Lecture #5
Sort (2)

Algorithm

JBNU Spring 2021

Jinhong Jung

In Previous Lecture

Sorting problem

- To efficiently rearrange elements in an array in an order

Basic sorting algorithms

- Take $\Theta(n^2)$ time for a worst case
- **Selection Sort**
  - Move the maximum to the end of unsorted area
    - Find the maximum directly by linearly searching
- **Bubble Sort**
  - Move the maximum to the end of unsorted area
    - Move the maximum by bubbling it up
- **Insertion Sort**
  - Pick the front of unsorted area
  - Insert it into its right position in sorted area

In This Lecture

Discussion on basic sorting algorithms

- Selection Sort
- Bubble Sort
- Insertion Sort

Advanced sorting algorithms

- Merge Sort
- Quick Sort

Outline

Discussion on basic sorting algorithms

Advanced sorting algorithms

Basic Sorting Algorithms

Take $\Theta(n^2)$ time for a worst case

- Selection, bubble, and insertion sort

Q. Which of them should we use?

- The answer depends on situations, but what situations?
- To answer this, we first need to analyze characteristics of each sorting algorithm
- To analyze these, we need to set criteria of ideal sorting algorithms
  - Especially for comparison-based algorithms

Properties of Sorting Algorithm

P1. Stability

P2. In-place algorithm

P3. # of comparisons (for worst cases)

P4. # of swaps (for worst cases)

P5. Adaptivity

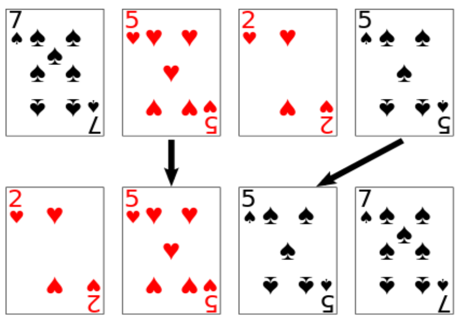
An algorithm is ideal if

- It is stable and adaptive
- It operates in place (i.e., takes $\Theta(1)$ extra space)
- Has $O(n \log n)$ and $O(n)$ comparisons and swaps, resp.

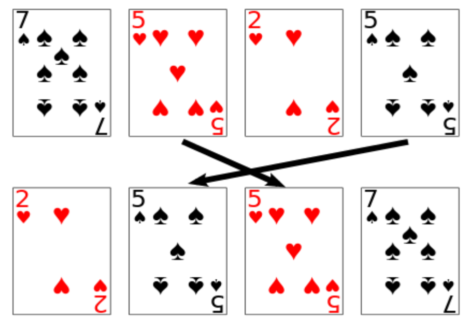
P1: Stability (1)

Stable sorting algorithm

- Duplicate elements are sorted in the same order that they appear input (i.e., equal keys are not reordered)



Stable



Not stable

  - Useful for sorting elements by a primary and secondary key
    - Suppose you sort a hand of cards in the order of ♣, ♦, ♥, and ♠, and within each suit, the cards are sorted by rank (number)
    - Step 1) sort the cards by rank using any sort
    - Step 2) do it again using a stable sort by suite

https://en.wikipedia.org/wiki/Sorting_algorithm#Stability

P1: Stability (2)

Stability of basic sorting algorithms

- (Basic) selection sort is **not stable**
  - Note that the leftmost maximum is linearly searched

5	4	5	1	2

➡

2	4	5	1	5

  - Q. Can we make the selection sort stable? (stable selection sort)
    - Yes, select the rightmost maximum & insert it into the end for each step
    - Push the remaining elements forward one cell to the left
- Bubble sort is **stable**
  - Duplicate items are never swapped
- Insertion sort is **stable**
  - A latter item never moves in the front of a former item when they have the same key

P2: In-place Algorithm

In-place sorting algorithms

- Sort the data within the input array (of size n) using no extra array (or data structure)
  - Algorithm which is not in-place is called **not-in-place/out-of-place**
- Require $\Theta(n)$ space to store input data, but usually use $\Theta(1)$ extra space for sorting
- Important when available memory is very limited

Selection, bubble, and insertion sorting algorithms are all in-place

- They do not use extra data structure whose size is proportional to n

P3&4: # of comparisons & swaps

Main components of comparison-based sorting algorithms

- Comparison & swap
- A single swap is heavier than a single comparison

Time complexity of each algorithm is analyzed based on # of comparisons & swaps

- Worst case: Input is sorted in reverse order
  - Selection sort: $\Theta(n^2)$ comparisons and $\Theta(n)$ swaps
  - Bubble sort: $\Theta(n^2)$ comparisons and $\Theta(n^2)$ swaps
  - Insertion sort: $\Theta(n^2)$ comparisons and $\Theta(n^2)$ swaps

Q. What about the best case?

Best Case Analysis (1)

Selection sort

- Swap is working only when two keys are different

```
def selection_sort(A, n):
    for end ← n downto 2:
        # selection stage
        k ← find the maximum's index among A[1⋯end]
        swap A[k] and A[end] (if they are different)
```

- Best case: an input array is already sorted
  - i-th maximum searching always takes $O(i)$ time regardless of the input
  - If it's sorted, then 'k' will be 'end'; thus, no swap is performed
- $\Theta(n^2)$ comparisons and $O(1)$ swap for the best case

Best Case Analysis (2)

Bubble sort

- Best case: an input array is already sorted

```
def bubble_sort(A, n):
    for end ← n downto 2:
        # bubble-up stage
        for i ← 1 to end – 1:
            if A[i] > A[i+1]:
                swap A[i] and A[i+1]
```

- In the bubble-up stage,
  - The comparison is always performed regardless of the input
  - If the input is sorted, no swap is performed
- Thus, $\Theta(n^2)$ comparisons and $O(1)$ swaps

Best Case Analysis (3)

Optimized bubble sort

- Best case: an input array is already sorted

```
def optimized_bubble_sort(A, n):
    for end ← n downto 2:
        # optimized bubble-up stage
        swapped ← false
        for i ← 1 to end - 1:
            if A[i] > A[i+1]:
                swap A[i] and A[i+1]
                swapped ← true
        if not swapped:
            return since it's sorted
```

  - If the input is sorted, then it'll be terminated at the first outer-loop
- Thus, $\Theta(n)$ comparisons and $O(1)$ swaps for the best case
  - The optimized stage is used for one-pass sorting checker
    - i.e., able to check whether the input is sorted within $O(n)$ time

Best Case Analysis (4)

Insertion sort

- Best case: an input array is already sorted

```
def insertion_sort(A, n):
    for i ← 2 to n:
        # insertion stage
        insert A[i] into its
        right position in A[1⋯i]
```

```
# i-th insertion stage
loc ← i – 1
item ← A[i]
while loc >= 1 and item < A[loc]:
    A[loc+1] ← A[loc]
    loc ← loc – 1
A[loc+1] ← item
```

- In the insertion stage, if it's already sorted
  - The inner while loop is not executive except for two comparisons
    - One backward movement is equivalent to one swap
    - For i-th loop, 2 comparisons and 0 swap are performed
- Thus, $\Theta(n)$ comparisons and $O(1)$ swaps for the best case

P5: Adaptivity (1)

Adaptive sorting algorithm

- Takes advantages of existing order in its input
  - The more presorted the input is, the faster it should be sorted
  - i.e., able to speed up to $O(n)$ if the input is **nearly** sorted
- Inversion of an array A
  - If $\mathrm{A}[i] > \mathrm{A}[j]$ where $i < j$, then the pair is called an inversion
  - $\mathrm{I}(\mathrm{A})$: number of all inversions in A
    - Inversions: (3, 1), (3, 2), (5, 1), (5, 2), (5, 4)
    - $\mathrm{I}(\mathrm{A}) = 5$

$i < j$

Index: $1 \quad 2 \quad 3 \quad 4 \quad 5$

Array: $3 \quad 5 \quad 1 \quad 2 \quad 4$

$\mathrm{A}[i] > \mathrm{A}[j]$

P5: Adaptivity (2)

Insertion sort

```
def insertion_sort(A, n):
    for i ← 2 to n:
        # insert A[i] into its right position in A[1⋯i]
        loc ← i - 1
        item ← A[i]
        while loc >= 1 and A[loc] > item:
            A[loc+1] ← A[loc]
            loc ← loc - 1
        A[loc+1] ← item
```

- Note that the while loop repeats only when the inversions on $\mathrm{A}[i]$ occurs
  - Roughly speaking, $T(n) = \mathrm{I}(\mathrm{A}) + n - 1$
  - If the input array A is nearly sorted, then $\mathrm{I}(\mathrm{A})$ is very small (i.e., almost constant) $\Rightarrow T(n) = C + n - 1 = \Theta(n)$
- Thus, insertions sort is adaptive!

P5: Adaptivity (3)

Optimized bubble sort

```
def optimized_bubble_sort(A, n):
    for end ← n downto 2:
        # optimized bubble-up stage
        swapped ← false
        for i ← 1 to end - 1:
            if A[i] > A[i+1]:
                swap A[i] and A[i+1]
                swapped ← true
        if not swapped:
            return # since it's sorted
```

- Optimized bubble sort is **adaptive**!
  - If the input is nearly sorted (e.g., one or two pairs are inversed), its complexity is roughly reduced to $O(n)$
  - Less adaptive than insertion sort due to the comparisons (> $\mathrm{I}(\mathrm{A})$)

P5: Adaptivity (4)

Bubble sort and selection sort

```
def bubble_sort(A, n):
    for end ← n downto 2:
        # bubble-up stage
        for i ← 1 to end - 1:
            if A[i] > A[i+1]:
                swap A[i] and A[i+1]
```

```
def selection_sort(A, n):
    for end ← n downto 2:
        # selection stage
        k ← find the maximum's
              index among A[1⋯end]
        swap A[k] and A[end]
```

- Regardless of the input patterns, they always performs $O(n^2)$ comparisons
- Thus, they are not adaptive

Summary

Name	Stable	In-place : Extra memory	# of comparisons Best	# of comparisons Worst	# of swaps Best	# of swaps Worst	Adaptive
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

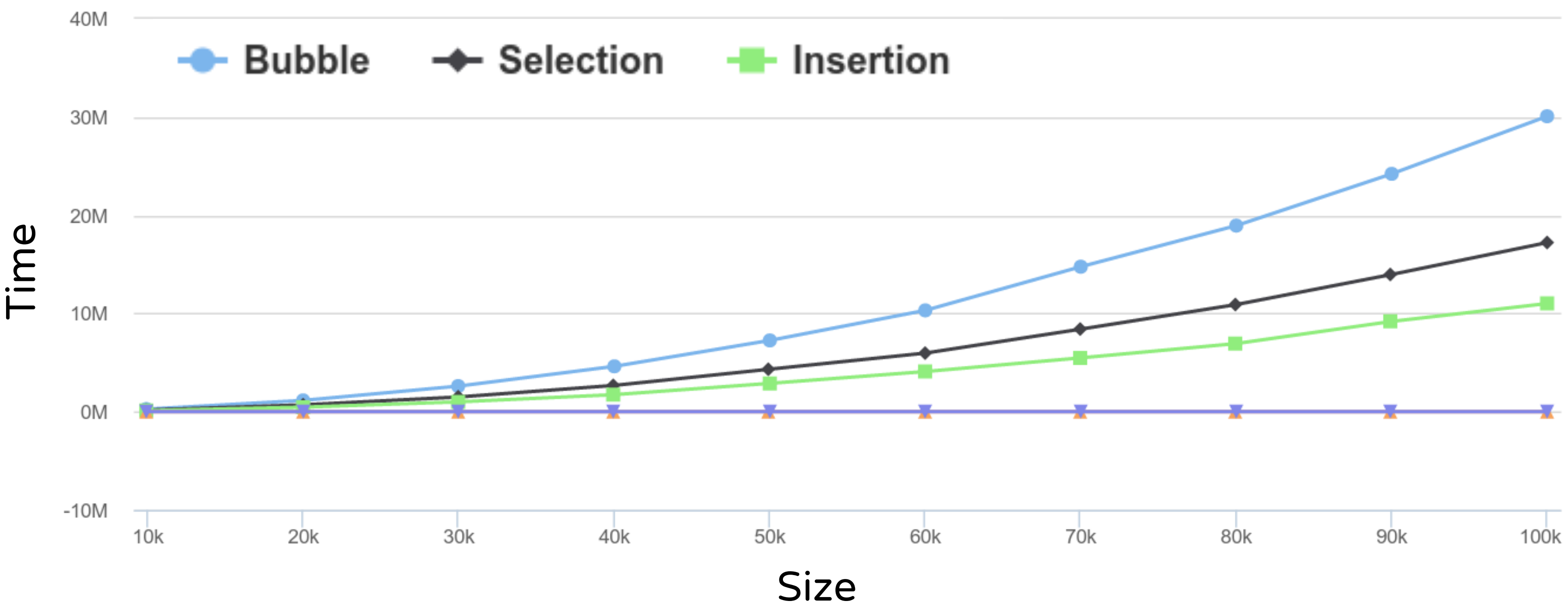
Remarks

- Selection sort can be modified to be stable
- No ideal answer in the above basic algorithms
  - The choice depends on applications
  - How about advanced sorting algorithms?

Practical Choice is Insertion

Among basic sorting algorithms ($O(n^2)$)

- Measure average time for randomly generated data



- Why? Insertion sort performs only necessary operations for sorting elements

https://titrias.com/ultimate-sorting-algorithms-comparison/

Outline

Discussion on basic sorting algorithms

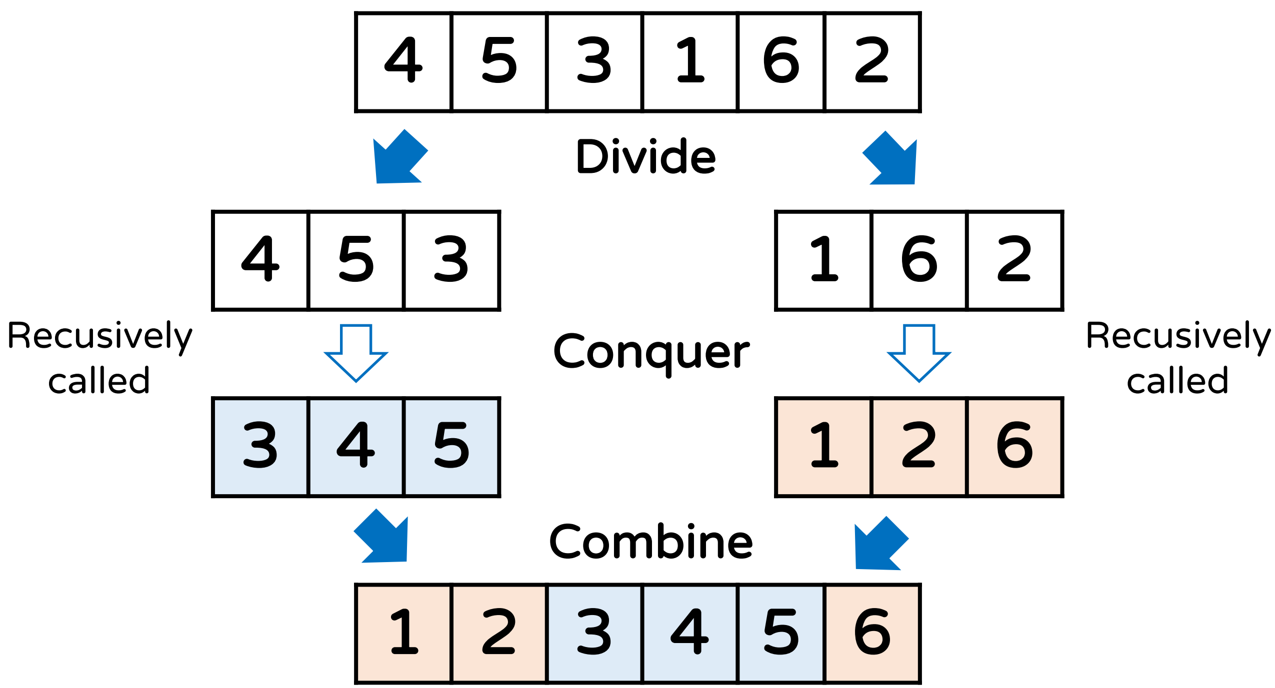
Advanced sorting algorithms

- **Merge Sort** ☞
- Quick Sort

Merge Sort (1)

Idea (based on divide & conquer)

- [**Divide**] split the input in half
- [**Conquer**] recursively sort the former and latter
- [**Combine**] merge the results so that merged entries are in the sorted order



Merge Sort (2)

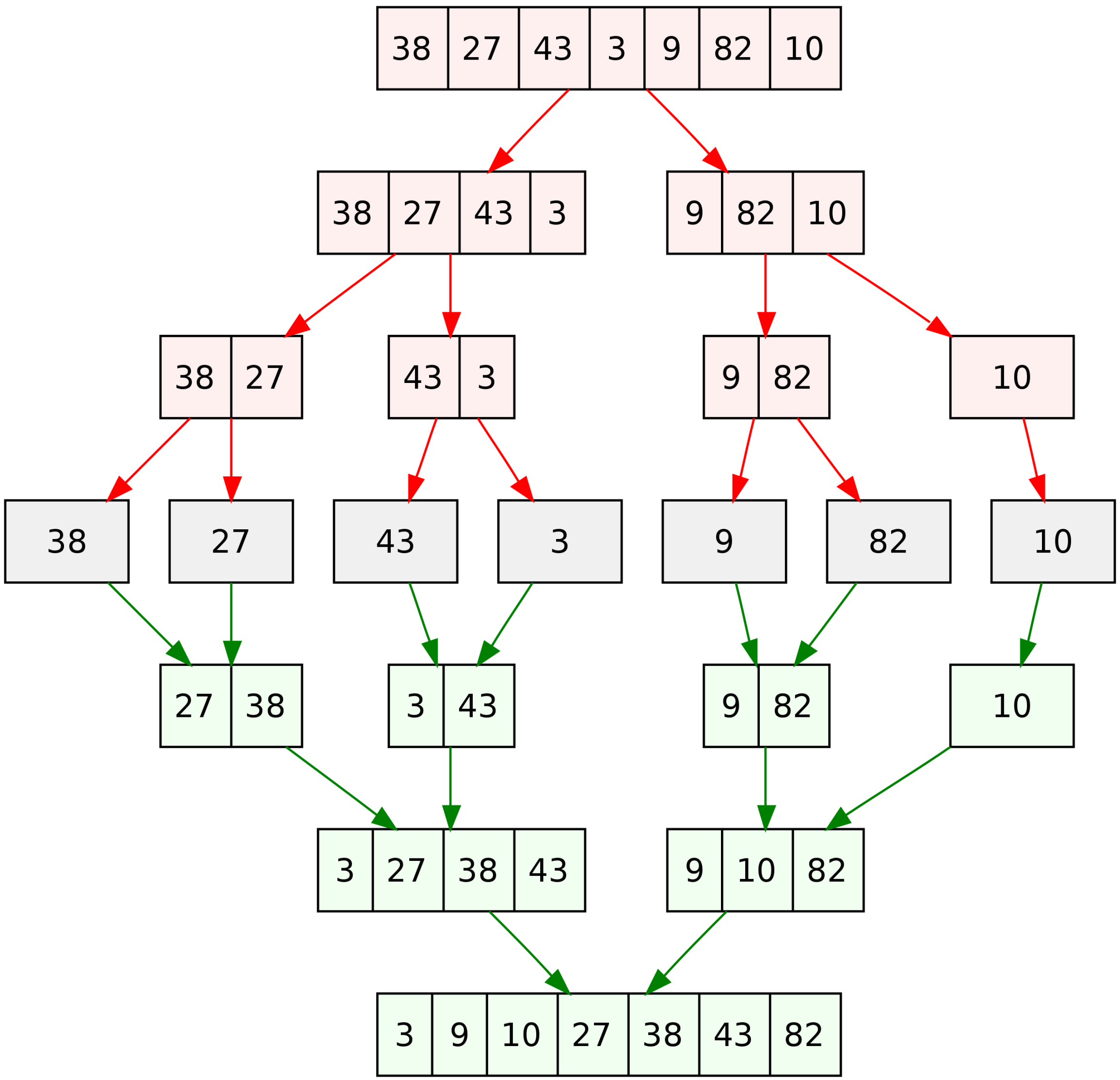
Pseudocode for merge sort

```
def merge_sort(A, l, r):
    if l < r:
        # [Divide] split the input in half
        pick middle point m in [l, r], i.e., m ← ⌊(l + r)/2⌋

        # [Conquer] recursively sort the former and latter
        merge_sort(A, l, m)
        merge_sort(A, m + 1, r)

        # [Combine] merge the results correctly
        merge(A, l, m, r)
```

- Then, `mege_sort(A, 1, n)` performs sorting the input array A in the ascending order

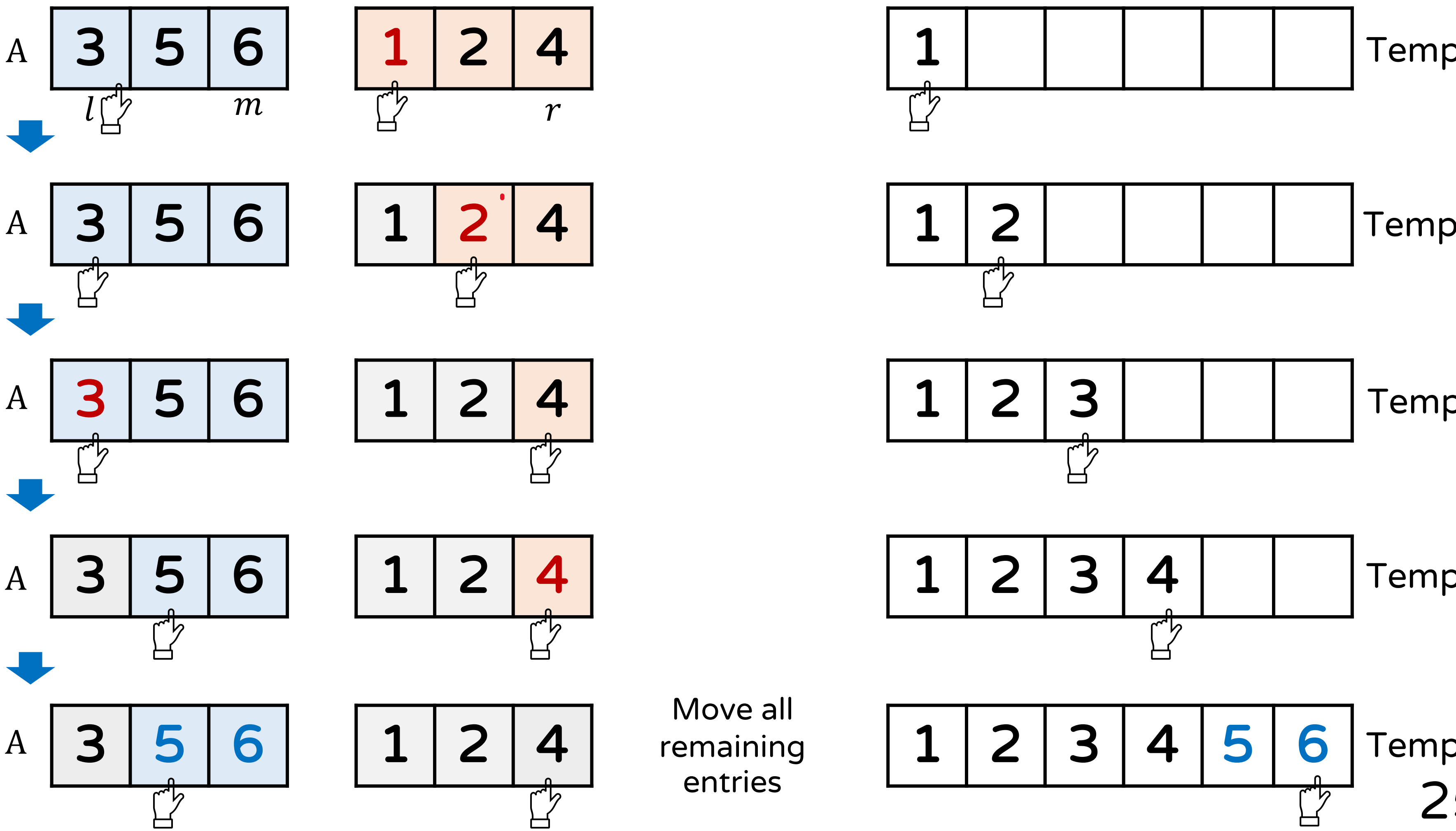
Merge Sort (3)

Example



Merge Sort (4) – How To Merge

Search for the minimum in each of them from the left to the right; move it to the temporary array



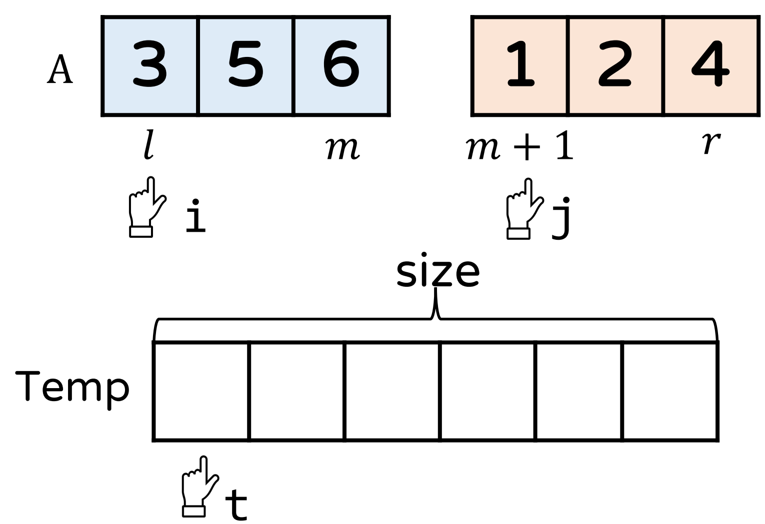
Merge Sort (5)

Pseudocode for merge sort

```
def merge(A, l, m, r):
    i ← l, j ← m+1, t ← 1
    set array temp[size] where size = r - l + 1

    while i <= m and j <= r:
        if A[i] <= A[j]:
            temp[t++] ← A[i++]
        else:
            temp[t++] ← A[j++]

    while i <= m:
        temp[t++] ← A[i++]
    while j <= r:
        temp[t++] ← A[j++]

    copy temp to A[l⋯r]
```

Sequential merge (move to temp)

Move remaining items



# of comparisons for a worst case: size - 1

Analysis Of Merge Sort (1)

Correctness of merge sort

```
def merge_sort(A, l, r):
    if l < r:
        m ← ⌊(l + r)/2⌋
        merge_sort(A, l, m)
        merge_sort(A, m + 1, r)
        merge(A, l, m, r)
```

- Poof by induction
  - Base case: when the size is 1, it is sorted by itself
  - Inductive step
    - Assume merge sort correctly sorts any array of size $k/2$
    - Let A_1 and A_2 be $\mathrm{A}[1 \cdots k/2]$ and $\mathrm{A}[k/2 + 1 \cdots k]$, respectively
    - Then, they are correctly sorted by the assumption since their sizes are $k/2$
    - And merge() will correctly merge them in a sorted order
    - Thus, merge sort correctly sorts an array of size k
    - By induction, merge sort correctly sorts an array of any size n

Analysis Of Merge Sort (2)

Time complexity of merge sort

```
def merge_sort(A, l, r):
    if l < r:
        m ← ⌊(l + r)/2⌋
        merge_sort(A, l, m)
        merge_sort(A, m + 1, r)
        merge(A, l, m, r)
```

- Let $T(n)$ be the time complexity of `mege_sort(A, 1, n)`
  - `merge()` takes $O(n)$ time when its input size is n

$$T(n) = \begin{cases} C, & l \geq r \\ 2T\left(\frac{n}{2}\right) + Cn, & l < r \end{cases}$$

- By master theorem, $T(n) = \Theta(n \log n)$

$$\frac{f(n)}{h(n)} = \frac{Cn}{n} = C = \Theta(1) \Rightarrow \Theta(h(n) \log n) = \Theta(n \log n)$$

Analysis Of Merge Sort (3)

Space complexity of merge sort

- $S(n) = \Theta(n)$
  - To store the input data whose size is n
  - Need a temporary array whose size is n
- For extra memory, it requires $\Theta(n)$ space
- This indicates merge sort is out-of-place algorithm!

Stability of merge sort

- Merge sort is stable
  - Since merge() searches for the minimum from the left to the right for each step
  - If there are duplicate items, the leftmost item are first moved to the temp array

Analysis Of Merge Sort (4)

of comparisons for a worst case

- Each merge performs `size-1` comparisons at most
- Sum of # of comparisons incurred by every node except leaves in the solution tree of merge sort = $O(n \log n)$

of swaps for a worst case

- There is no swap operation, but if a single movement to temp = a single swap
  - Each merge perform `size` movements at most
- Thus, it is $O(n \log n)$

Adaptivity of merge sort

- Merge sort is not adaptive since the comparisons and movements are always performed regardless of input

Summary

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

Remarks

- Selection sort can be modified to be stable
- A swap operation is replaced with
  - A backward movement in insertion sort
  - A movement to temp array in merge sort
- No ideal answer in the above algorithms

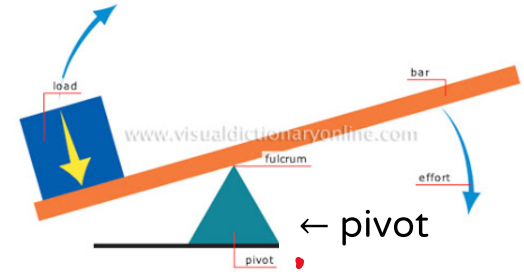
Outline

Discussion on basic sorting algorithms

Advanced sorting algorithms

- Merge Sort
- **Quick Sort - Part I** ☜

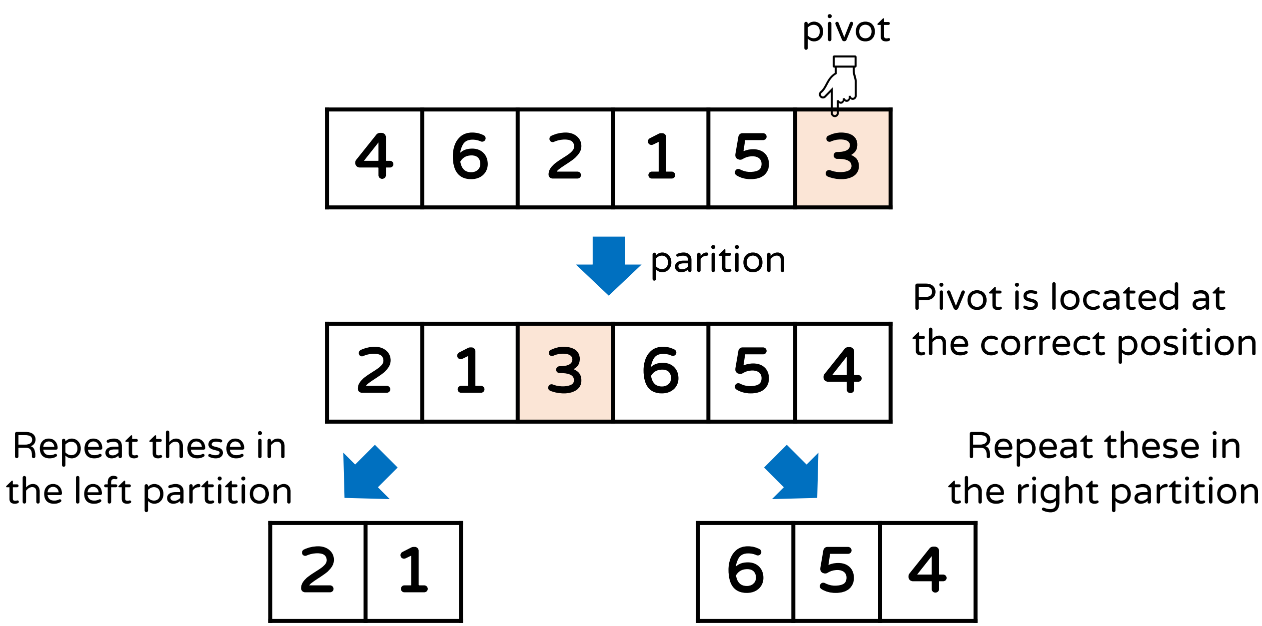
Quick Sort (1)



← pivot

Idea (based on divide & conquer)

- **[Divide]** select a pivot & partition the input array based on the pivot satisfying the following condition
  - Elements before pivot ≤ pivot ≤ elements after pivot
- **[Conquer]** recursively sort left & right partitions, resp.



Quick Sort (2)

Pseudocode of quick sort

```
def quick_sort(A, l, r):
    if l < r:
        # [Divide] pick a pivot and partition A by the pivot
        p ← partition(A, l, r) # p is the pivot's index

        # [Conquer] recursively sort the left and right
        quick_sort(A, l, p - 1)
        quick_sort(A, p + 1, r)
```

- **Main issue is how to design the partition function**
  - Numerous ways for selecting a pivot(s)
    - In the textbook, the last element for a given partition is selected as a pivot
  - Need to be efficiently partitioned in place

What You Need To Know

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

Merge sort

- Divide the array in half & sort them recursively

Quick sort

- Divide it based on a pivot & sort them recursively

In Next Lecture

Advanced sorting algorithms

- Quick Sort – Part II
  - How to partition an array based on the selected pivot
  - Analyze the properties of quick sort

- Heap Sort

[Advice]

- If you have some spare time, think about the average time complexity of each algorithm that we've learned

Thank You