Lecture #7
Sort (4)

Algorithm

JBNU Spring 2021

Jinhong Jung

In Previous Lecture

Desired properties of a sorting algorithm

Name	Stable	In-place : Extra memory	# of comparisons		# of swaps		Adaptive
			Best	Worst	Best	Worst	
Selection	No	Yes: $O(1)$	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n)$	No
Bubble	Yes	Yes: $O(1)$	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n^2)$	No
Opt. bubble	Yes	Yes: $O(1)$	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$	Yes
Insertion	Yes	Yes: $O(1)$	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$	Yes
Merge	Yes	No: $O(n)$	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	No
Quick	No	Yes: $O(\log n)$	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$O(n^2)$	No

Quick sort

- Divide the input based on a pivot & sort them recursively
  - Lomuto partition gives $O(n\log n)$ average time complexity
  - (Optimized) quick sort has $O(\log n)$ extra space

Heap sort

- Build a heap from the input & repeatedly extract the max

In This Lecture

Analysis of heap sort

Discussion on advanced sorting algorithms

- Which of them is better when?

Theoretic lower bound of comparison-based sorting algorithm

- Can we make a sorting algorithm faster than $\Omega(n \log n)$?

Non-comparative sorting algorithms

- Fast under special conditions

Outline

Analysis of heap sort

Discussion on advanced sorting algorithms

Theoretic lower bound of comparison-based sorting algorithm

Non-comparative sorting algorithms

Heap Sort (1) - Remind

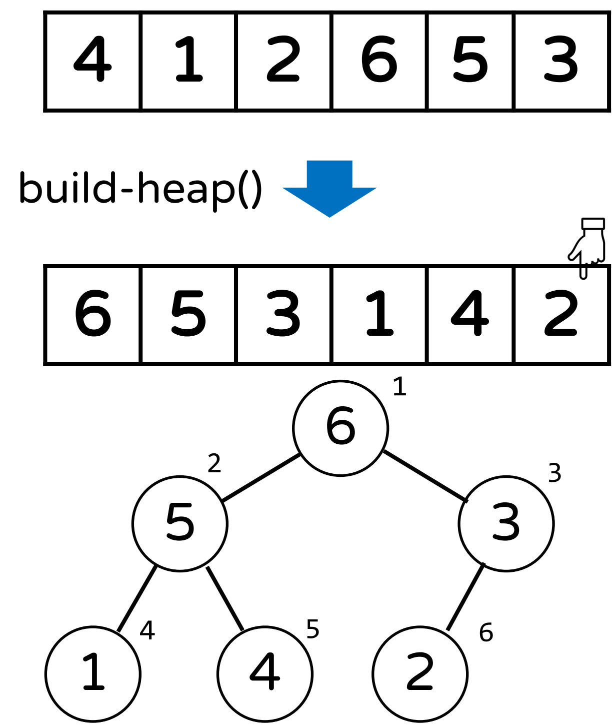
Idea of heap sort

- **Step 1)** Build a heap from the input
- **Step 2)** For each loop, extract the max from the heap & place it into the front of sorted area

```
def heap-sort(A, n):
    build-heap(A, n)
    for i ← n downto 2:
        swap A[i] and A[1]
        down-heap(A, 1, i-1)
```

```
def build-heap(A, n):
    for i ← ⌊n/2⌋ downto 1:
        down-heap(A, i, n)
```

Input A



heap-sort()

Heap Sort (2) - Analysis

Correctness of heap sort

- At each time, the maximum is correctly extracted from the heap and the sorted area is correctly expanded

Time complexity of heap sort

- Heap sort requires $\Theta(n \log n)$ time
  - $\Theta(n)$ for building a heap from the input array (refer to 114p)
  - $\Theta(n \log n)$ for sorting elements based on the heap
    - At each time, it requires $O(\log n)$ at most due to down-heap() $\Rightarrow O(n \log n)$
    - Theoretical lower bound of a comparative algorithm is $\Omega(n \log n)$

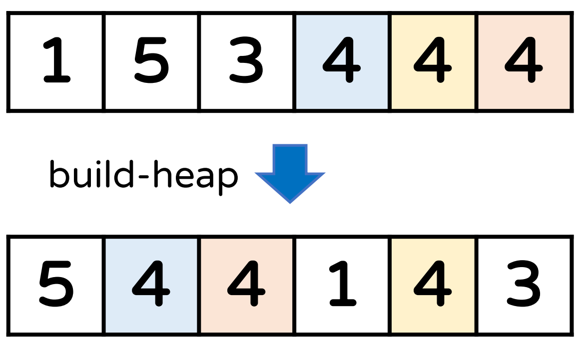
Space complexity of heap sort

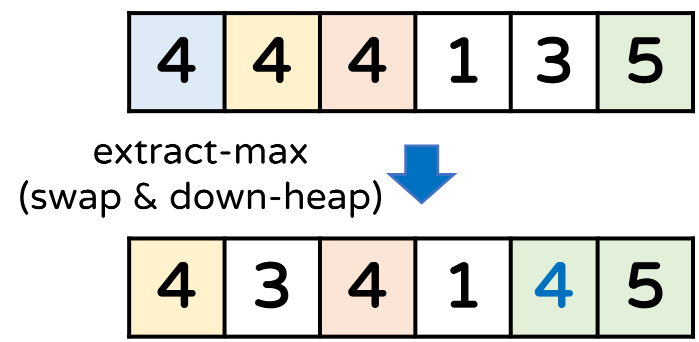
- $S(n) = \Theta(n)$
  - $O(n)$ is required to store n input data
  - $O(1)$ is required for extra space $\Rightarrow$ In-place algorithm

Heap Sort (3) - Analysis

Stability of heap sort

- Heap sort is not stable
  - Relative order of duplicated items can be inverted while building the heap and extracting the max





Adaptivity of heap sort

- Heap sort is not adaptive because of building of the max heap

Summary

Desired properties of a sorting algorithm

Name	Stable	In-place : Extra memory	# of comparisons		# of swaps		Adaptive
			Best	Worst	Best	Worst	
Selection	No	Yes: $O(1)$	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n)$	No
Bubble	Yes	Yes: $O(1)$	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n^2)$	No
Opt. bubble	Yes	Yes: $O(1)$	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$	Yes
Insertion	Yes	Yes: $O(1)$	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$	Yes
Merge	Yes	No: $O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No
Quick	No	Yes: $O(\log n)$	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$	No
Heap	No	Yes: $O(1)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No

Remarks

- No ideal answer in the above algorithms
- The average case time complexity of {merge, quick, heap} sort is $O(n \log n)$
  - Which of them is better when?

Outline

Analysis of heap sort

Discussion on advanced sorting algorithms ☜

Theoretic lower bound of comparison-based sorting algorithm

Non-comparative sorting algorithms

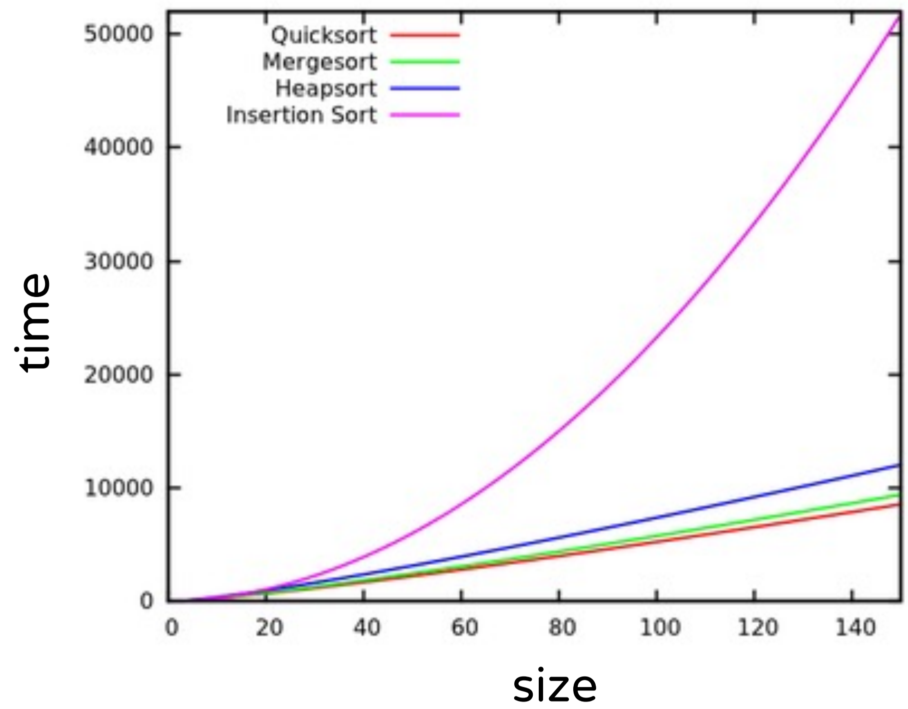
Which Is Better When? (1)

Suppose all of items are located in memory

- Then, **pracical choice is quick sort** since it has the lowest average time cost

- Quicksort: $11.667(n+1)\ln(n) - 1.74n - 18.74$
- Mergesort: $12.5n\ln(n)$
- Heapsort: $16n\ln(n) + 0.01n$
- Insertionsort: $2.25n^2 + 7.75n - 3ln(n)$

Ref: The Art of Computer Programming



Which Is Better When? (2)

What if the items are in a singly linked list?

- Then, merge sort is better since it can do merge() in one single pass
  - Quick sort and heap sort require swap operations which are inefficient in such a list

- This implies merge sort is beneficial **for sorting items on a disk** which forces us to read them sequentially
  - Merge sort is default for external sorting, and it’s also easy-to-parallelize

Which Is Better When? (3)

Suppose $O(n \log n)$ time and $O(1)$ extra space should be guaranteed (& no need to be super fast)

- Then, heap sort should be used
  - Quick sort has $O(n^2)$ time and $O(\log n)$ space for worst case
  - Merge sort has $O(n)$ space for worst case

What if we just need top-k items, not all?

- Then, heap sort is better (i.e., it takes $O(k \log n + n)$ time)
  - Extract the maximum k times from the heap (partially sorted)
  - Quick and merge sort need to sort all items requiring $O(n \log n)$ time in average

Outline

Analysis of heap sort

Discussion on advanced sorting algorithms

Theoretic lower bound of comparison-based sorting algorithm

Non-comparative sorting algorithms

Better Sorting Algorithm?

Q. Can we make a comparative sorting algorithm faster than $n\log n$ time?

- e.g., is there a sorting algorithm in $O(n)$?

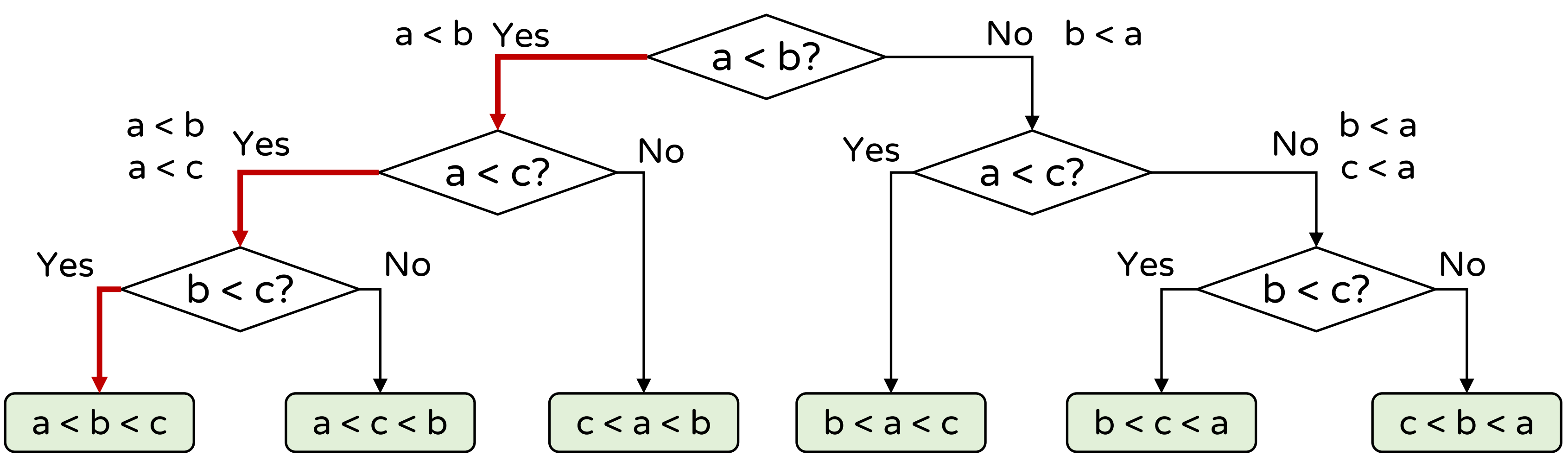
Unfortunately, it's impossible if we should compare arbitrary two elements

- The theoretical lower bound is $\Omega(n \log n)$
- Why?

Lower Bound Of Sorting (1)

Consider the problem of sorting $\{a, b, c\}$ comprised of three distinct items

- A sorting algorithm is represented as **decision tree**
  - Each node of the decision tree represents binary comparison
    - If a < b < c, then the comparisons on the red path are performed
    - **i.e., # of comparisons required in the worst case = the height of the tree**



Lower Bound Of Sorting (2)

Decision tree of a sorting algorithm of n items

- Binary tree having $n!$ leaf nodes
  - $n!$ indicates # of all possible permutations of n items
- The height of the binary tree having $n!$ leaf nodes is at least $\lceil \log_2 n! \rceil$
  - Tree of height h has at most 2^h leaf nodes, i.e., $2^h \geq n! \Leftrightarrow h \geq \log_2 n!$

$$\lceil \log_2 n! \rceil \geq \log_2 n!$$

$$= \sum_{i=1}^{n} \log_2 i \geq \sum_{i=1}^{n/2} \log_2 n/2$$

$$= \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log_2 n)$$

Outline

Analysis of heap sort

Discussion on advanced sorting algorithms

Theoretic lower bound of comparison-based sorting algorithm

Non-comparative sorting algorithms

- Counting sort
- Radix sort

Counting Sort (1)

Conditions for counting sort

- **C1)** Element of the array A should be natural number
  - Can include 0 if the array's index starts from 0
- **C2)** The maximum element should be at most k
  - If k is unknown, the maximum of the array A is set to k

$\mathrm{A} =$

5	2	2	3	3	1

$k = 5$

Main idea of counting sort

- **1)** Count each element of the array A from key 1 to k
- **2)** Enumerate each key by its frequency in the ascending order

Counting Sort (2)

Step 1) Count each element of the array A

- From key 1 to k

```
def counting_sort(A, k, n):
    initialize count of size k with zero
```

count

```
    for key in A:
        count[key] += 1
```

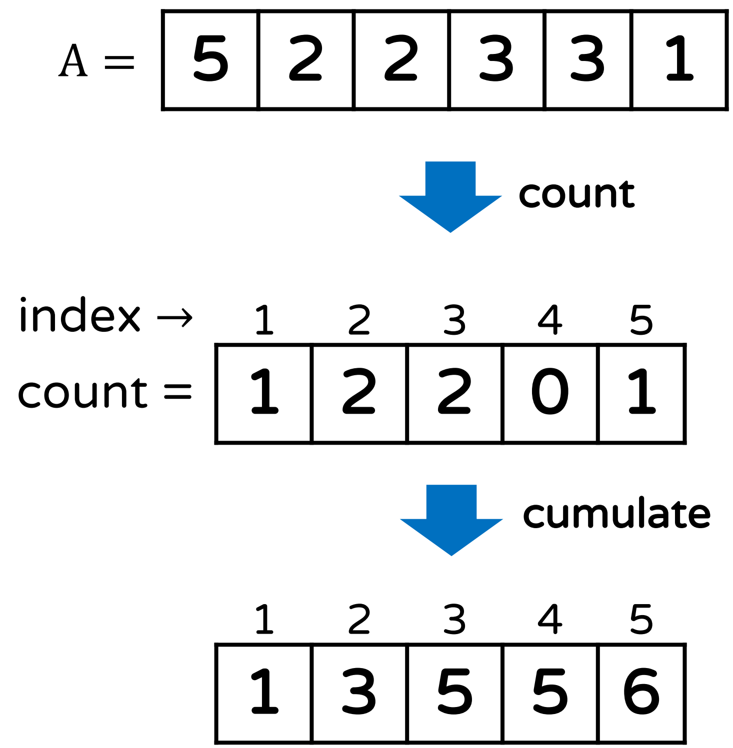
cumulate

```
    for i ← 2 to k:
        count[i] ← count[i] + count[i-1]
```

```
    initialize sorted_A of size n
```

sort

```
    for key in A:
        sorted_A[count[key]] = key
        count[key] -= 1

    return sorted_A
```



each value indicates the index where its element is located at

Counting Sort (3)

Step 1) Count each element of the array A

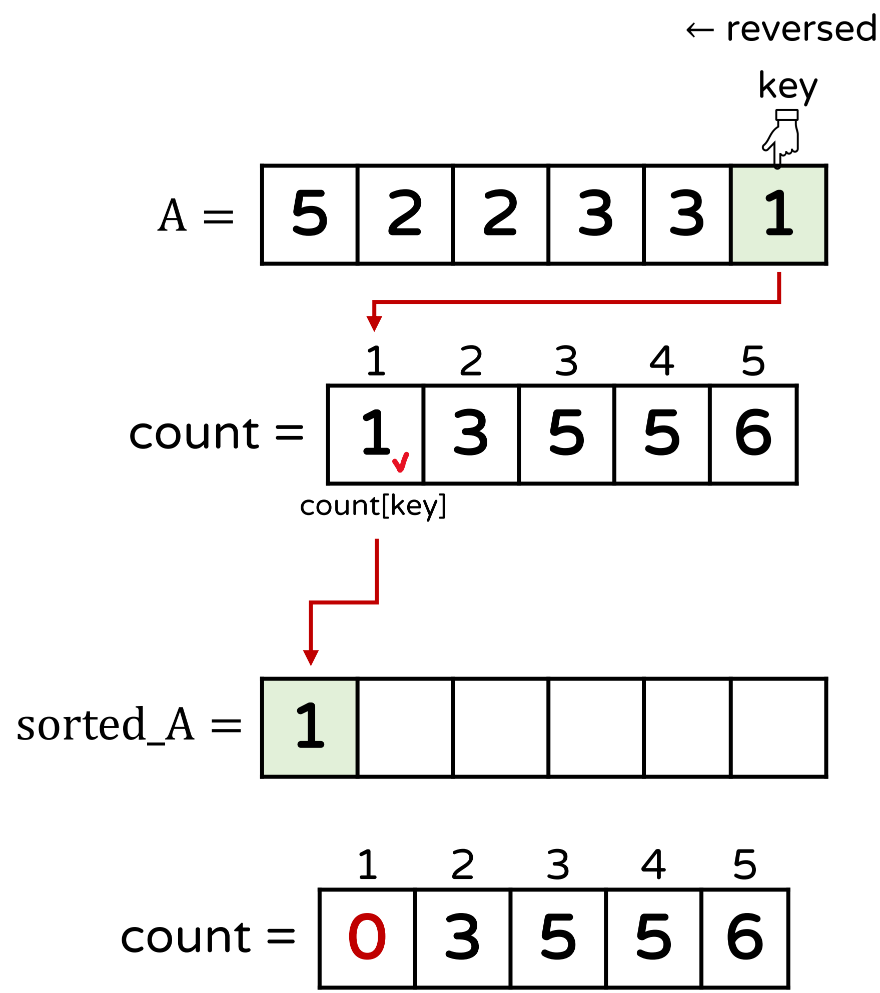
- From key 1 to k

```
def counting_sort(A, k, n):
    initialize count of size k with zero

    for key in A:
        count[key] += 1

    for i ← 2 to k:
        count[i] ← count[i] + count[i-1]

    initialize sorted_A of size n

    for key in reversed(A):
        sorted_A[count[key]] = key
        count[key] -= 1

    return sorted_A
```

count

cumulate

sort



Counting Sort (4)

Step 1) Count each element of the array A

- From key 1 to k

```
def counting_sort(A, k, n):
    initialize count of size k with zero
```

count

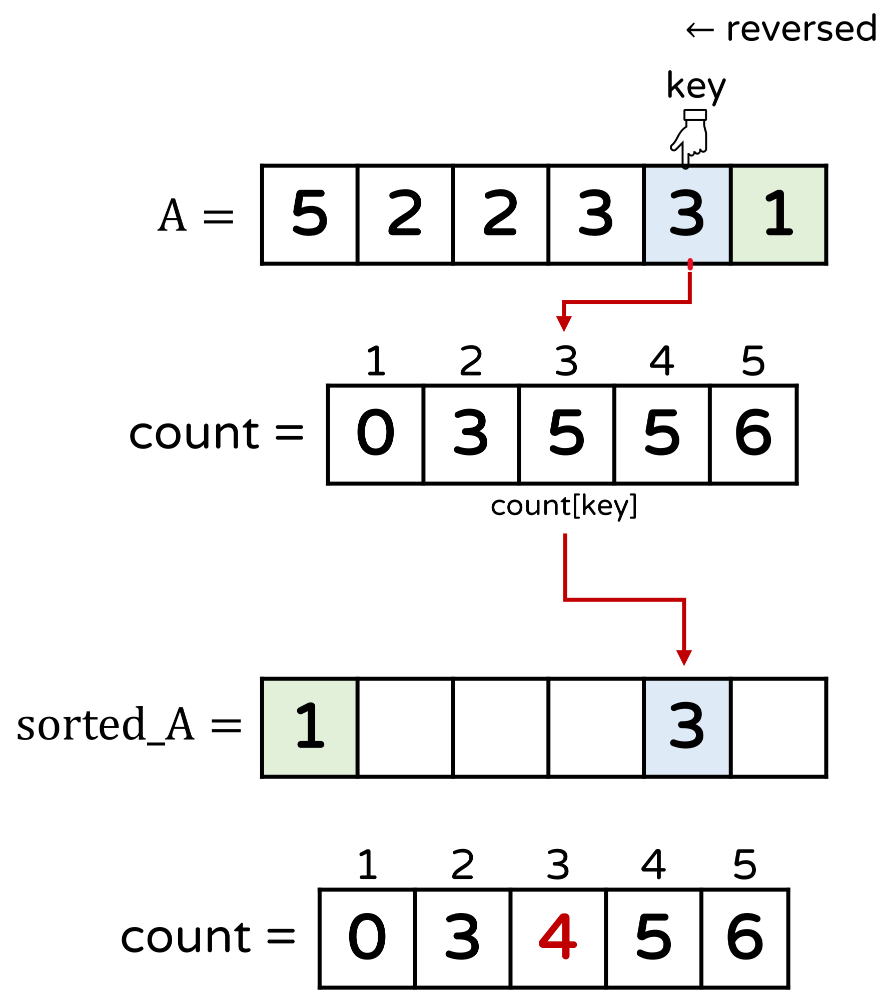
```
    for key in A:
        count[key] += 1
```

cumu late

```
    for i ← 2 to k:
        count[i] ← count[i] + count[i-1]

    initialize sorted_A of size n
```

sort

```
    for key in reversed(A):
        sorted_A[count[key]] = key
        count[key] -= 1

    return sorted_A
```

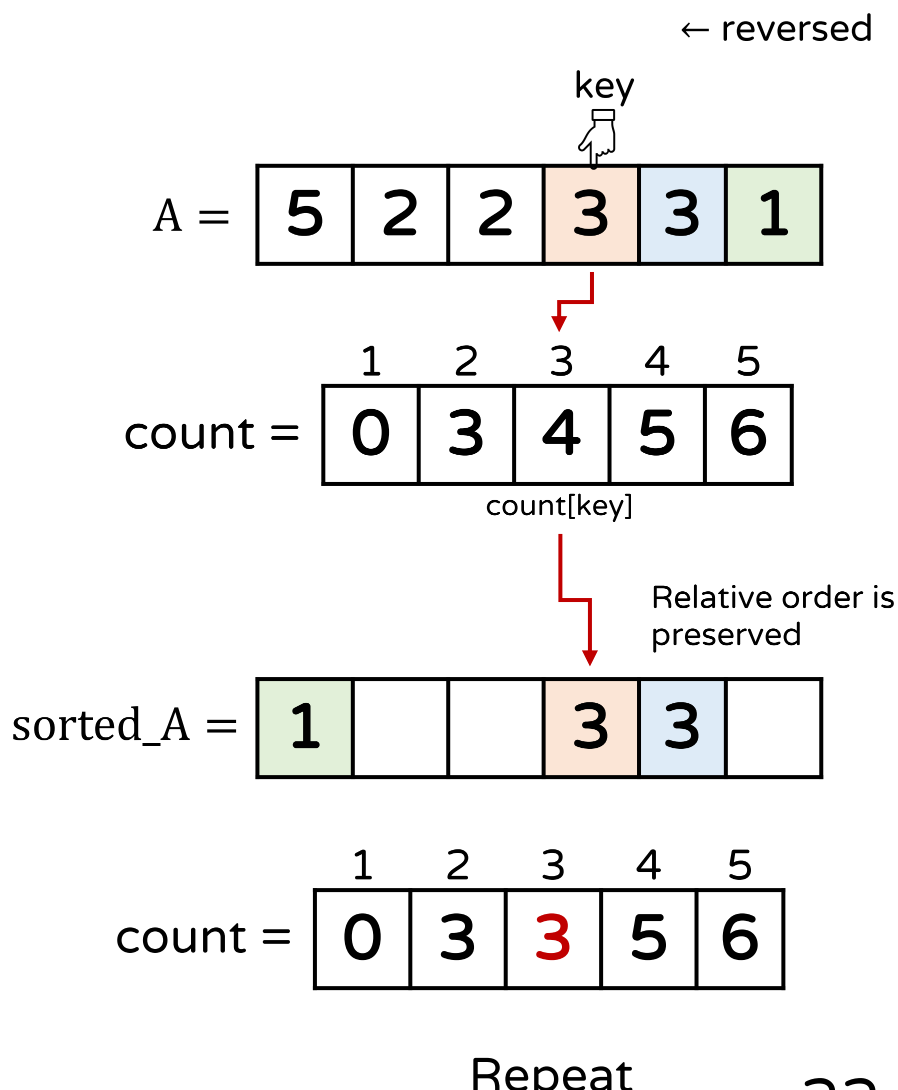


Counting Sort (5)

Step 1) Count each element of the array A

- From key 1 to k

```
def counting_sort(A, k, n):
    initialize count of size k with zero

    for key in A:                                  # count
        count[key] += 1

    for i ← 2 to k:                                # cumulate
        count[i] ← count[i] + count[i-1]

    initialize sorted_A of size n

    for key in reversed(A):                        # sort
        sorted_A[count[key]] = key
        count[key] -= 1

    return sorted_A
```



Counting Sort (6) - Analysis

Time & space complexity of counting sort

- $\Theta(n+k)$ for worst case (if $n \geq k$, then it's $\Theta(n)$)
  - If $k = n \log n$, it's $\Theta(n \log n) \Rightarrow$ no need to use it in this case

```
def counting_sort(A, k, n):
```

$\Theta(k)$ time

```
    initialize count of size k with zero
```

$\Theta(k)$ space

$\Theta(n)$ time

```
    for key in A:
        count[key] += 1
```

$\Theta(k)$ time

```
    for i ← 2 to k:
        count[i] ← count[i] + count[i-1]
```

$\Theta(n)$ time

```
    initialize sorted_A of size n
```

$\Theta(n)$ space (including A)

```
    for key in reversed(A):
        sorted_A[count[key]] = key
        count[key] -= 1

    return sorted_A
```

Counting Sort (7) - Analysis

Is counting sort in-place?

- **Counting sort is not-in-place**
  - Because of $\Theta(n + k)$ extra memory space

Stability of counting sort

- **Counting sort is stable**
  - Because the relative order of duplicate items is preserved

Adaptivity of counting sort

- **Counting sort is not adaptive**
  - Because the counting and sorting parts do not take the advantage of pre-sortness
  - But its time complexity is $\Theta(n)$

Outline

Analysis of heap sort

Discussion on advanced sorting algorithms

Theoretic lower bound of comparison-based sorting algorithm

Non-comparative sorting algorithms

- Counting sort
- Radix sort

Radix Sort (1)

Radix (base): # of unique digits

Conditions for radix sort

- **C1)** An element is represented by unique units such as digits or alphabet
  - e.g., [170, 45, 2, 24] or [b, ba, c, d, ef]
  - In the textbook, natural decimal numbers are considered, and the maximum number of digits is denoted by w

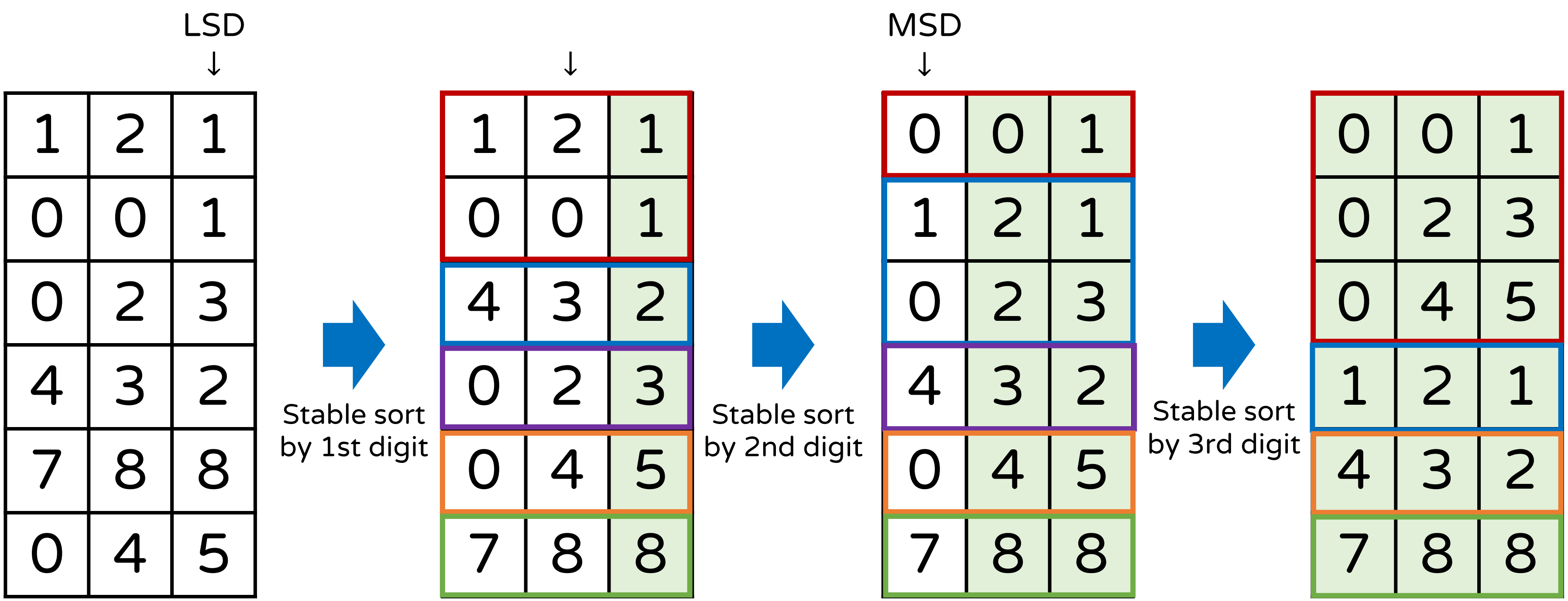
Main idea of radix sort

- **1)** From the least significant digit to the most significant digit, repeat **stably sorting** the input numbers based on each digit

Radix Sort (2)

Example of radix sort

- **1)** From the least significant digit to the most significant digit, repeat **stably sorting** the input numbers based on each digit



Radix Sort (3)

Pseudocode of radix sort

```
def radix_sort(A, n, w):
    for i ← 1 to w:
        A ← stable sort on A by the i-th digit
```

- If the elements of A are decimal numbers, then counting sort can be used for each step
  - Note that counting sort is a stable sorting algorithm and efficient (i.e., $\Theta(n + k) = \Theta(n)$ where $k = 10$)
  - Introduce an easier version using queues (but its principle is the same as the counting sort!)

Radix Sort (4)

```
def radix_sort(A, n, w):
    queue Q[10]
    for i ← 1 to w:
        # A ← stable sort on A by the i-th digit
        for j ← 1 to n: # push each number into d-th bucket sequentially
            d ← digit(A[j], i) # extract number d on i-th digit
            Q[d].enqueue(A[j])

        p ← 1
        for d ← 0 to 9: # extract each number from d-th bucket sequentially
            while Q[d] is not empty:
                A[p++] ← Q[d].dequeue()
```

1	2	1
0	1	1
0	2	3
4	3	2
7	8	8

Q

0	1	2	3	4	5	6	7	8	9
	121	432	023					788	
	011								

1	2	1
0	1	1
4	3	2
0	2	3
7	8	8

Radix Sort (5)

```
def radix_sort(A, n, w):
    queue Q[10]
    for i ← 1 to w:
        # A ← stable sort on A by the i-th digit
        for j ← 1 to n: # push each number into d-th bucket sequentially
            d ← digit(A[j], i) # extract number d on i-th digit
            Q[d].enqueue(A[j])

        p ← 1
        for d ← 0 to 9: # extract each number from d-th bucket sequentially
            while Q[d] is not empty:
                A[p++] ← Q[d].dequeue()
```

1	2	1
0	1	1
4	3	2
0	2	3
7	8	8

Q

0	1	2	3	4	5	6	7	8	9
	011	121	432					788	
		023							

0	1	1
1	2	1
0	2	3
4	3	2
7	8	8

Radix Sort (6)

```
def radix_sort(A, n, w):
    queue Q[10]
    for i ← 1 to w:
        # A ← stable sort on A by the i-th digit
        for j ← 1 to n: # push each number into d-th bucket sequentially
            d ← digit(A[j], i) # extract number d on i-th digit
            Q[d].enqueue(A[j])

        p ← 1
        for d ← 0 to 9: # extract each number from d-th bucket sequentially
            while Q[d] is not empty:
                A[p++] ← Q[d].dequeue()
```

0	1	1
1	2	1
0	2	3
4	3	2
7	8	8

Q

0	1	2	3	4	5	6	7	8	9
011	121			432			788		
023									

0	1	1
0	2	3
1	2	1
4	3	2
7	8	8

Radix Sort (7) - Analysis

Time complexity of radix sort

- $\Theta(w(n+k))$ for worst case (if w & k are small, it's $\Theta(n)$)
  - n: the number of items to be sorted
  - w: the length of digits of an item
  - k: the maximum number of digits

$$\overbrace{1234123}^{w=7}$$

k=10 for decimal scale

```
def radix_sort(A, n, w):
    queue Q[10]
    for i ← 1 to w:
        for j ← 1 to n:
            d ← digit(A[j], i) # d ← (A[j] / pow_10[i - 1]) % 10
            Q[d].enqueue(A[j])

        p ← 1
        for d ← 0 to 9:
            while Q[d] is not empty:
                A[p++] ← Q[d].dequeue()
```

$\Theta(n)$ time

$\Theta(n+k)$ time

\# of dequeues: n

\# of empty checks: $\sum_{d=0}^{9}(|Q[d]|+1) = n+k$

Radix Sort (8) - Analysis

Space complexity of radix sort

- $\Theta(n + k)$ for worst case (if k is small, it's $\Theta(n)$)
  - $\Theta(n)$ is required for storing the input data in both A and queues
  - $\Theta(k)$ is required for the array of queues
- Radix sort is not-in-place algorithm

Stability of radix sort

- **Radix sort is stable in nature (due to counting sort)**

Adaptivity of radix sort

- **Radix sort is not adaptive (due to counting sort)**
  - But its time complexity is $\Theta(n)$

Discussion

Counting sort and radix sort are under the same time complexity

- Counting sort is beneficial for repeated numbers in a limited range
  - e.g., sorting 1 million numbers all having value between 1 to 100
- Radix sort is beneficial when numbers are not so much repeated, but their lengths are fixed
  - e.g., sorting back account numbers of 1 million people each having 14-digit account numbers

What You Need To Know

Name	Stable	In-place : Extra memory	# of comparisons		# of swaps		Adaptive
			Best	Worst	Best	Worst	
Selection	No	Yes: $O(1)$	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n)$	No
Bubble	Yes	Yes: $O(1)$	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n^2)$	No
Opt. bubble	Yes	Yes: $O(1)$	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$	Yes
Insertion	Yes	Yes: $O(1)$	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$	Yes
Merge	Yes	No: $O(n)$	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	No
Quick	No	Yes: $O(\log n)$	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$O(n^2)$	No
Heap	No	Yes: $O(1)$	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	No
Counting	Yes	No: $O(n+k)$	No comparison : $O(1)$		Time: $O(n+k)$		No
Radix	Yes	No: $O(n+k)$	No comparison : $O(1)$		Time: $O(w(n+k))$		No

$\overbrace{1234123}^{w=7}$

k=10 for decimal scale

Remarks

- No ideal answer in the above algorithms
- The average case time complexity of {merge, quick, heap} sort is $O(n\log n)$

In Next Lecture

Selection algorithm

- Find i-th smallest number in an array
- Can we find the number in linear time for a worst case?

Thank You