COMP 550
Algorithm and Analysis

Sorting Algorithms

Based on CLRS Secs. 6-8

Some slides are adapted from ones by prior instructors Prof. Plaisted and Prof. Osborne

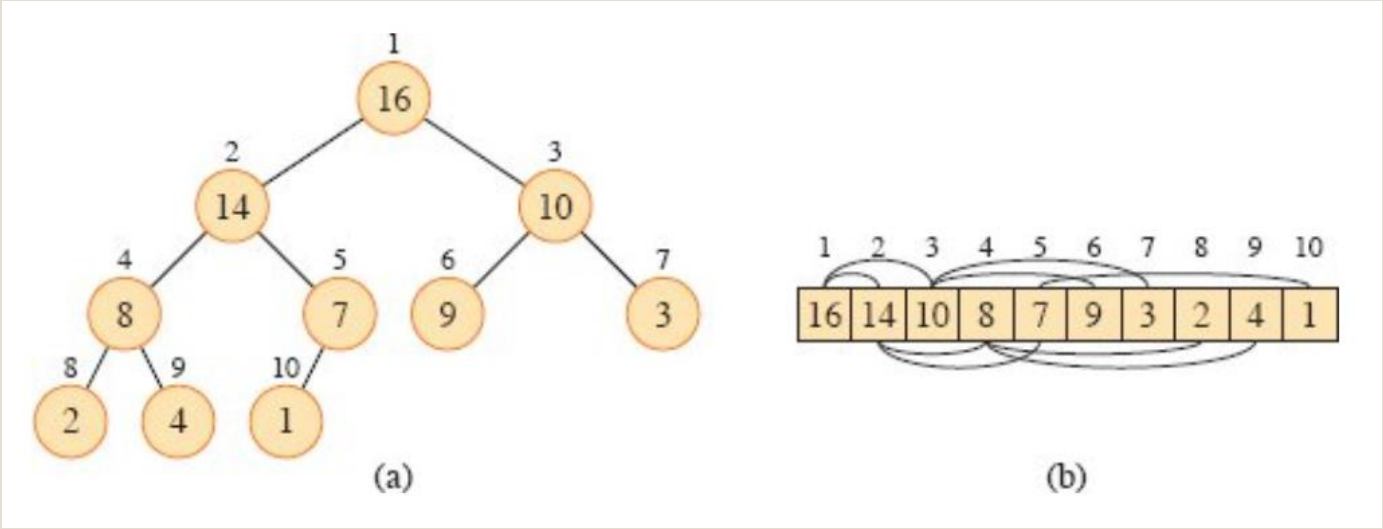
Sorting Algorithms

- We've already studied

Algorithm	Running Time
Insertion Sort	$O(n^2)$
Selection Sort	$\Theta(n^2)$
Merge Sort	$\Theta(n \lg n)$

Heap Sort

- Binary Heap: Can either be MAX-heap or MIN-heap
  - Complete binary tree (all levels except the last one are full)
  - Each node has a key



From CLRS

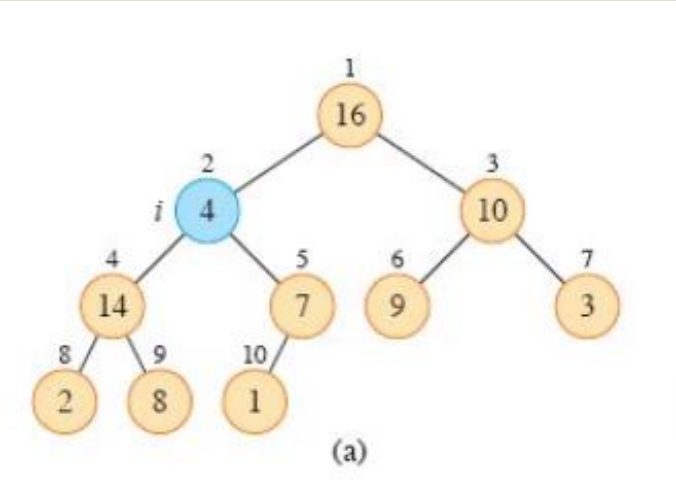
  - Satisfies heap property:
    - A node's key is larger than its children's key (MAX-heap)

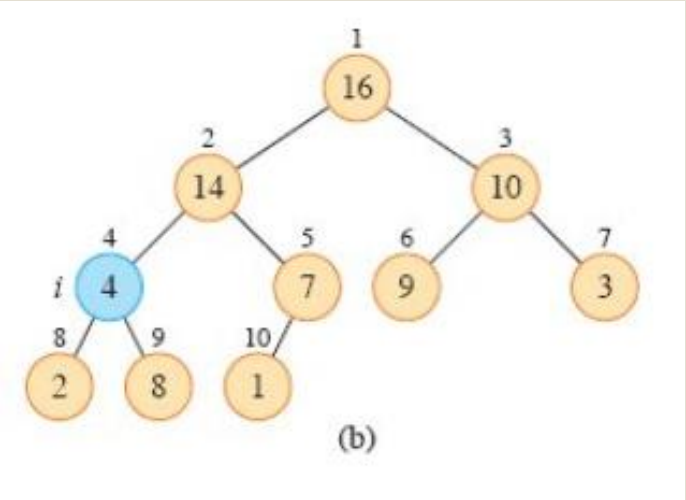
Heap Sort

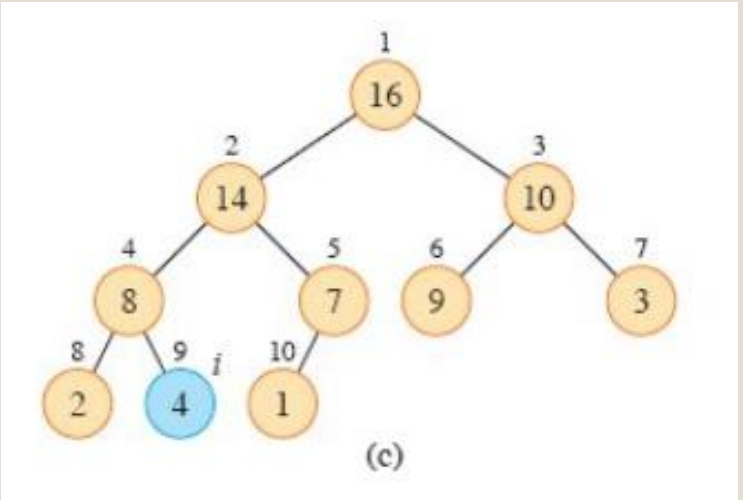
- MAX-Heap Operations:

Operation	Purpose	Running Time
MAX-heapify(A, i)	Percolates down A[i] that potentially violates MAX-heap property	$O(\lg n)$
Build-MAX-Heap(A, n)	Converts A[1:n] to a MAX-Heap	$O(n)$

Illustration of MAX-heapify(A, 2) from CLRS







Heap Sort

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

- Heap Sort running time = $\Theta(n \lg n)$

- In exams, you'll NOT be asked about how Build-Max-Heap and Max-Heapify work.
  - However, you should know their time complexity

Priority Queue

- Max-Priority Queue or Min-Priority Queue
  - Based on MAX- or Min-Heap
  - Enables dequeuing of MAX or MIN element in $O(1)$ time

- Has very useful applications:
  - Ordering processes by their priorities in operating systems
  - Ordering events in event-driven simulation

Priority Queue

- MAX-Priority queue operations:

Operation	Purpose	Running Time
Insert	Insert a value	$O(\lg n)$
Maximum	Returns the maximum element w/o deleting it	$O(1)$
Extract-Max	Dequeue the maximum element	$O(\lg n)$
Increase-Key	Increase key of an element	$O(\lg n)$

- We'll be needing priority queues when solving problems of future topics
- You need to know what operations are supported and what are their running times
- For exams, you don't need to know how these operations actually work.

Quick Sort

- A very well-known sorting algorithm
- We'll not cover it in detail now (will do later if we get time)
- Basic idea: Divide & Conquer

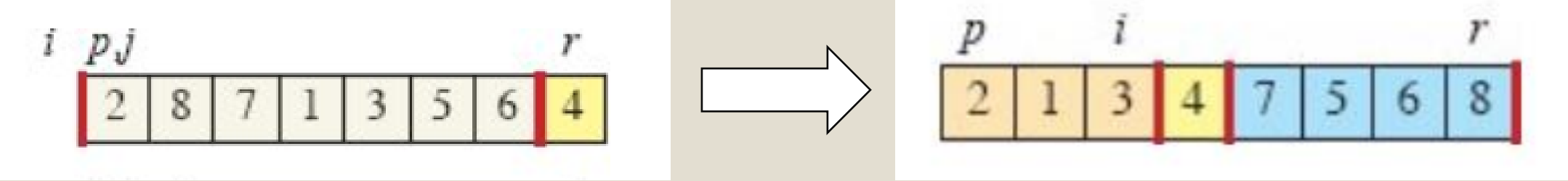
Quick Sort

Divide step:

Take an element k of the array as pivot.

Place k in its correct position in the sorted array

Place all elements smaller than k in its left and large elements in right



Conquer step: Recursively sort left and right side of k

Combine step: Nothing to do!

Quick Sort

Running time: Worst-case $\Theta(n^2)$, Average-case $\Theta(n \lg n)$

Very efficient on average: small constant factor hidden in $\Theta(n \lg n)$

In-place sorting can be done by Quick Sort

- No extra O(n) memory is required for sorting (No additional array needed)
- Which sorting algorithm needs extra array?

Announcement

Midterm 2: November 13, 2023 (in class)

Final Exam: Dec. 14th, Thursday, 4:00 PM

- We have three options for final exam
  - Dec. 14th, Thursday, 4:00 PM (M/W/F 3:35 PM)
  - Dec. 15th, Friday, 8:00 AM (M/W/F 3:35 PM)
  - Dec. 14th, Friday, 4:00 PM (Classes not otherwise…)

Sorting Algorithms

Algorithm	Running Time
Insertion Sort	$O(n^2)$
Selection Sort	$\Theta(n^2)$
Merge Sort	$\Theta(n \lg n)$
Heap Sort	$O(n \lg n)$
Quick Sort	$O(n^2)$

Seems like the worst-case running time is stuck at $\Theta(n \lg n)$

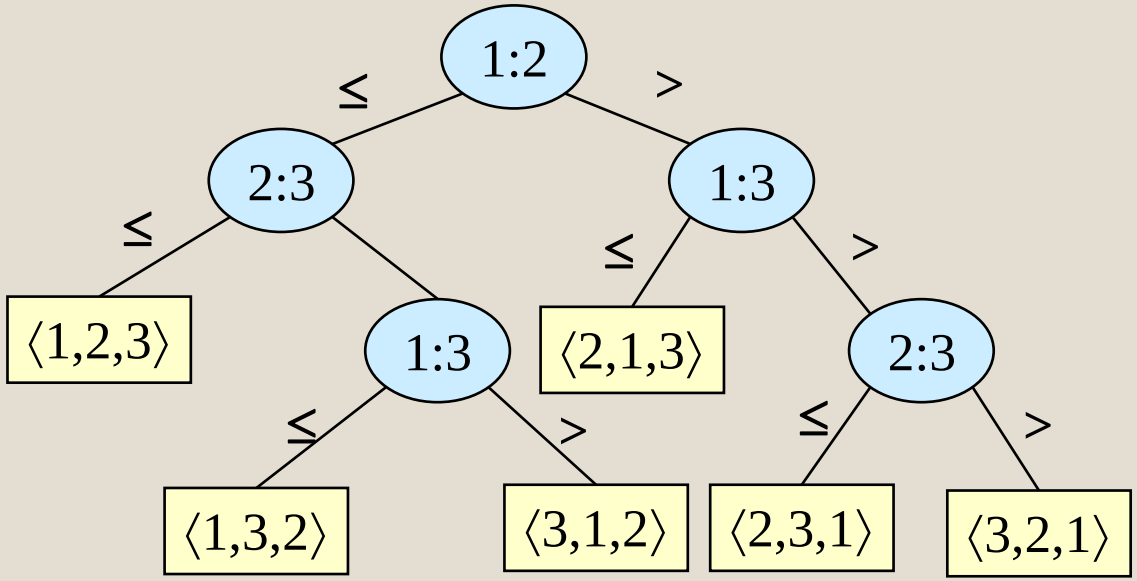
Can we do better?

Sorting Algorithms: Lower Bound

- *Comparison sorts* cannot run faster than $\Theta(n \lg n)$
  - In other words, comparison sorts require $\Omega(n \lg n)$ time
- Comparison sort: a sort that orders elements based on comparing elements.
- We can say A is greater than B only if
  - We've compared A to B or
  - We've compared A to something known to be greater than B.

Decision Trees

- Numbers are indices not elements.
- Leaves indicate the sorted order of the original elements.
- Go left iff the first (left) element is smaller.
- Path length indicates the number of comparisons needed to reach a solution.
  - Tree height = worst-case running time



Decision Tree for sorting a 3-item array

Decision Trees

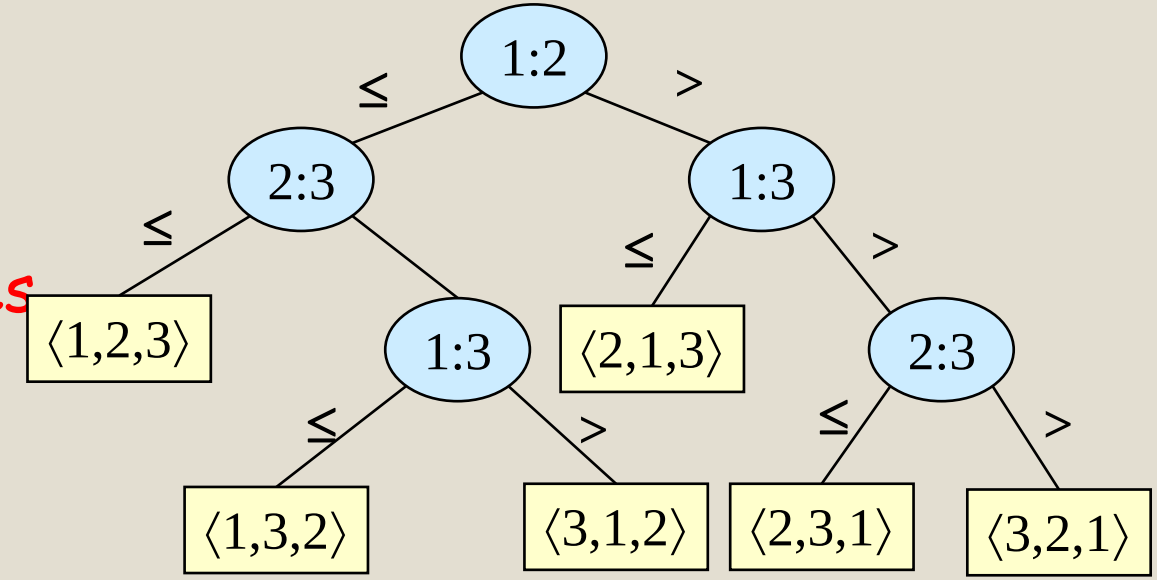
Theorem 8.1: Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof:

Goal: Lower bound the tree height

- Decision tree must have at least $n!$ leaves
- A binary tree of height h has at most 2^h leaves.
- Let l be the number of leaves.

$$n! \leq l \leq 2^h$$



Decision Tree for sorting a 3-item array

Decision Trees

Theorem 8.1: Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof

$$n! \leq l \leq 2^h$$

$\Longrightarrow$ {Transitive Property}

$$n! \leq 2^h$$

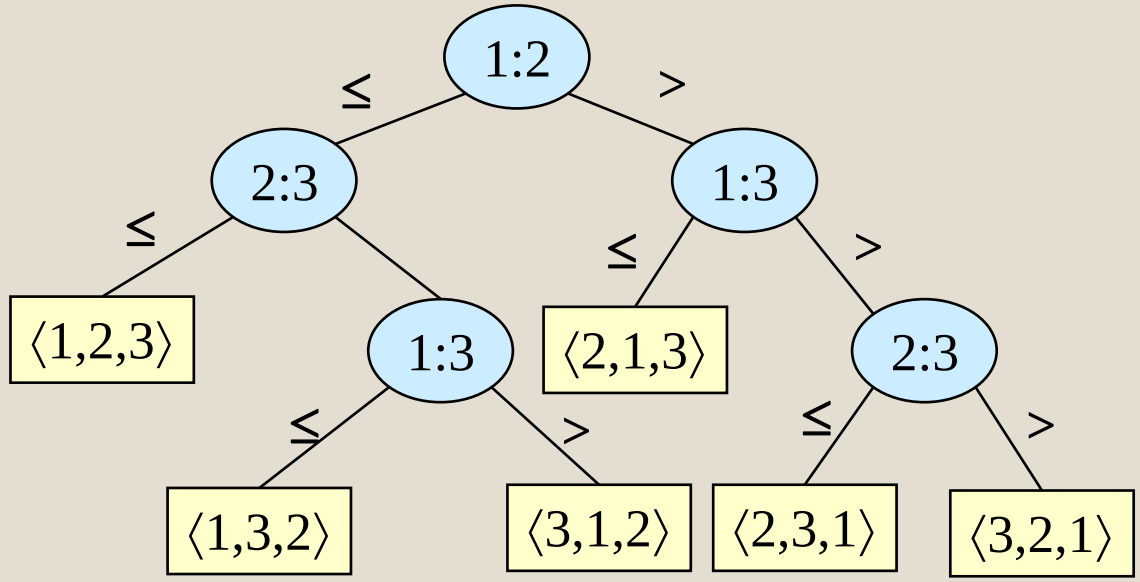
$\Longrightarrow$ {Take log in each side}

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

$\Longrightarrow$ {Eq. 3.28, Sec. 3.3}

$$\Omega(n \log n) \leq h$$

There exists a permutation that requires at least $\Omega(n \log n)$ to reach.



Decision Tree for sorting a 3-item array

Non-comparison Sorts

- We've seen that, for any comparison sort

  Worst-case running time $\geq cn \lg n$ (for some c and all $n \geq n_0$)

- Can we sort without making comparisons?
  - If so, can they have better worst-case running time?
- Answer of both is yes
  - But we need to make assumption about input
  - Example: Counting Sort, Radix Sort, Bucket Sort

Counting Sort

- Key *assumption*: numbers to be sorted are integers in $\{0, 1, 2, \ldots, k\}$.
- Input: $A[1:n]$, where $A[j] \in \{0, 1, 2, \ldots, k\}$ for $j = 1, 2, \ldots, n$.

  Array A and values n and k are given as parameters.
- Output: $B[1:n]$ sorted.
- Auxiliary Storage: $C[0..k]$
- Runs in linear time if $k = O(n)$.

Counting Sort

```
COUNTING-SORT(A, n, k)
1   let B[1:n] and C[0:k] be new arrays
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to n
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i − 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

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

Array to sort, $A[i] \in \{0, 1, \ldots, 5\}$

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

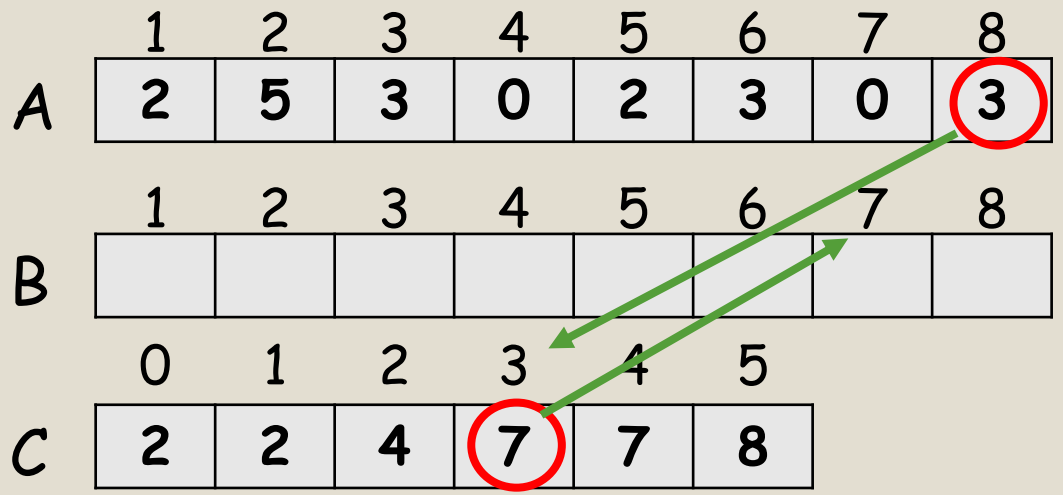
Array C after line 5

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

Array C after line 8

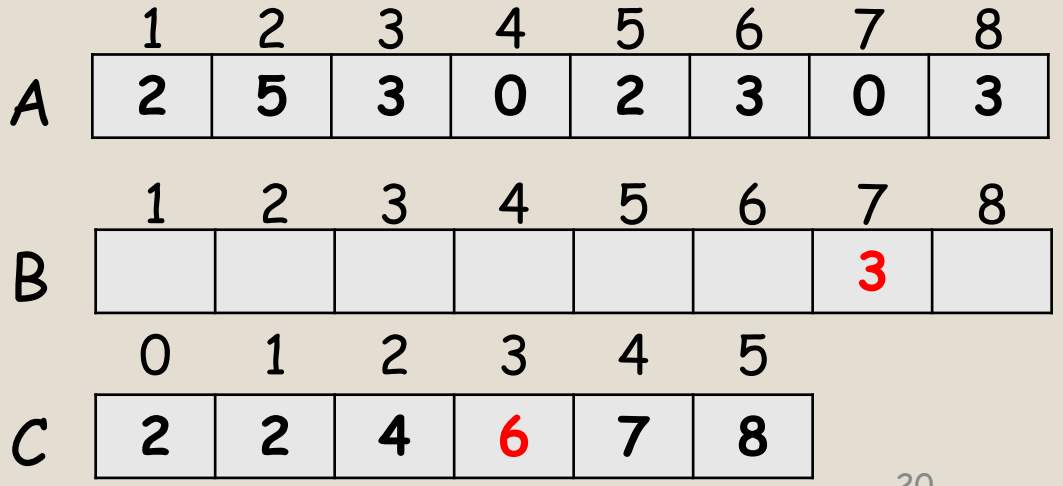
Counting Sort

```
COUNTING-SORT(A, n, k)
 1  let B[1:n] and C[0:k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```



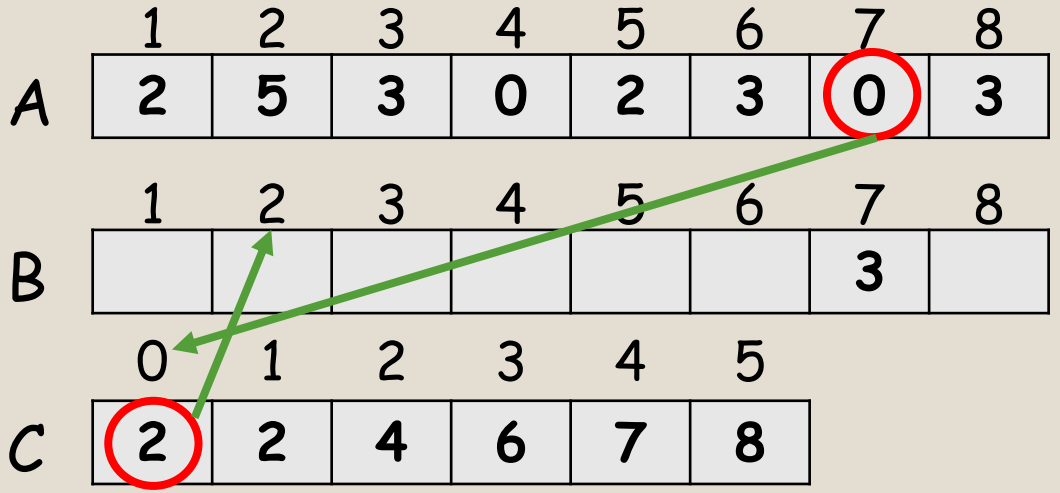
Before 1st iteration of for loop in lines 11-13

After 1st iteration of for loop in lines 11-13



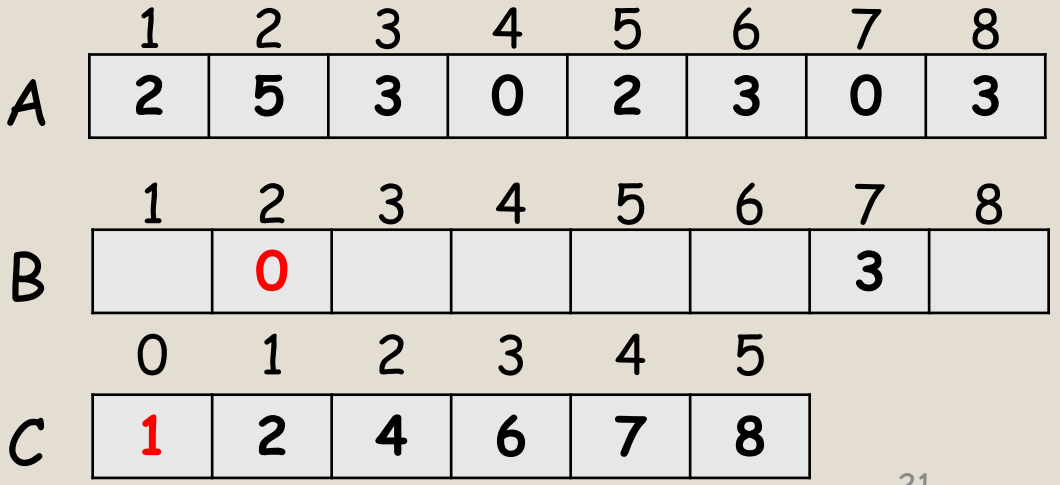
Counting Sort

```
COUNTING-SORT(A, n, k)
 1  let B[1:n] and C[0:k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```



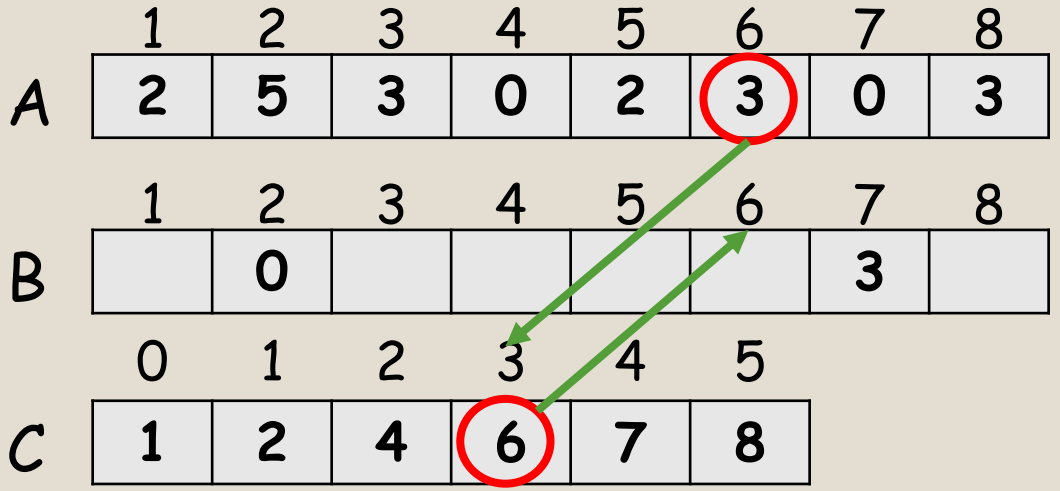
Before 2nd iteration of for loop in lines 11-13

After 2nd iteration of for loop in lines 11-13



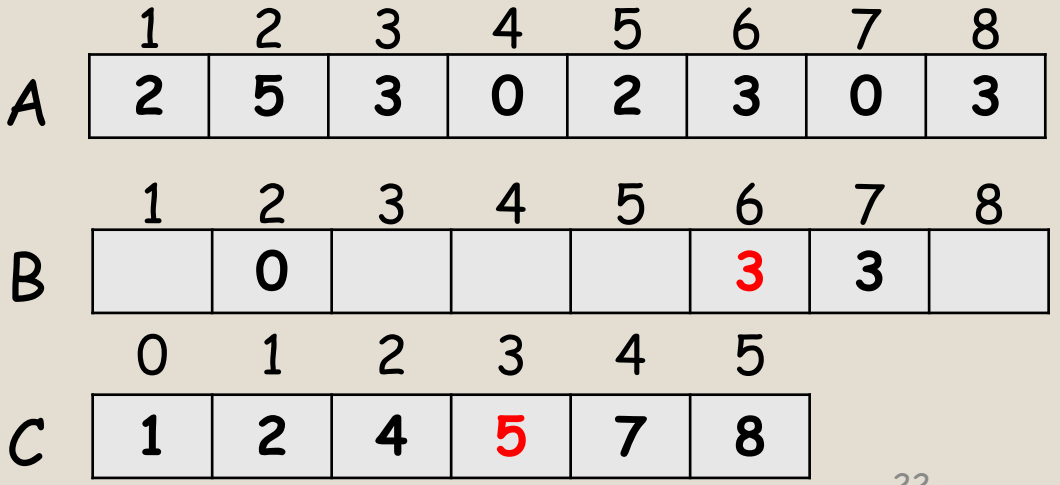
Counting Sort

```
COUNTING-SORT(A, n, k)
 1  let B[1:n] and C[0:k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```



Before 3rd iteration of for loop in lines 11-13

After 3rd iteration of for loop in lines 11-13



Counting Sort

```
COUNTING-SORT(A, n, k)
 1  let B[1:n] and C[0:k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

After all iterations

	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	2	3	3	3	5

Question: What about array *C* at the end?

What does each element in *C* mean?

Counting Sort

```
COUNTING-SORT(A, n, k)
1   let B[1:n] and C[0:k] be new arrays
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to n
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i − 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

After all iterations

	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	2	3	3	3	5

Question: What is about array C at the end?

What does each element in C mean?

Counting Sort: Time Complexity

```
COUNTING-SORT(A, n, k)
1   let B[1 : n] and C[0 : k] be new arrays
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to n
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i − 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Lines 2-3: $\boldsymbol{\Theta(k)}$

Lines 4-5: $\boldsymbol{\Theta(n)}$

Lines 7-8: $\boldsymbol{\Theta(k)}$

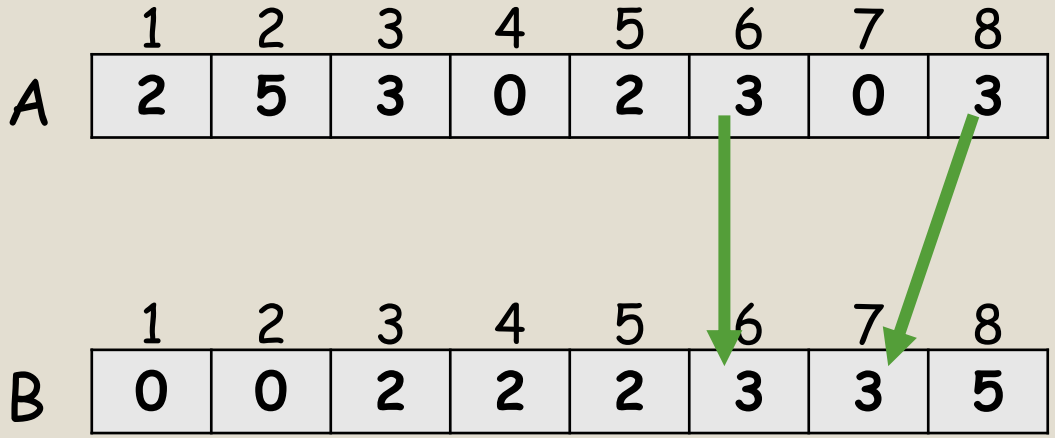
Lines 11-13: $\boldsymbol{\Theta(n)}$

Running time: $\Theta(n + k)$

$k = O(n) \Longrightarrow$ *Running time:* $\Theta(n)$

Counting Sort

- Counting Sort is **stable**
  - Same value elements appear in the same order in the output array as in the input array

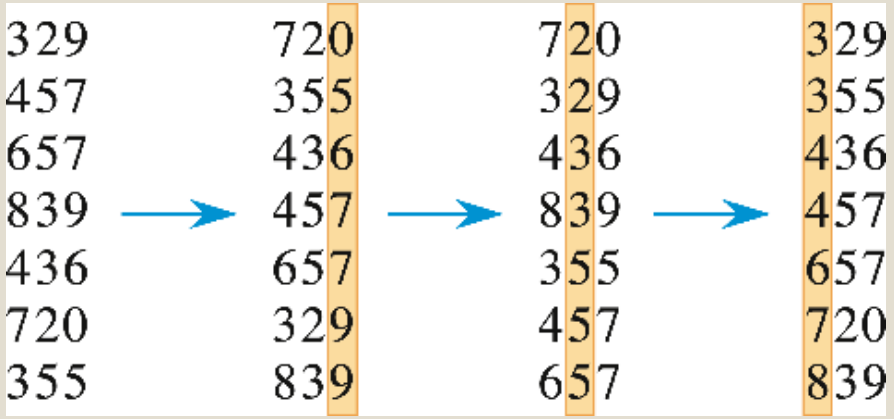


Radix Sort

Key idea: sort on the "least significant digit" first and on the remaining digits in sequential order. The sorting method used to sort each digit must be "stable". **Why?**

```
RADIX-SORT(A, n, d)
1  for i = 1 to d
2      use a stable sort to sort array A[1 : n] on digit i
```

Start by sorting the *least* significant digit.



Radix Sort: Time Complexity

```
RADIX-SORT(A, n, d)
1  for i = 1 to d
2      use a stable sort to sort array A[1 : n] on digit i
```

Start by sorting the *least* significant digit.

- n numbers, each number has d digits, each digit can have k possible values
  - For example, in the previous example, $d = 3$ and $k = 10$
- If counting sort is used for each pass, then each pass takes $\Theta(n + k)$ time (i.e., line 2 takes $\Theta(n + k)$ time)
- Running time of radix sort: $\Theta(d \cdot (n + k))$ time
  - $d = \Theta(1)$ and $k = \Theta(n) \Longrightarrow$ Running time is linear

Bucket Sort

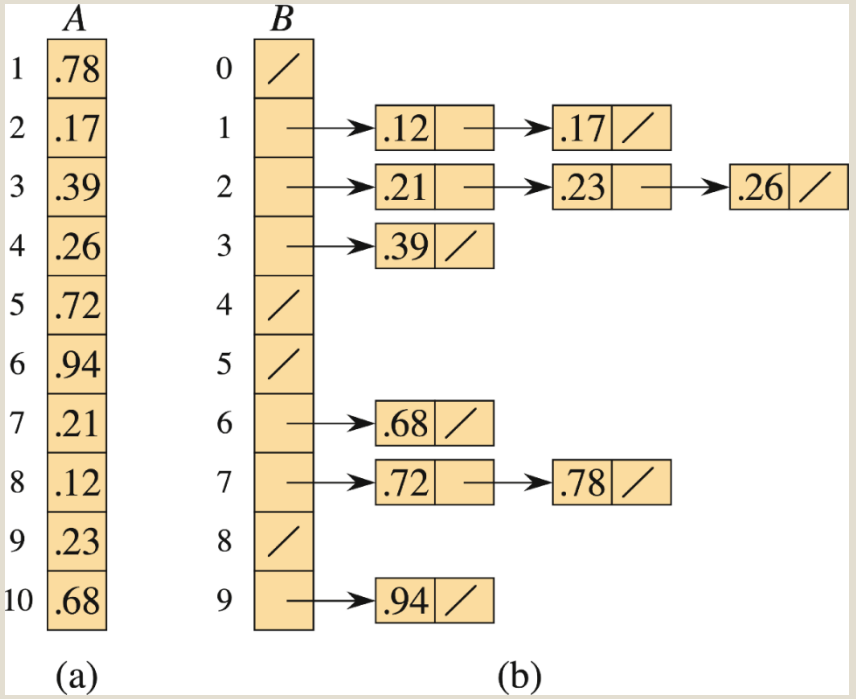
- Assumes input is generated by a random process that distributes the elements uniformly over [0, 1).
- **Idea:**
  - Divide [0, 1) into *n* equal-sized buckets.
  - Distribute the *n* input values into the buckets.
  - Sort each bucket.
  - Then go through the buckets in order, listing elements in each one.

Bucket Sort

Input: $A[1:n]$, where $0 \le A[i] < 1$ for all i.

Auxiliary array: $B[0:n-1]$ of linked lists, each list initially empty.

```
BUCKET-SORT(A, n)
1  let B[0:n − 1] be a new array
2  for i = 0 to n − 1
3       make B[i] an empty list
4  for i = 1 to n
5       insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7       sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```



Bucket Sort: Time Complexity

Input: $A[1:n]$, where $0 \le A[i] < 1$ for all i.
Auxiliary array: $B[0:n-1]$ of linked lists, each list initially empty.

```
BUCKET-SORT(A, n)
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3       make B[i] an empty list
4  for i = 1 to n
5       insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7       sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```

All lines except line 7 takes $O(n)$ time:

- Line 1: $\Theta(1)$ time
- Lines 2-3: $\Theta(n)$ time
- Lines 4-6: $\Theta(n)$ time
- Line 8: $\Theta(n)$ time
- Line 9: $\Theta(1)$ time

Line 7 takes $O(n_i^2)$ time.

- n_i is a random variable
- n_i denotes #of items in bucket i

Bucket Sort: Time Complexity

Input: $A[1:n]$, where $0 \le A[i] < 1$ for all i.
Auxiliary array: $B[0:n-1]$ of linked lists, each list initially empty.

```
BUCKET-SORT(A, n)
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3       make B[i] an empty list
4  for i = 1 to n
5       insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7       sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```

Running Time, $T(n) = \Theta(n) + \sum_{i=1}^{n} O(n_i^2)$

Worst-case: $T(n) = O(n^2)$

- All items in a single bucket

Best-case: $T(n) = \Theta(n)$

- One item in each bucket

Average-case:

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

Assuming, uniform distribution

$$E[T(n)] = \Theta(n)$$

Thank You!