leftarrow n$  down to 1  
  do  $B[C[A[j]]] \leftarrow A[j]$   
      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```





# ANALYSIS

$\Theta(k)$  { **for**  $i \leftarrow 1$  **to**  $k$       Very large  $k$ ?  
          **do**  $C[i] \leftarrow 0$

$\Theta(n)$  { **for**  $j \leftarrow 1$  **to**  $n$   
          **do**  $C[A[j]] \leftarrow C[A[j]] + 1$       Counting Mat

$\Theta(k)$  { **for**  $i \leftarrow 2$  **to**  $k$   
          **do**  $C[i] \leftarrow C[i] + C[i-1]$       Accumulation Mat

$\Theta(n)$  { **for**  $j \leftarrow n$  **down to**  $1$   
          **do**  $B[C[A[j]]] \leftarrow A[j]$   
               $C[A[j]] \leftarrow C[A[j]] - 1$

---

$\Theta(n + k)$



# RUNNING TIME

If  $k = O(n)$ , then counting sort takes  $\Theta(n)$  time.

- But, sorting takes  $\Omega(n \lg n)$  time!
- Where's the fallacy?

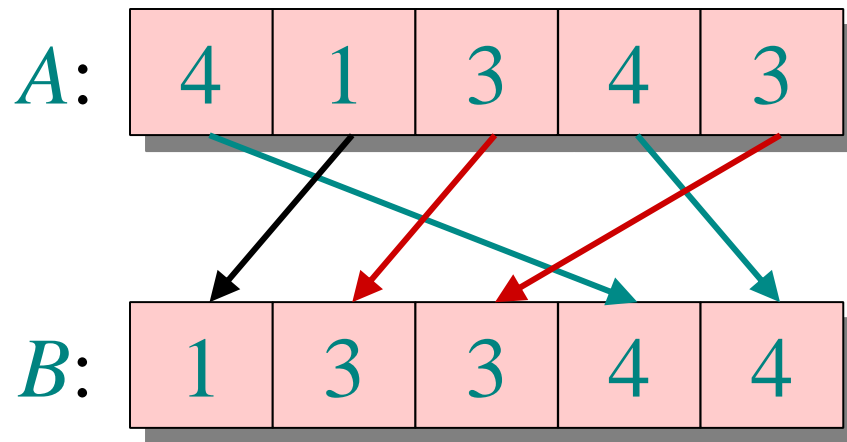
## Answer:

- *Comparison sorting* takes  $\Omega(n \lg n)$  time.
- Counting sort is not a *comparison sort*.
- In fact, not a single comparison between elements occurs!



# STABLE SORTING

Counting sort is a *stable* sort: it preserves the input order among equal elements.



**Exercise:** What other sorts have this property?

Bubble sort, insertion sort, BST, Merge sort, etc.



# COUNTING SORT

- *Why don't we always use counting sort?*
  - Because it depends on range  $k$  of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
  - Answer: no,  $k$  too large ( $2^{32} = 4,294,967,296$ )
  - Map the range of index to  $0 \sim n$  (not really helpful)



# IMPROVEMENT BY RADIX SORT

- In fact, each number is composed of digits.
  - The range of each digit is limited.
  - We can run counting sort on each digit.

8 3 9

4 3 6

7 2 0

3 5 5

3 2 9

4 5 7

6 5 7

8 3 9

4 3 6

7 2 0

3 5 5

3 2 9

4 5 7

6 5 7

8 3 9

4 3 6

7 2 0

3 5 5

3 2 9

4 5 7

6 5 7

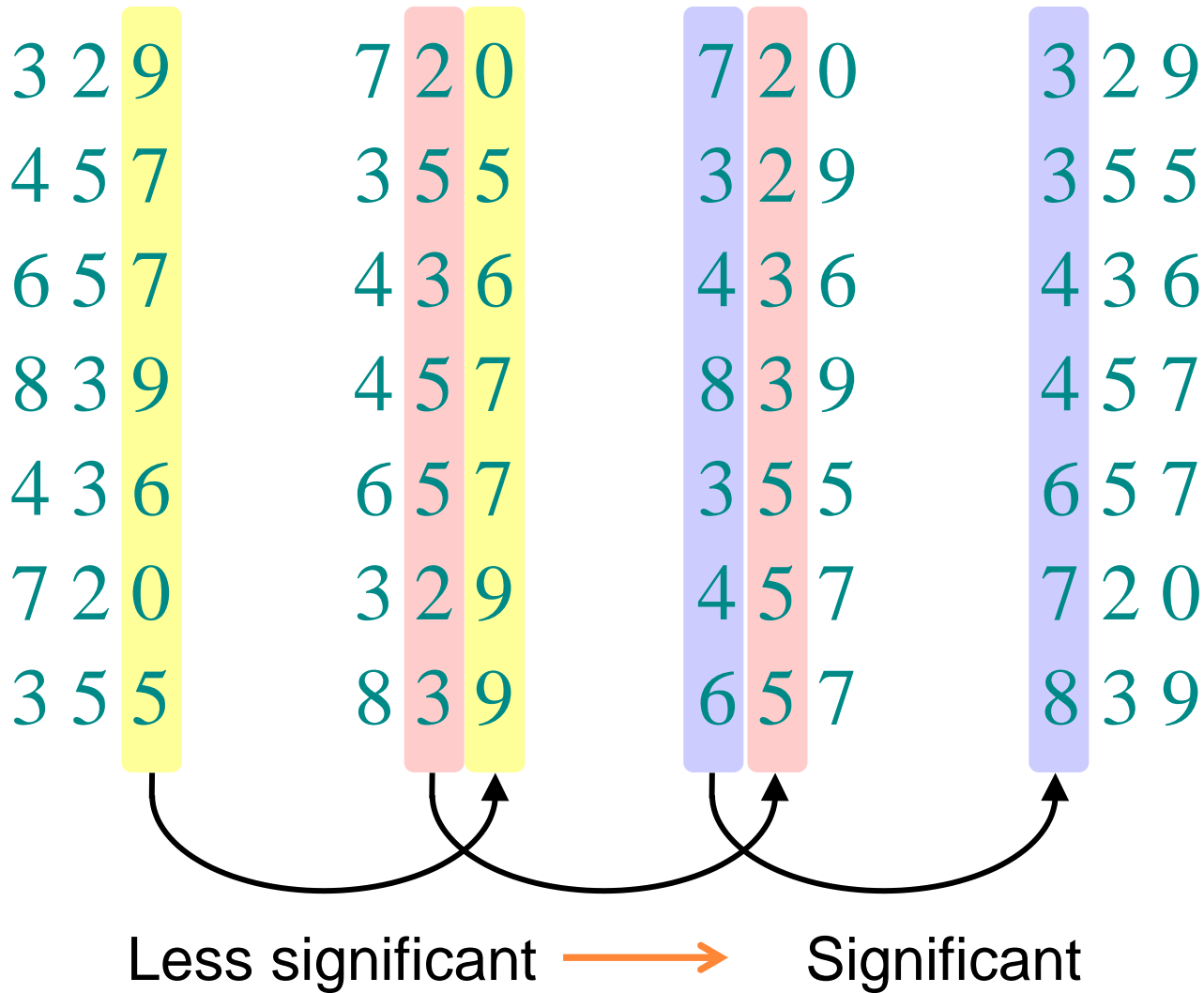


# RADIX SORT

- *Origin*: Herman Hollerith's card-sorting machine for the 1890 U.S. Census.
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on *least-significant digit first* with auxiliary *stable* sort.



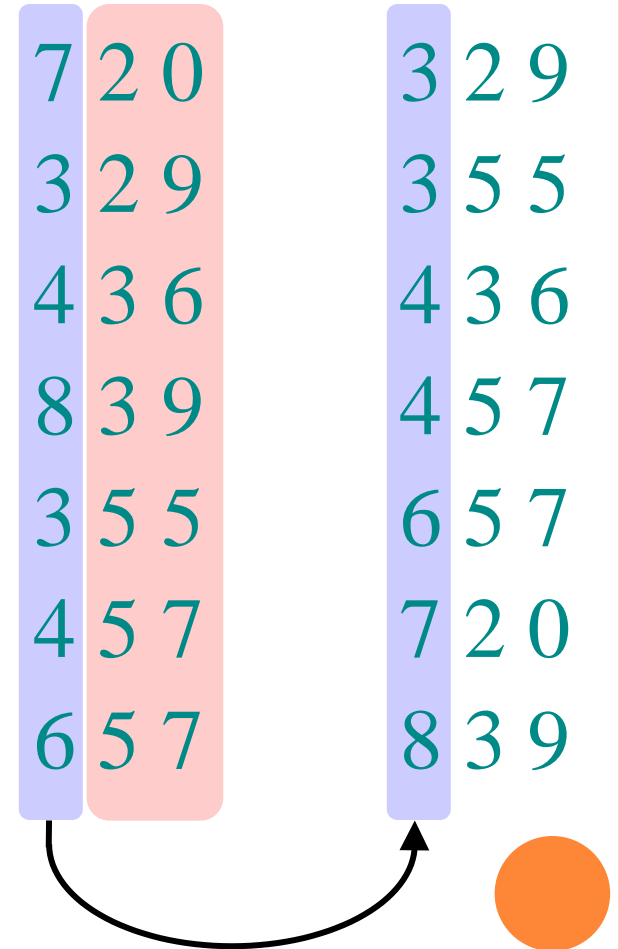
# OPERATION OF RADIX SORT



# CORRECTNESS OF RADIX SORT

*Induction on digit position*

- Assume that the numbers are sorted by their low-order  $t - 1$  digits.
- Sort on digit  $t$

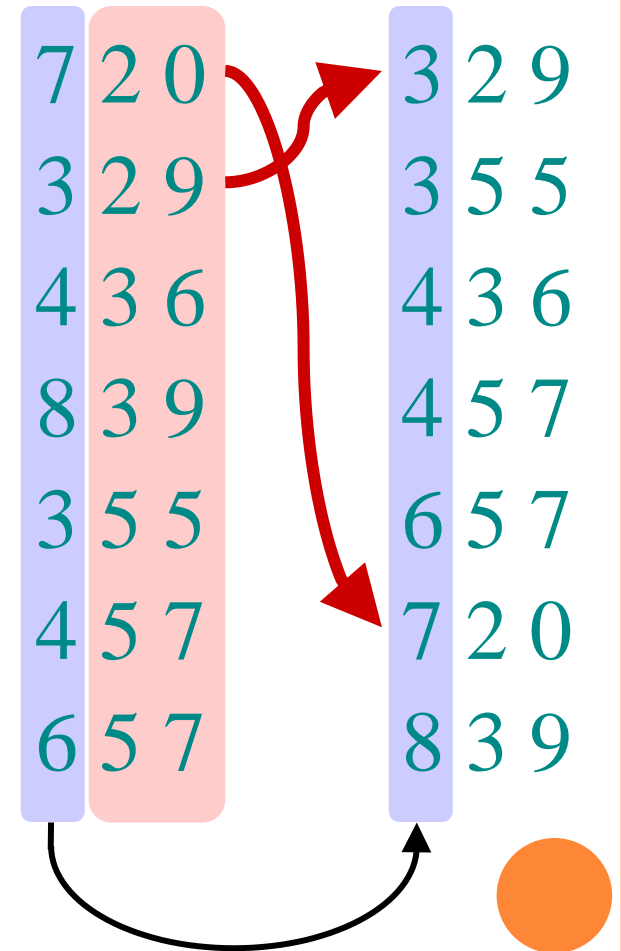




# CORRECTNESS OF RADIX SORT

*Induction on digit position*

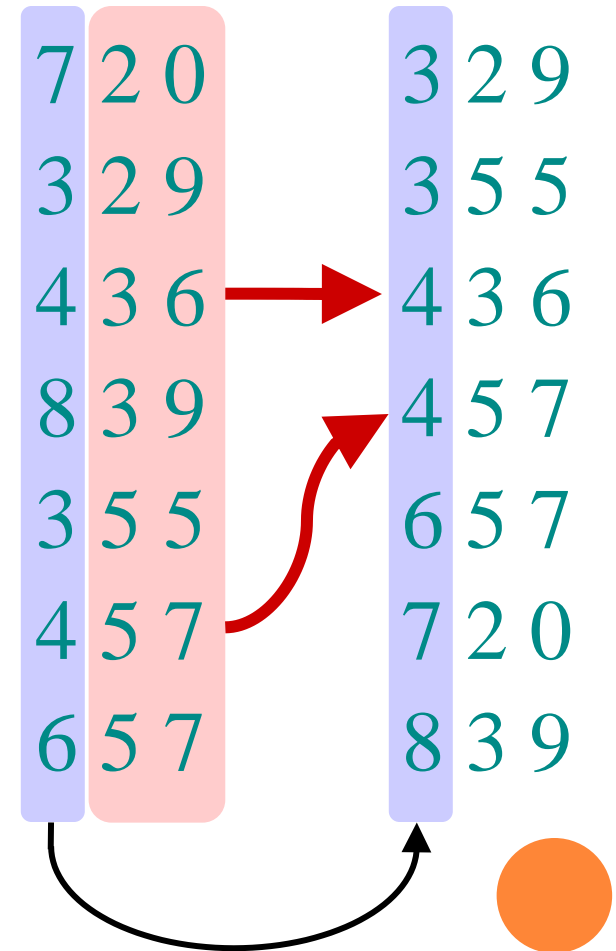
- Assume that the numbers are sorted by their low-order  $t - 1$  digits.
- Sort on digit  $t$ 
  - Two numbers that differ in digit  $t$  are correctly sorted.



# CORRECTNESS OF RADIX SORT

*Induction on digit position*

- Assume that the numbers are sorted by their low-order  $t - 1$  digits.
- Sort on digit  $t$ 
  - Two numbers that differ in digit  $t$  are correctly sorted.
  - Two numbers equal in digit  $t$  are put in the same order as the input  $\Rightarrow$  correct order.



# ANALYSIS OF RADIX SORT

- Assume counting sort is the **auxiliary stable sort**.
- Sort  $n$  computer words of  $b$  bits each.
- Each word can be viewed as having  $b/r$  base- $2^r$  digits.

**Example:** 32-bit word 

$r = 8 \Rightarrow b/r = 4$  passes of counting sort on base- $2^8$  digits; or  $r = 16 \Rightarrow b/r = 2$  passes of counting sort on base- $2^{16}$  digits.

*How many passes should we make?*



# ANALYSIS (CONTINUED)

**Recall:** Counting sort takes  $\Theta(n + k)$  time to sort  $n$  numbers in the range from 0 to  $k - 1$ .

If each  $b$ -bit word is broken into  $r$ -bit pieces, each pass of counting sort takes  $\Theta(n + 2^r)$  time. Since there are  $b/r$  passes, we have

$$T(n, b) = \Theta\left(\frac{b}{r}(n + 2^r)\right).$$

Choose  $r$  to minimize  $T(n, b)$ :

- Increasing  $r$  means fewer passes, but as  $r \gg \lg n$ , the time grows exponentially.



# ANALYSIS (CONTINUED)

Minimize  $T(n, b)$  by differentiating and setting to 0.

Or, just observe that we don't want  $2^r \gg n$ , and there's no harm asymptotically in choosing  $r$  as large as possible subject to this constraint.

Choosing  $r = \lg n$  implies  $T(n, b) = \Theta(bn/\lg n)$ .

- For numbers in the range from 0 to  $n^d - 1$ , we have  $b = d \lg n \Rightarrow$  radix sort runs in  $\Theta(dn)$  time.




# CONCLUSIONS

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

**Example** (32-bit numbers):

- At most 3 passes when sorting  $\geq 2000$  numbers.
- Merge sort and quick sort do at least  $\lceil \lg 2000 \rceil = 11$  passes.

**Downside:** Can't sort in place using counting sort. Also, Unlike quick sort, radix sort displays little locality of reference, and thus a well-tuned quick sort fares better sometimes on modern processors, with steep memory hierarchies.



# ORIGIN OF RADIX SORT

Hollerith's original 1889 patent alludes to a most-significant-digit-first radix sort:

*“The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then resorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards.”*

Least-significant-digit-first radix sort seems to be a folk invention originated by machine operators.



# BUCKET SORT

- In Bucket sort, the range  $[a,b]$  of input numbers is divided into  $m$  equal sized intervals, called **buckets**.
- Each element is placed in its appropriate bucket.
- If the numbers are **uniformly** divided in the range, the buckets can be expected to have roughly identical number of elements.
- Elements in the buckets are locally sorted.
- The expected time of this algorithm is  $\Theta(n)$ .





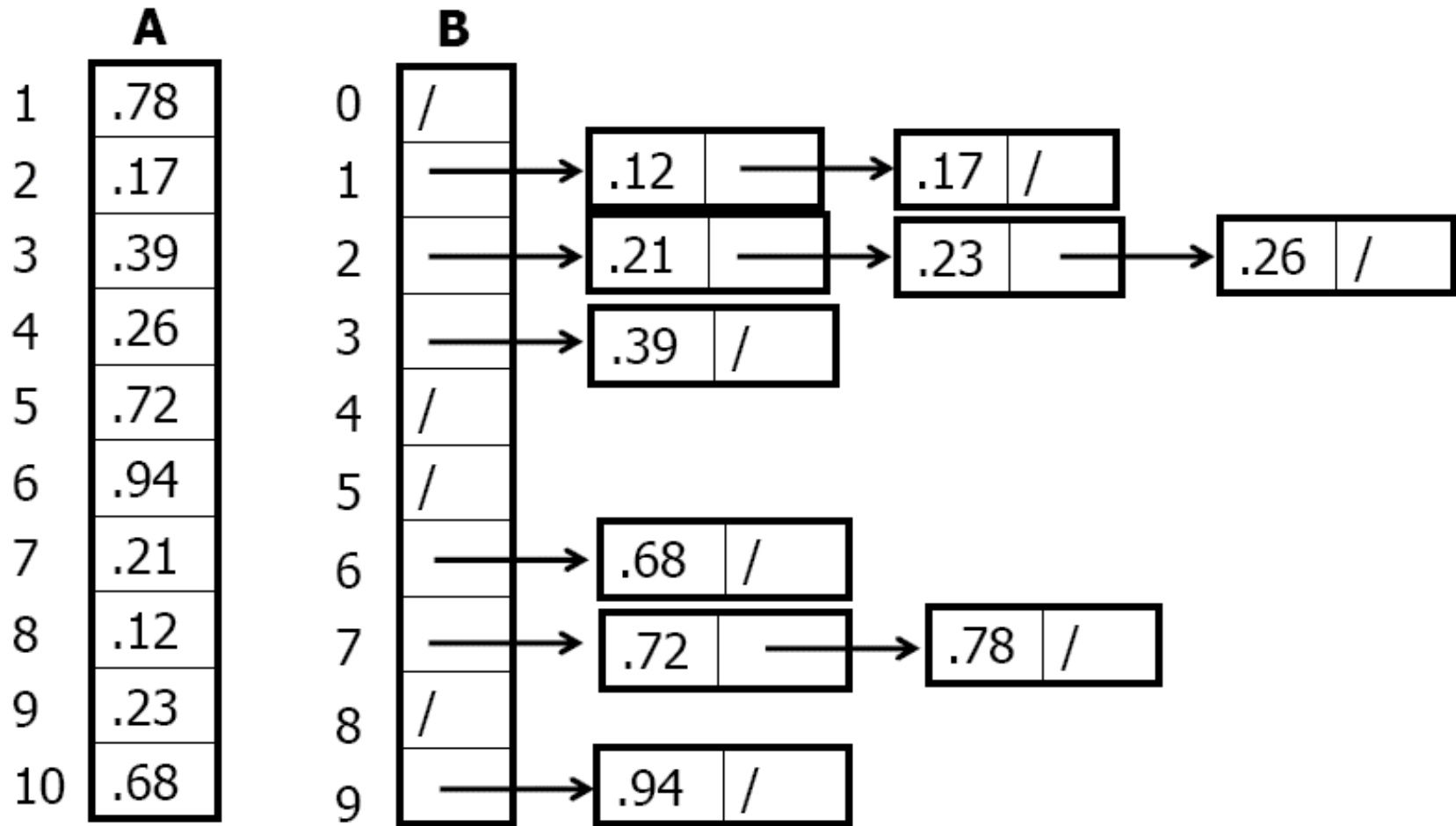
# BUCKET SORT

## Bucket Sort Algorithm ( A )

1.  $n \leftarrow \text{length}( [A] )$
2. for  $i \leftarrow 1$  to  $n$   
do insert  $A[i]$  into list  $B[ \lfloor n A[i] \rfloor ]$
3. for  $i \leftarrow 0$  to  $n-1$   
do sort  $B[i]$  with Insertion Sort
4. Concatenate the lists  $B[0], B[1], \dots, B[n]$   
together in order



# BUCKET SORT



# ANALYSIS OF BUCKET SORT

- All operations except for insertion sort take  $O(n)$  time in the worst case.
- Let  $n_i$  be the random variable denoting the number of elements placed in bucket  $B[i]$ .
- Since the running time of insertion sort is  $O(n^2)$ , the running time of bucket sort is:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$



# ANALYSIS OF BUCKET SORT

- Taking expectations of both sides and using linearity of expectation, we have:

$$\begin{aligned} E(T(n)) &= E \left[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \end{aligned}$$



# ANALYSIS OF BUCKET SORT

- Define  $X_{ij} = I\{A[j] \text{ falls in bucket } i\}$  means the event that  $A[j]$  falls in bucket  $i$ , then we have

$$n_i = \sum_{j=1}^n X_{ij}$$

- By expand the square and regroup terms:

$$\begin{aligned} E[n_i^2] &= E \left[ \left( \sum_{j=1}^n X_{ij} \right)^2 \right] = E \left[ \sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik} \right] \\ &= E \left[ \sum_{j=1}^n X_{ij}^2 + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n X_{ij} X_{ik} \right] \\ &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n E[X_{ij} X_{ik}] \end{aligned}$$



# ANALYSIS OF BUCKET SORT

- Now, we know

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$

$$E[n_i^2] = \sum_{j=1}^n E[X_{ij}^2] + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n E[X_{ij} X_{ik}]$$

- Indicator random variable  $X_{ij}$  is 1 with probability  $1/n$  and 0 otherwise, so we have:

$$E[X_{ij}^2] = 1^2 * \frac{1}{n} + 0^2 * (1 - \frac{1}{n}) = \frac{1}{n}$$



# ANALYSIS OF BUCKET SORT

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$

$$E[n_i^2] = \sum_{j=1}^n E[X_{ij}^2] + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n E[X_{ij} X_{ik}]$$

- When  $k \neq j$ , the variables  $X_{ij}$  and  $X_{ik}$  are independent, and hence, we have

$$E[X_{ij} X_{ik}] = E[X_{ij}] E[X_{ik}] = \frac{1}{n} * \frac{1}{n} = \frac{1}{n^2}$$



# ANALYSIS OF BUCKET SORT

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$

$$E[n_i^2] = \sum_{j=1}^n E[X_{ij}^2] + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n E[X_{ij}X_{ik}]$$

- By substituting these two expected values into above equations, we have

$$E[n_i^2] = n * \frac{1}{n} + n(n-1) * \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n}$$

$$E[T(n)] = \Theta(n) + n * (2 - \frac{1}{n}) = \Theta(n)$$

- Thus, the entire bucket sort algorithm runs **in linear expected time**.