Lecture 5
Sorting Algorithms

Subject Lecturer: Kevin K.F. YUEN, PhD.

Acknowledgement: Slides were offered from Prof. Ken Yiu.
Some parts have been revised and indicated.

Sorting on array

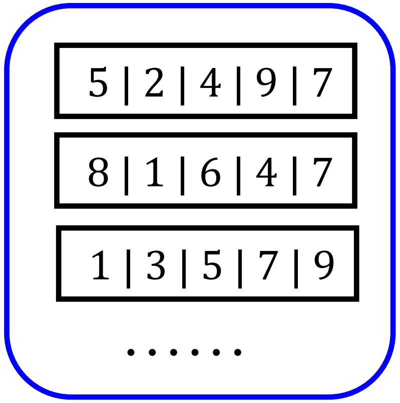
- Applications of sorting
  - Websites, databases
- Sorting problem
  - Input: an array *A* of *n* numbers
  - Output: a sorted array *A* (in ascending order)

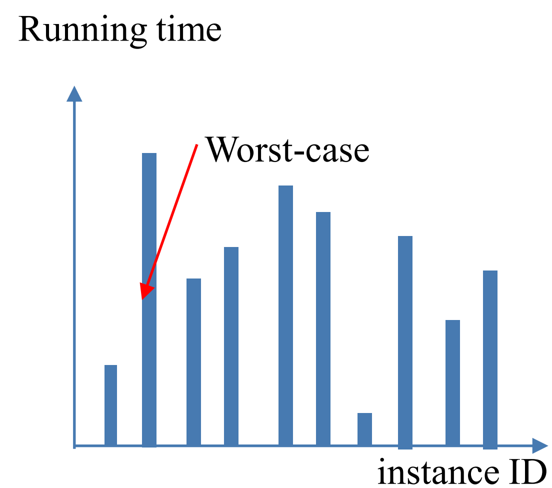
Input:	29 | 70 | 85 | 40 | 47 | 26 | 13 | 59
Output:	13 | 26 | 29 | 40 | 47 | 59 | 70 | 85

Time complexity of sorting algorithms

Algorithm	*Worst-case time complexity*	*Average-case time complexity*
Selection sort	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$
Heap sort	$O(n \log n)$	$O(n \log n)$
Merge sort	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n^2)$	$O(n \log n)$

Possible input arrays at n=5





Our Roadmap

- Slow sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
- Fast sorting algorithms (by *divide-and-conquer*)
  - Merge sort
  - Quick sort
- How to *analyze* the running time of these divide-and-conquer algorithms?

Selection Sort

- In the outer for-loop, each iteration
  - The subarray $A[0..i-1]$ is already sorted
  - Select the smallest value from $A[i..n-1]$, swap it with $A[i]$

input 5 | 2 | 4 | 9 | 7

i=0, after line 6: 2 | 5 | 4 | 9 | 7

i=1, after line 6: 2 | 4 | 5 | 9 | 7

i=2, after line 6: 2 | 4 | 5 | 9 | 7

i=3, after line 6: 2 | 4 | 5 | 7 | 9

Selection-Sort (Array $A[0..n-1]$)

```
1. for integer i ← 0 to n–2
2.       k ← i
3.       for integer j ← i+1 to n–1
4.             if  A[k] > A[j]  then
5.                    k ← j
6.       swap A[i] and A[k]
```

Bubble Sort

- Scan the array sequentially
  - Swap adjacent elements if they are not in the ascending order
- Repeat the above until no swaps are needed

<u>Bubble-Sort (Array $A[0..n-1]$)</u>

```
repeat
    isUpdated ← false
    for integer i ← 1 to n–1
        if A[i–1] > A[i] then
            swap A[i–1] and A[i]
            isUpdated ← true
until isUpdated = false
```

input	5 | 2 | 4 | 9 | 7
repeat iteration 1, swap #1	2 | 5 | 4 | 9 | 7
repeat iteration 1, swap #2	2 | 4 | 5 | 9 | 7
repeat iteration 1, swap #3	2 | 4 | 5 | 7 | 9
repeat iteration 2, no swap	2 | 4 | 5 | 7 | 9

Insertion Sort

- Let x be the value of $A[i]$ at line 2 (before the while-loop)
- Move x to the position such that
  - The left hand size $\leq x$
  - The right hand size $> x$

? | ? | ? | x | …

$\leq x$ | x | $>x$ | …

<u>Insertion-Sort (Array $A[0..n-1]$)</u>

1. for integer $i \leftarrow 1$ to $n-1$
2. $\quad j \leftarrow i$
3. $\quad$ while $j > 0$:
4. $\qquad$ *if(*$A[j-1] > A[j]$*)*
5. $\qquad\quad$ swap $A[j]$ and $A[j-1]$
6. $\qquad j \leftarrow j-1$

input	5 | 2 | 4 | 9 | 7
i=1, swap #1	2 | 5 | 4 | 9 | 7
i=2, swap #1	2 | 4 | 5 | 9 | 7
i=3, no swap	2 | 4 | 5 | 9 | 7
i=4, swap #1	2 | 4 | 5 | 7 | 9

Updated. KKF Yuen

Best-Case Input vs. Worst-Case Input

- In *comparison sorting* algorithms, the ***number of element comparisons*** can be used to estimate the running time
  - E.g., the comparison $A[i-1] > A[i]$ in Bubble Sort (Line 4)

- *Examples* for the Bubble Sort at n=5
  - The best-case input at n=5 — 2 | 4 | 5 | 7 | 9
  - The worst-case input at n=5 — 9 | 7 | 5 | 4 | 2

- How many element comparisons are performed by Bubble Sort in these two examples?

Bubble-Sort (Array $A[0..n-1]$)

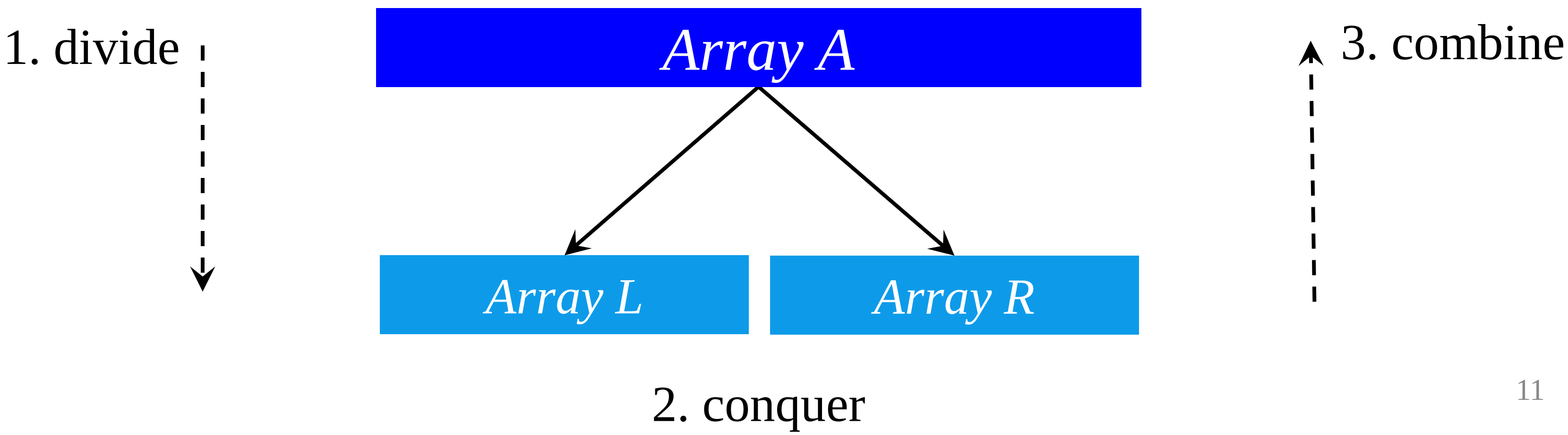
```
1. repeat
2.     isUpdated ← false
3.     for integer i ← 1 to n–1
4.         if  A[i–1] > A[i]  then
5.             swap A[i–1] and A[i]
6.             isUpdated ← true
7. until  isUpdated = false
```

Our Roadmap

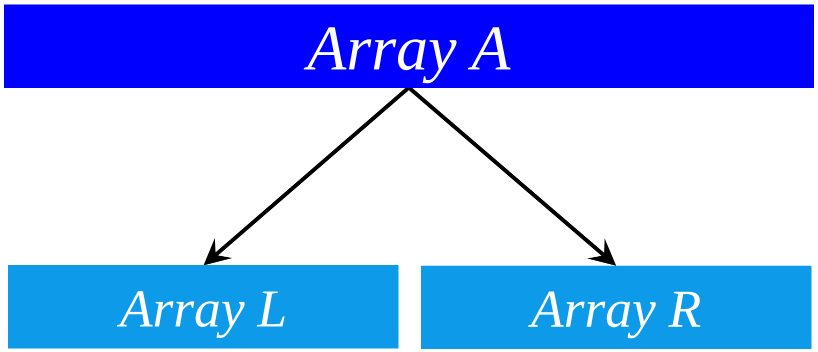
- Slow sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
- Fast sorting algorithms (by *divide-and-conquer*)
  - Merge sort
  - Quick sort
- How to *analyze* the running time of these divide-and-conquer algorithms?

Divide and Conquer (D&C)

- *Divide and Conquer* (D&C) is a technique for designing algorithms
  - **Divide**: divide the problem into smaller subproblems
  - **Conquer**: solve each subproblem recursively
  - **Combine**: combine the solutions of subproblems into the solution of the original problem



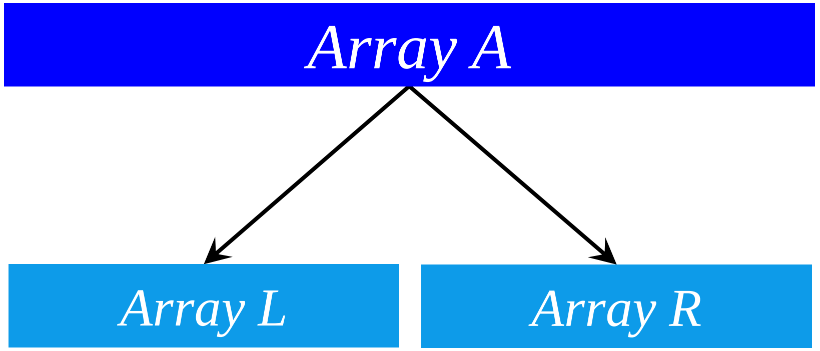
Divide-and-Conquer



Algorithm	**Merge Sort**	**Quick Sort**
Divide	Divide the array *A* into two equal-sized sub-arrays *L* and *R*	Partition the array *A* into two sub-arrays *L* (with small values) and *R* (with large values)
Conquer	Sort each sub-array recursively	Sort each sub-array recursively
Combine	Merge the sorted sub-arrays *L* and *R*	NIL

Merge Sort

- *Divide*: divide the array A into two sub-arrays (L and R) of $n/2$ numbers each
- *Conquer*: sort two sub-arrays recursively
- *Combine*: merge two **sorted** sub-arrays into a sorted array

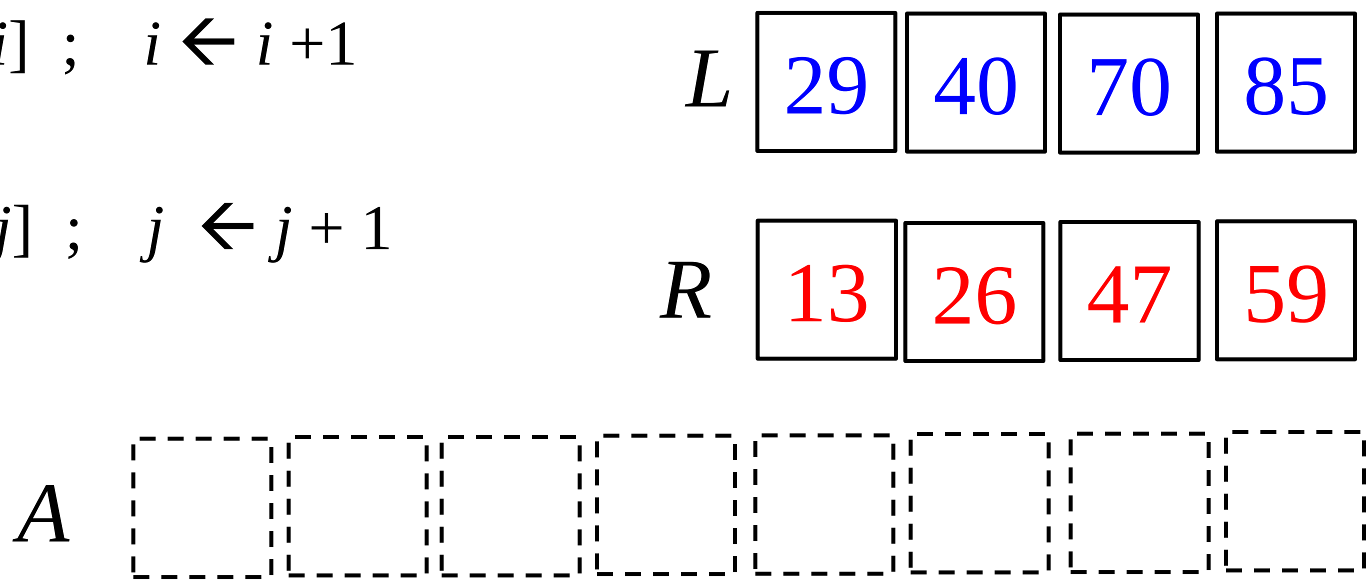


Merge Sort: Combine Phase

Merge (Array $L[0..n_L-1]$, Array $R[0..n_R-1]$)

Pre-condition:
Arrays L and R are already sorted

1. $n_A \leftarrow n_L + n_R$
2. create a new array $A[0..n_A-1]$
3. $i \leftarrow 0$; $j \leftarrow 0$
4. for $k \leftarrow 0$ to n_A-1
5.   if ($j=n_R$) or ($i< n_L$ and $L[i] < R[j]$)
6. $A[k] \leftarrow L[i]$; $i \leftarrow i+1$
7.   else
8. $A[k] \leftarrow R[j]$; $j \leftarrow j+1$
9. return A



Merge Sort

<u>Merge-Sort (Array $A[0..n–1]$)</u>

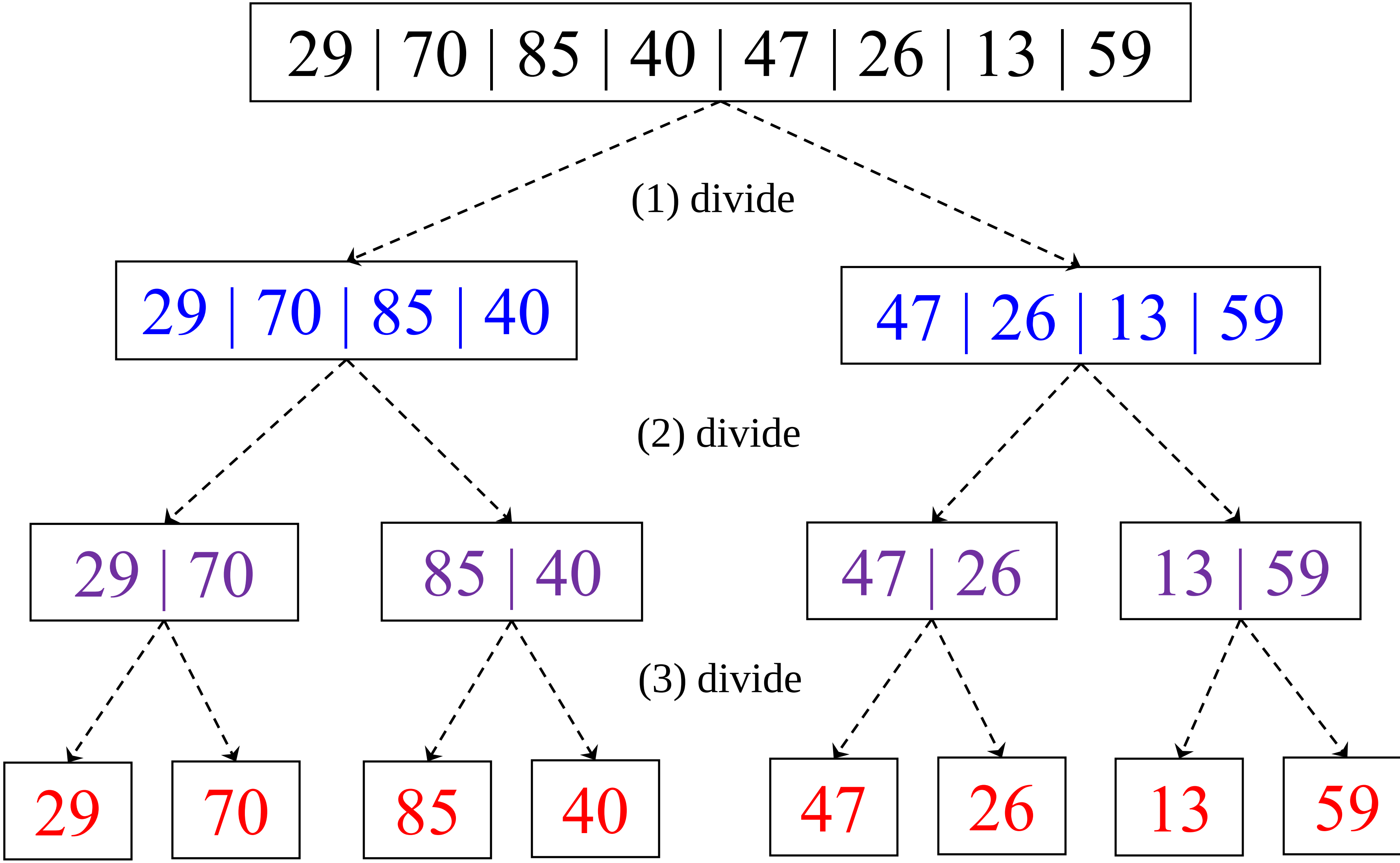
1. if $n > 0$
2. $m \leftarrow \lfloor n/2 \rfloor$
3. Array $L \leftarrow A[0..m–1]$
4. Array $R \leftarrow A[m..n–1]$
5. Merge-Sort (L)
6. Merge-Sort (R)
7. $A[0..n–1] \leftarrow$ Merge (L, R)

Divide: divide the array into two sub-arrays of $n/2$ numbers each

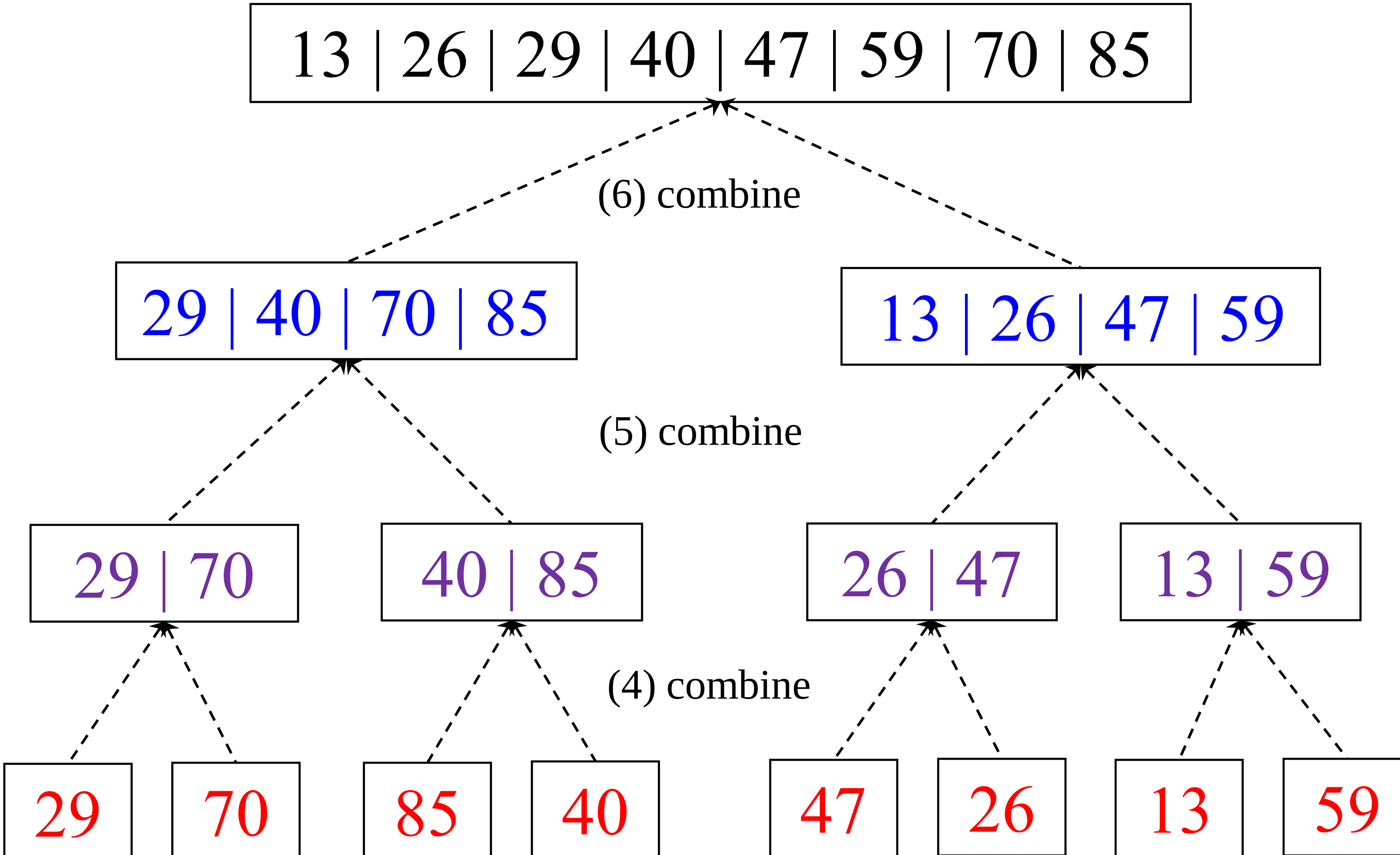
Conquer: sort two sub-arrays recursively

Combine: merge two sorted sub-arrays into a sorted array

Merge Sort: Divide Phase



Merge Sort: Combine Phase

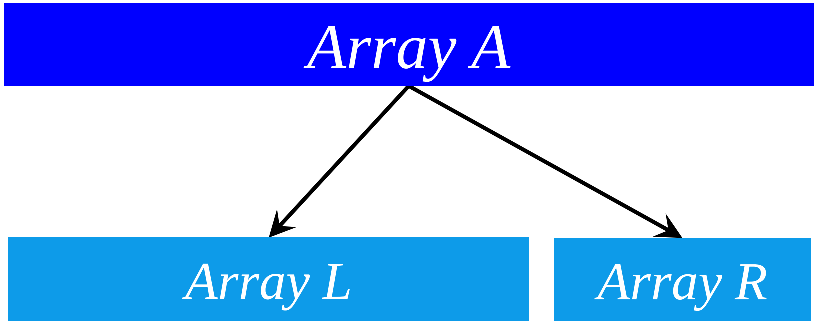


Quick Sort

- *Divide*: divide the array *A* into two sub-arrays *L* (smaller items) and *R* (larger items)
  - Note: *L* and *R* may have different sizes

- *Conquer*: sort two sub-arrays recursively

- *Combine*: no further work. Why?



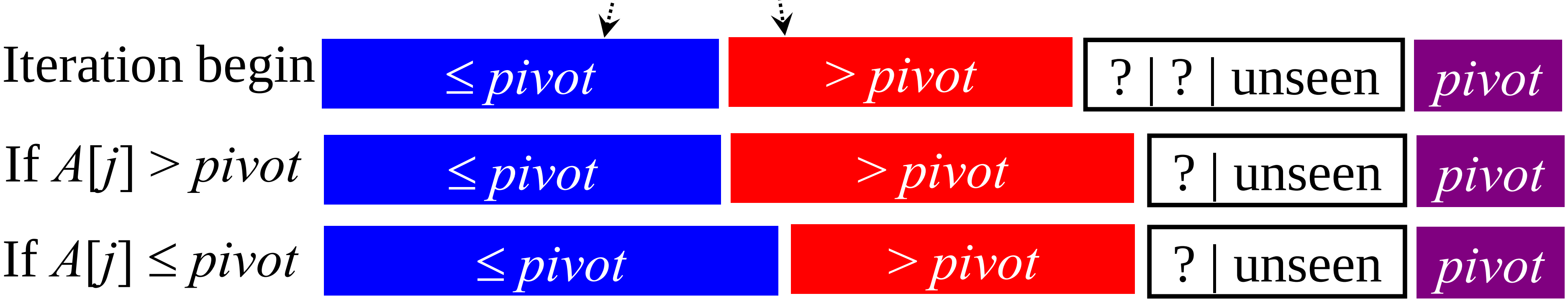
Quick Sort: Divide Phase

<u>Partition (Array $A[l..h]$)</u>

1. $pivot \leftarrow A[h]$
2. $i \leftarrow l - 1$
3. for $j \leftarrow l$ to $h - 1$
4.   if $A[j] \leq pivot$
5. $i \leftarrow i + 1$
6.   swap A[i] and A[j]
7. swap $A[i+1]$ and $A[h]$
8. return $i + 1$

- Position of sub-array A: from l to h
- Pick a **pivot** as the last item
- Sub-array $A[l..i]$: values $\leq pivot$
- Sub-array $A[i+1..j-1]$: values $> pivot$

- In each loop iteration, how do we maintain these conditions?
- What's clever about Line 6?



Quick Sort: Divide Phase

Execution order

29	70	85	40	47	26	13	59
29	70	85	40	47	26	13	59
29	70	85	40	47	26	13	59
29	40	85	70	47	26	13	59
29	40	47	70	85	26	13	59
29	40	47	26	85	70	13	59
29	40	47	26	13	70	85	59
29	40	47	26	13	59	85	70

Meaning of colours:

$\leq$ *pivot*

$>$ *pivot*

unseen

pivot

Quick Sort

First call: Quick-Sort ($A[0..n-1]$)

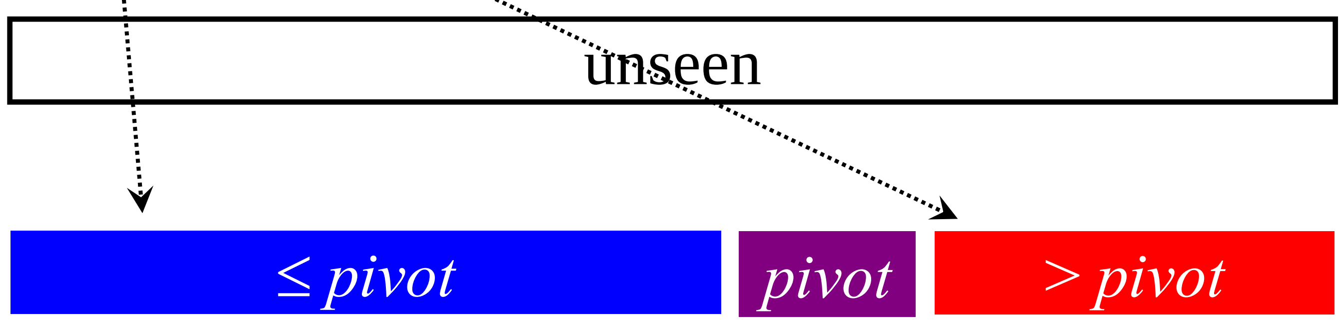
Quick-Sort (Array $A[l..h]$)

1. if $l < h$
2. $p \leftarrow$ Partition ($A[l..h]$)
3. Quick-Sort ($A[l..p-1]$)
4. Quick-Sort ($A[p+1..h]$)

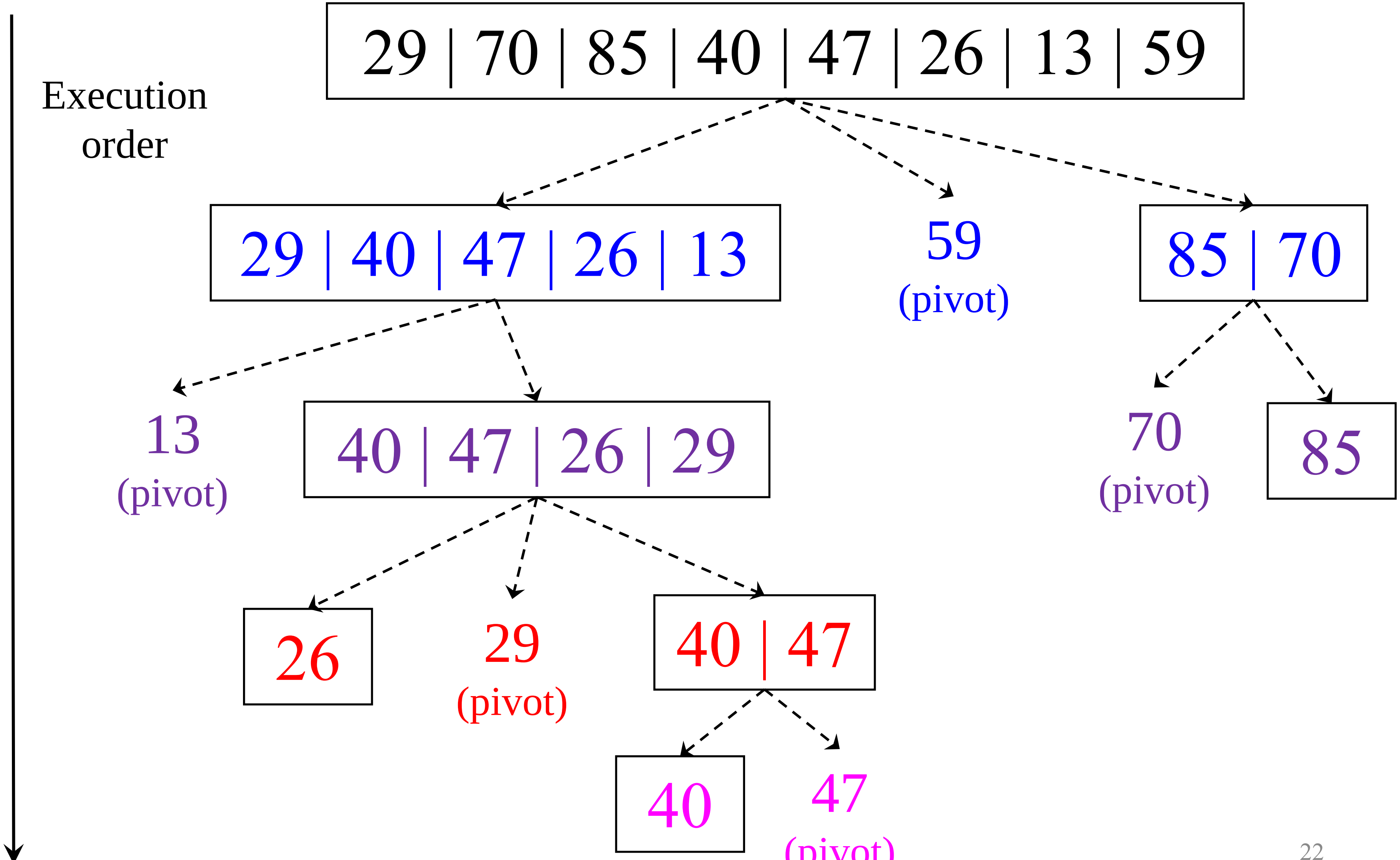
- *Divide*: Line 2
  - Partition the array by a pivot
  - Note that these two sub-arrays may have different sizes
- *Conquer*: Lines 3-4
  - Sort each sub-array recursively
- *Combine*: NIL

Before partition

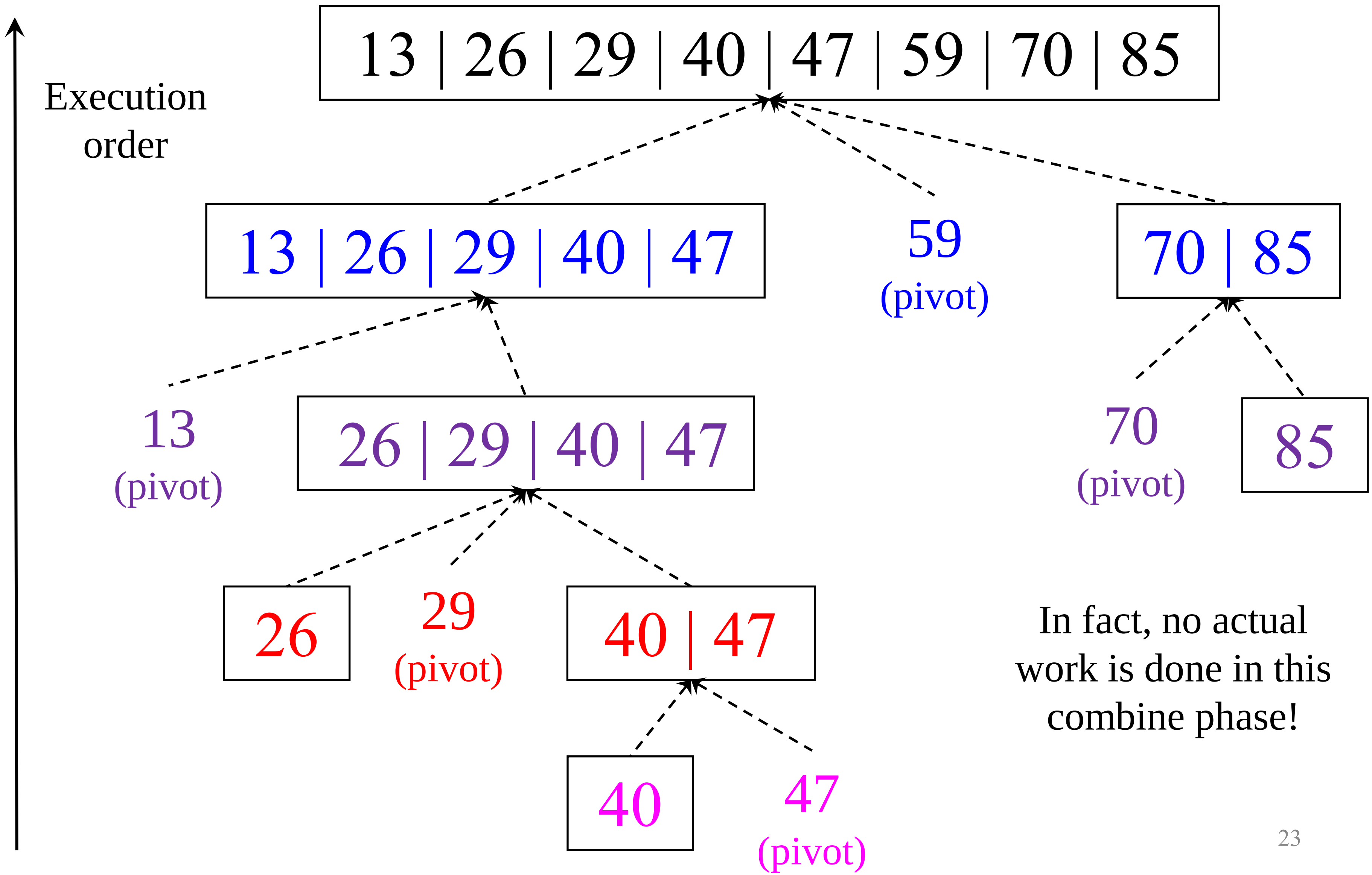
After partition



Quick Sort: Divide Phase



Quick Sort: Combine Phase



In fact, no actual work is done in this combine phase!

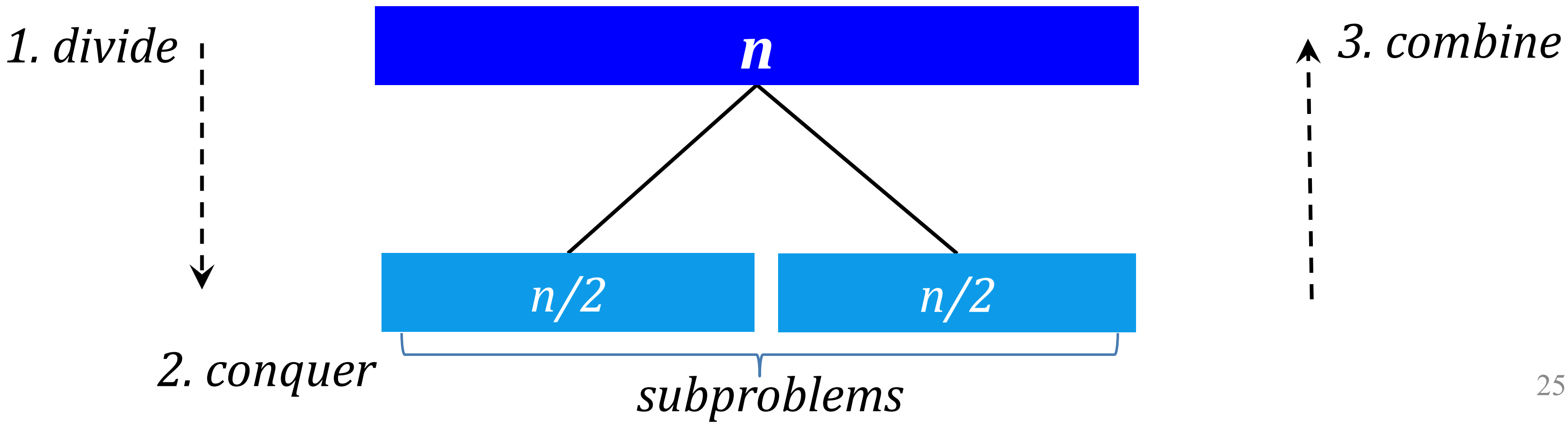
Our Roadmap

- Slow sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
- Fast sorting algorithms (by *divide-and-conquer*)
  - Merge sort
  - Quick sort
- How to *analyze* the running time of these divide-and-conquer algorithms?

Recurrence: running time of D&C Algorithm

- Let T(n) be the running time of algorithm at input size n
- *Example*: running time of Merge-Sort
  - Divide the problem into 2 subproblems
  - Each subproblem has 1/2 of the original problem size
  - The combine phase takes O(n) time (we will analyze this later)
- The **recurrence** of the running time of Merge-Sort

$$T(n) = 2\,T(n/2) + O(n)$$



Solving Recurrence

- [*Step 1*] Find the **recurrence** of the running time of Merge-Sort

$$T(n) = 2\,T(n/2) + O(n)$$

- [Step 2] After solving the recurrence, we get:

$$T(n) = O\,(n \log_2 n)$$

- We skip the details of solving the recurrence
  - Beyond the scope of this course

Running Time of D&C Algorithm

- To analyze the running time of a D&C algorithm, we do:
  - *Step 1*. Find the recurrence of the algorithm (*we now study*)
  - *Step 2*. Solve the recurrence

- We will analyze the running time of:
  - The Merge-sort algorithm
  - The Quick-sort algorithm

Running Time of **Merge** (for Merge Sort)

***Pre-condition**:*
Arrays L and R are already sorted

Merge (Array $L[0..n_L-1]$, Array $R[0..n_R-1]$)

1. $n_A \leftarrow n_L + n_R$
2. create a new array $A[0..n_A-1]$
3. $i \leftarrow 0$; $j \leftarrow 0$
4. for $k \leftarrow 0$ to n_A-1
5.   if ($j=n_R$) or ($i< n_L$ and $L[i] < R[j]$)
6. $A[k] \leftarrow L[i]$; $i \leftarrow i+1$
7.   else
8. $A[k] \leftarrow R[j]$; $j \leftarrow j+1$
9. return A

- The input size is $n_A = n_L + n_R$
- Lines 1, 3: O(1) time
- Lines 2, 4: $O(n_A)$ time
- Lines 5-8: same as Line 4
- Total time: $O(n_A)$

Running Time of Merge Sort

<u>Merge-Sort (Array $A[0..n-1]$)</u>

1. if $n > 0$
2. $m \leftarrow \lfloor n/2 \rfloor$
3. Array $L \leftarrow A[0..m-1]$
4. Array $R \leftarrow A[m..n-1]$
5. Merge-Sort (L)
6. Merge-Sort (R)
7. $A[0..n-1] \leftarrow$ Merge (L, R)

- Let $T(n)$ be the running time of Merge Sort
  - Lines 3, 4: $O(n)$ time
  - Line 5: $T(n/2)$ time
  - Line 6: $T(n/2)$ time
  - Line 7: $O(n)$ time
    - Running time of Merge
- Recurrence of Merge Sort

  $$T(n) = 2\,T(n/2) + O(n)$$

- After solving recurrence:

  $$T(n) = O(\, n \log n \,)$$

Running Time of **Partition** (for Quick Sort)

<u>Partition (Array $A[l..h]$)</u>

1. $pivot \leftarrow A[h]$
2. $i \leftarrow l - 1$
3. for $j \leftarrow l$ to $h - 1$
4.   if $A[j] \leq pivot$
5. $i \leftarrow i + 1$
6.   swap A[i] and A[j]
7. swap $A[i+1]$ and $A[h]$
8. return $i + 1$

- The input size n is $h - l + 1$
- Lines 1, 2: O(1) time
- Line 3: O(n) time
- Lines 4-6: same as Line 3
- Lines 7,8: O(1) time
- Total time: O(n) time

Running Time of **Quick Sort**

First call: Quick-Sort ($A[0..n{-}1]$)

<u>Quick-Sort (Array $A[l..h]$)</u>

1. if $l < h$
2. $p \leftarrow$ Partition ($A[l..h]$)
3. Quick-Sort ($A[l..p{-}1]$)
4. Quick-Sort ($A[p{+}1..h]$)

How do we solve this recurrence?

- The input size n is $h - l + 1$
- Let $T(n)$ be the running time of Quick Sort
  - Line 1: $O(1)$ time
  - Line 2: $O(n)$ time
  - Let x be the number of items in the left sub-array
  - Line 3: $T(x)$ time
  - Line 4: $T(n{-}x{-}1)$ time
- Recurrence of Quick Sort:

$$T(n) = T(x) + T(n{-}x{-}1) + O(n)$$

Quick Sort: Best Case Running Time

- Recurrence: $T(n) = T(x) + T(n-x-1) + c\,n$
  - where c is a constant
- The ***best case*** happens when both sub-arrays have the same size, i.e., $x = n-x-1 = (n-1)/2$

$$
\begin{aligned}
T(n) &= 2\,T(\,(n-1)/2\,) + c\,n \\
&\le 2\,T(\,n/2\,) + c\,n
\end{aligned}
$$

- Solving this recurrence, we get: $T(n) = O(n \log n)$
  as the best case running time of Quick-sort

Quick Sort: Worst Case Running Time

- Recurrence: $T(n) = T(x) + T(n\!-\!x\!-\!1) + c\,n$

- The ***worst case*** happens when the left (or the right) sub-array is the largest, i.e., $x = n\!-\!1$ and $n\!-\!x\!-\!1 = 0$

$$T(n) = T(n\!-\!1) + c\,n$$

Input size	Time
n	$c\,n$
$n-1$	$c\,(n-1)$
$n-2$	$c\,(n-2)$
…	…
1	$c\,(1)$

Thus, we derive: $T(n)$

$= c\,n + c\,(n-1) + c\,(n-2) + \ldots + c\,(1)$

$= c\,n\,(n+1)/2$

$= O(n^2)$

$\leq$ *pivot* | *pivot*

Quick Sort: Worst-Case Input

- *Quick Sort* takes O(n^2) time for the worst-case input:

- Is the following array the worst case input (at n=8)?
  - Run *Quick Sort*, find the pivot and partitions (i.e., subarrays) in each recursive call

13	26	29	40	47	59	70	85

$\leq$ *pivot* | *pivot*

Partition (Array $A[l..h]$)

```
pivot ← A[h]
i ← l – 1
for j ← l  to  h – 1
    if  A[j] ≤ pivot
        i ← i +1
        swap A[i] and A[j]
swap A[i+1] and A[h]
return i + 1
```

Randomized Quick Sort

- To *reduce the chance* of the worst case in Quick Sort, we can pick a random pivot (in the Divide phase)
  - See Lines 2-3 in the algorithm below

- Expected running time:

  O(n log n)

- Worst-case running time:

  still O(n^2)

but it occurs with very low chance

<u>Rand-Quick-Sort (Array $A[l..h]$)</u>

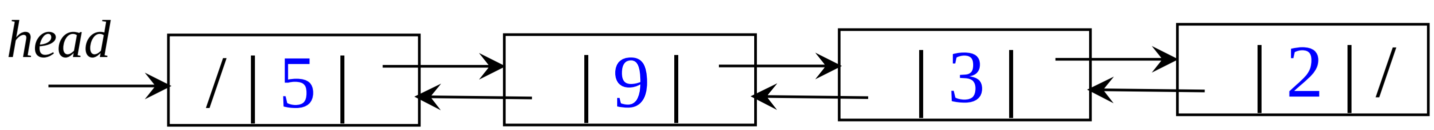
1. if $l < h$
2. **$r \leftarrow$ pick a random position in $l..h$**
3. **swap A[r] with A[h]**
4. $p \leftarrow$ Partition ($A[l..h]$)
5. Rand-Quick-Sort ($A[l..p-1]$)
6. Rand-Quick-Sort ($A[p+1..h]$)

Sorting on linked lists

- The sorting algorithms on arrays call the statement "swap $A[i]$ and $A[j]$"
  - This statement can be executed in O(1) time

5 | 9 | 3 | 2

- How to execute such statement on a (doubly) linked list?
  - Maintain the references to the i-th and the j-th element, e.g., i_{ref}, j_{ref}
  - Update the nodes of i_{ref}, j_{ref}, and their neighbors



Summary

- Selection sort, Bubble sort, Insertion sort
- Merge sort, Quick sort
- Worst-case input of a sorting algorithm
- Time complexity of a sorting algorithm
- Please read Chapter 12 in the book "*Data Structures and Algorithms in Java*", 6^{th} Edition