

Lecture 4

Sorting Algorithms

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

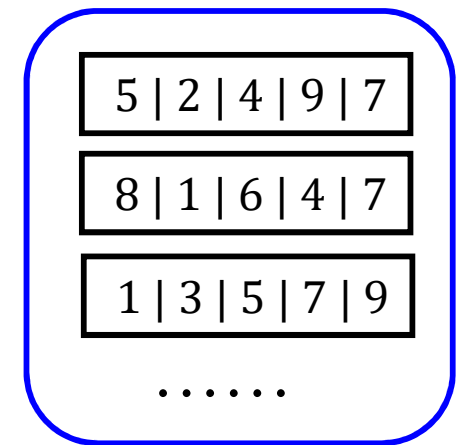
Output:

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

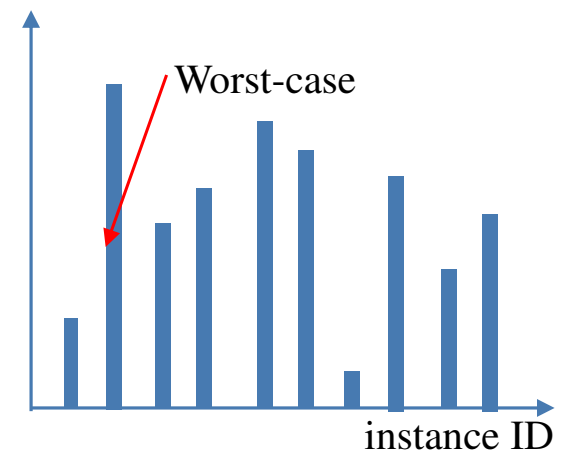
Time complexity of sorting algorithms

<i>Algorithm</i>	<i>Worst-case time complexity</i>	<i>Average-case time complexity</i>
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 $n=5$



Running time



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?
- ◆ Can we sort faster than $O(n \log n)$ in the worst case?

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 $i \leftarrow 0$ to $n-2$
2. $k \leftarrow i$
3. for integer $j \leftarrow i+1$ to $n-1$
4. if $A[k] > A[j]$ then
5. $k \leftarrow j$
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

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

1. repeat

2. $isUpdated \leftarrow \text{false}$

3. for integer $i \leftarrow 1$ to $n-1$

4. if $A[i-1] > A[i]$ then

5. swap $A[i-1]$ and $A[i]$

6. $isUpdated \leftarrow \text{true}$

7. until $isUpdated = \text{false}$

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

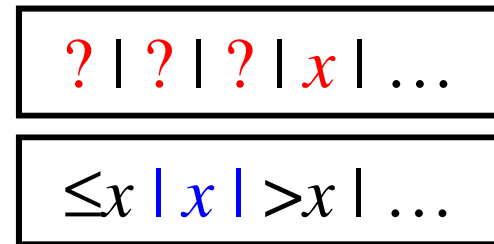
2 | 4 | 5 | 7 | 9

repeat iteration 2, no swap

2 | 4 | 5 | 7 | 9

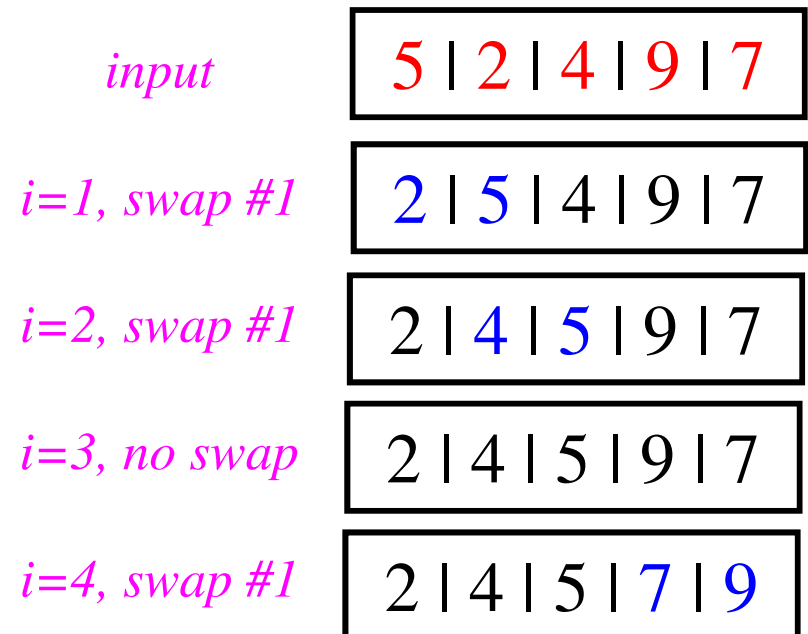
Insertion Sort

- ◆ 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 $> x$



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

1. for integer $i \leftarrow 1$ to $n-1$
2. $j \leftarrow i$
3. while $j > 0$ and $A[j-1] > A[j]$
4. swap $A[j]$ and $A[j-1]$
5. $j \leftarrow j-1$



Best-Case Input vs. Worst-Case Input

- ◆ In *comparison sorting* algorithms, the running time depends on the *number of element comparisons*
 - ◆ 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 number of element comparisons

=

- ◆ The worst-case input at $n=5$

9 | 7 | 5 | 4 | 2

- ◆ The number of element comparisons

=

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

1. repeat
2. $isUpdated \leftarrow \text{false}$
3. for integer $i \leftarrow 1$ to $n-1$
4. if $A[i-1] > A[i]$ then
5. swap $A[i-1]$ and $A[i]$
6. $isUpdated \leftarrow \text{true}$
7. until $isUpdated = \text{false}$

Our Roadmap

- ◆ Slow sorting algorithms



- ◆ Fast sorting algorithms (by *divide-and-conquer*)

 - ◆ Merge sort

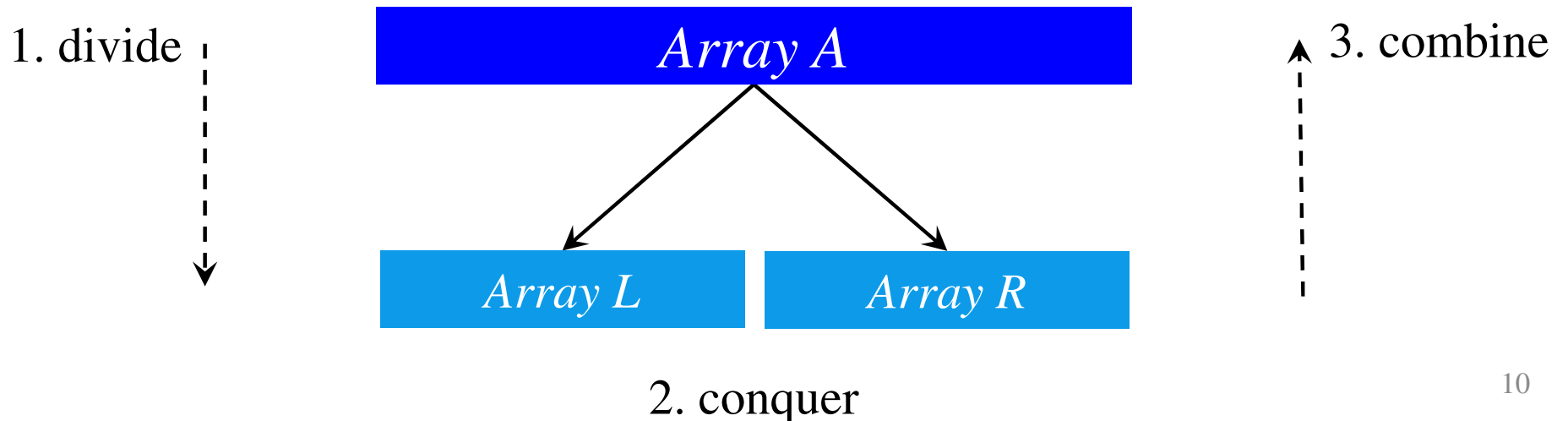
 - ◆ Quick sort

- ◆ How to *analyze* the running time of these divide-and-conquer algorithms?

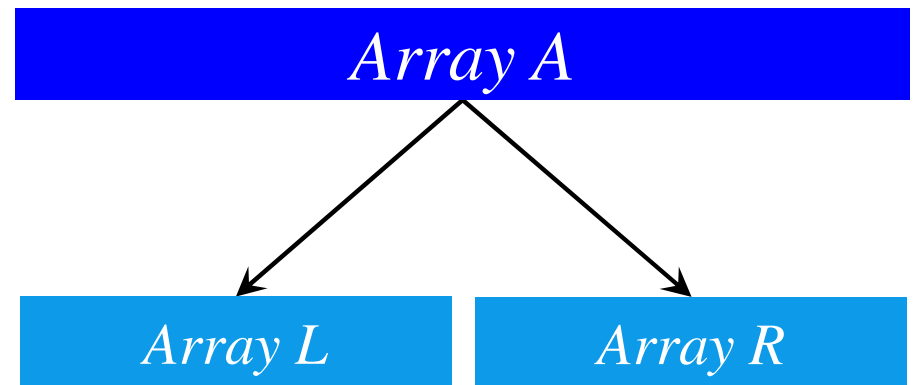
- ◆ Can we sort faster than $O(n \log n)$ in the worst case?

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



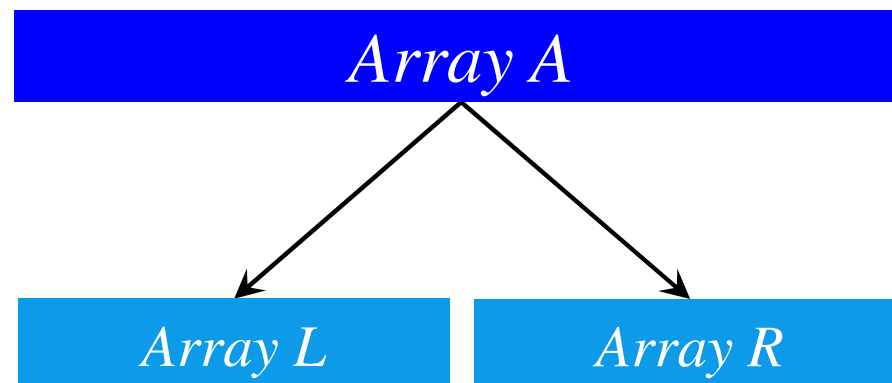
Divide-and-Conquer



<i>Algorithm</i>	Merge Sort	Quick Sort
<i>Divide</i>	Divide the array <i>A</i> into two equal-sized sub-arrays <i>L</i> and <i>R</i>	Partition the array <i>A</i> into two sub-arrays <i>L</i> (with small values) and <i>R</i> (with large values)
<i>Conquer</i>	Sort each sub-array recursively	Sort each sub-array recursively
<i>Combine</i>	Merge the sorted sub-arrays <i>L</i> and <i>R</i>	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 $R[0..n_R-1]$)

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 $(i < n_L \text{ and } L[i] < R[j])$
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

Pre-condition:

Arrays L and R are already sorted

L

29	40	70	85
----	----	----	----

R

13	26	47	59
----	----	----	----

A

--	--	--	--	--	--	--	--

Merge Sort

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

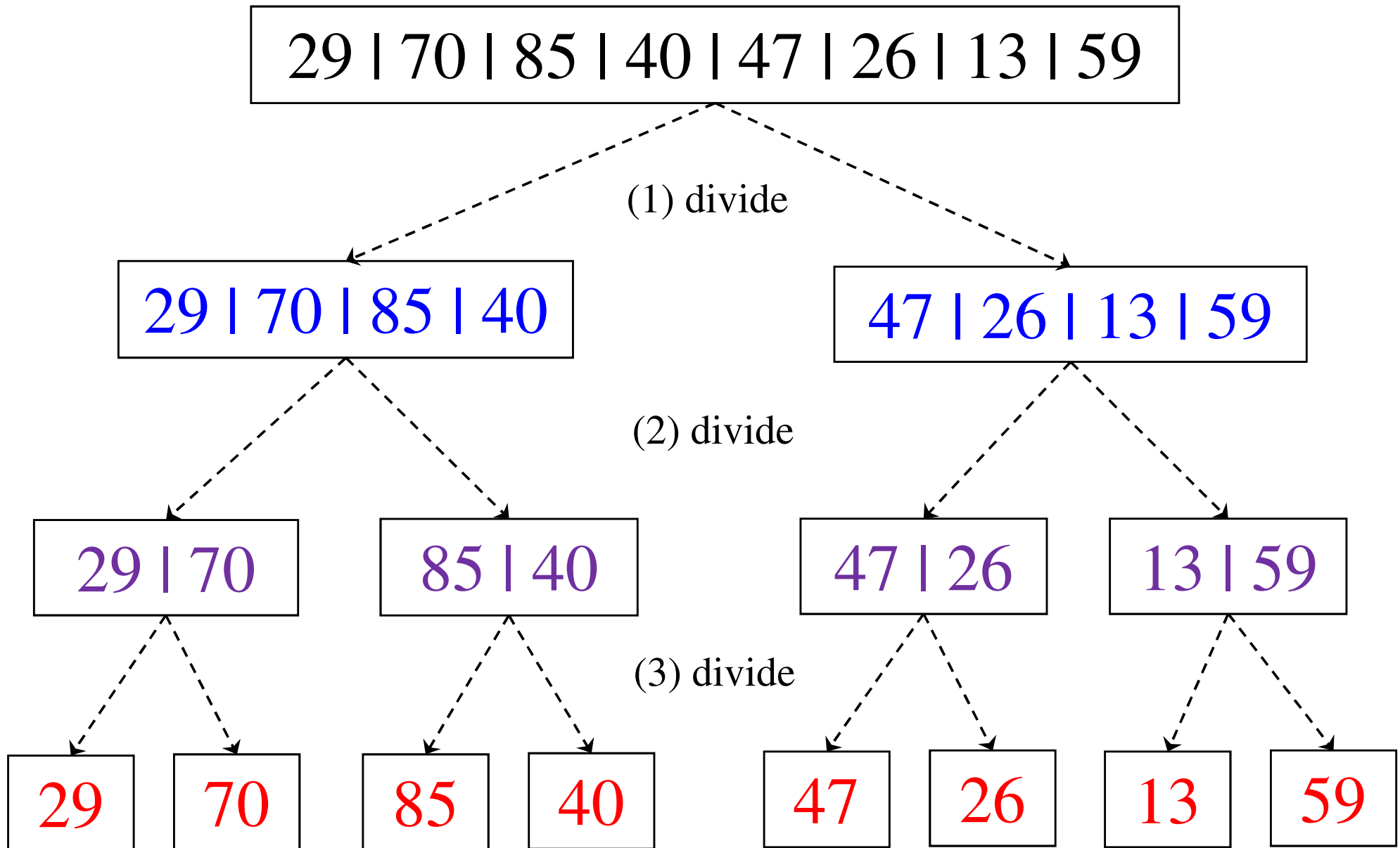
1. if $n > 1$
2. $m \leftarrow \lfloor n/2 \rfloor$
3. Array $L \leftarrow A[0..m-1]$
4. Array $R \leftarrow A[m..n-1]$
5. Merge-Sort (L)
6. Merge-Sort (R)
7. $A[0..n-1] \leftarrow \text{Merge} (L, R)$

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

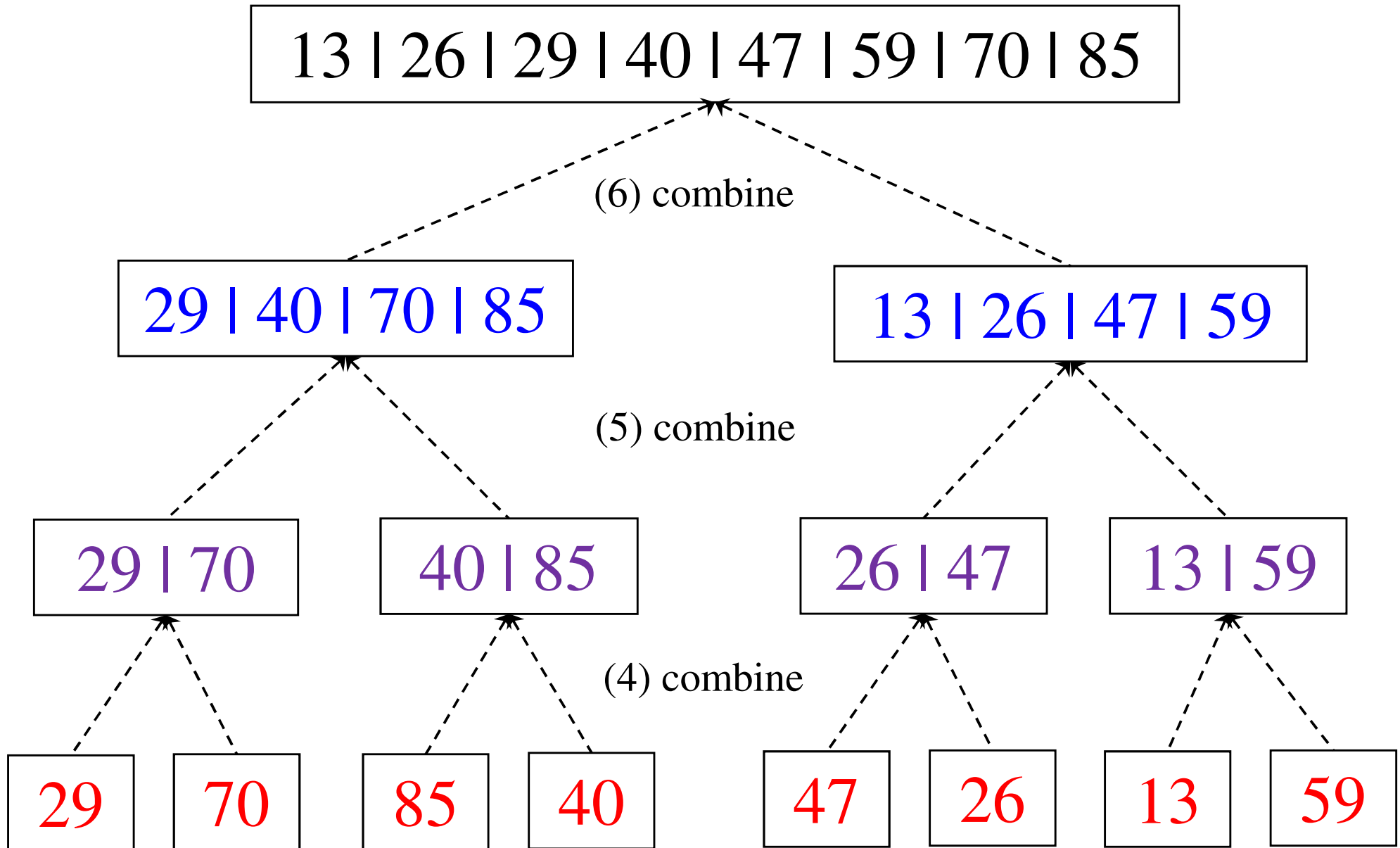
Conquer: sort two sub-arrays recursively

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

Merge Sort: Divide Phase

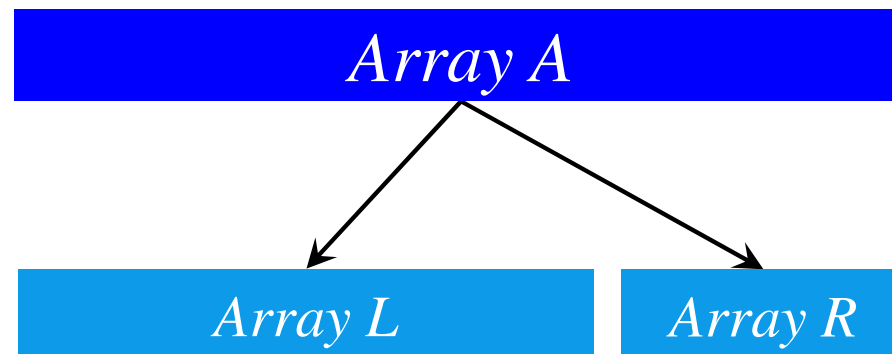


Merge Sort: Combine Phase



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

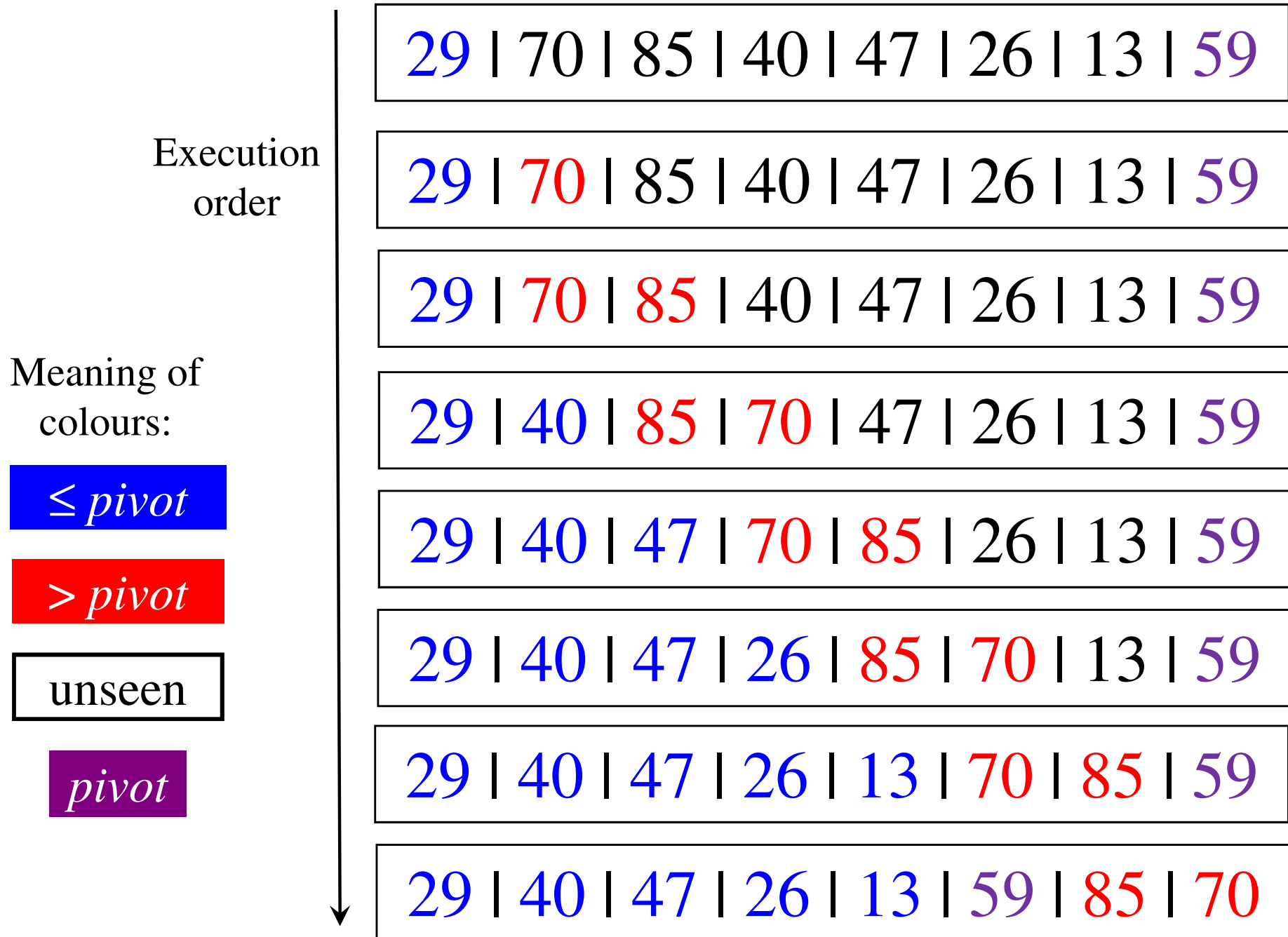
Partition (Array $A[l..h]$)

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?

Iteration begin	$\leq pivot$	$> pivot$	? ? unseen	<i>pivot</i>
If $A[j] > pivot$	$\leq pivot$	$> pivot$	? unseen	<i>pivot</i>
If $A[j] \leq pivot$	$\leq pivot$	$> pivot$	? unseen	<i>pivot</i>

Quick Sort: Divide Phase



Quick Sort

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

◆ *Divide:* Line 2

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

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

◆ 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

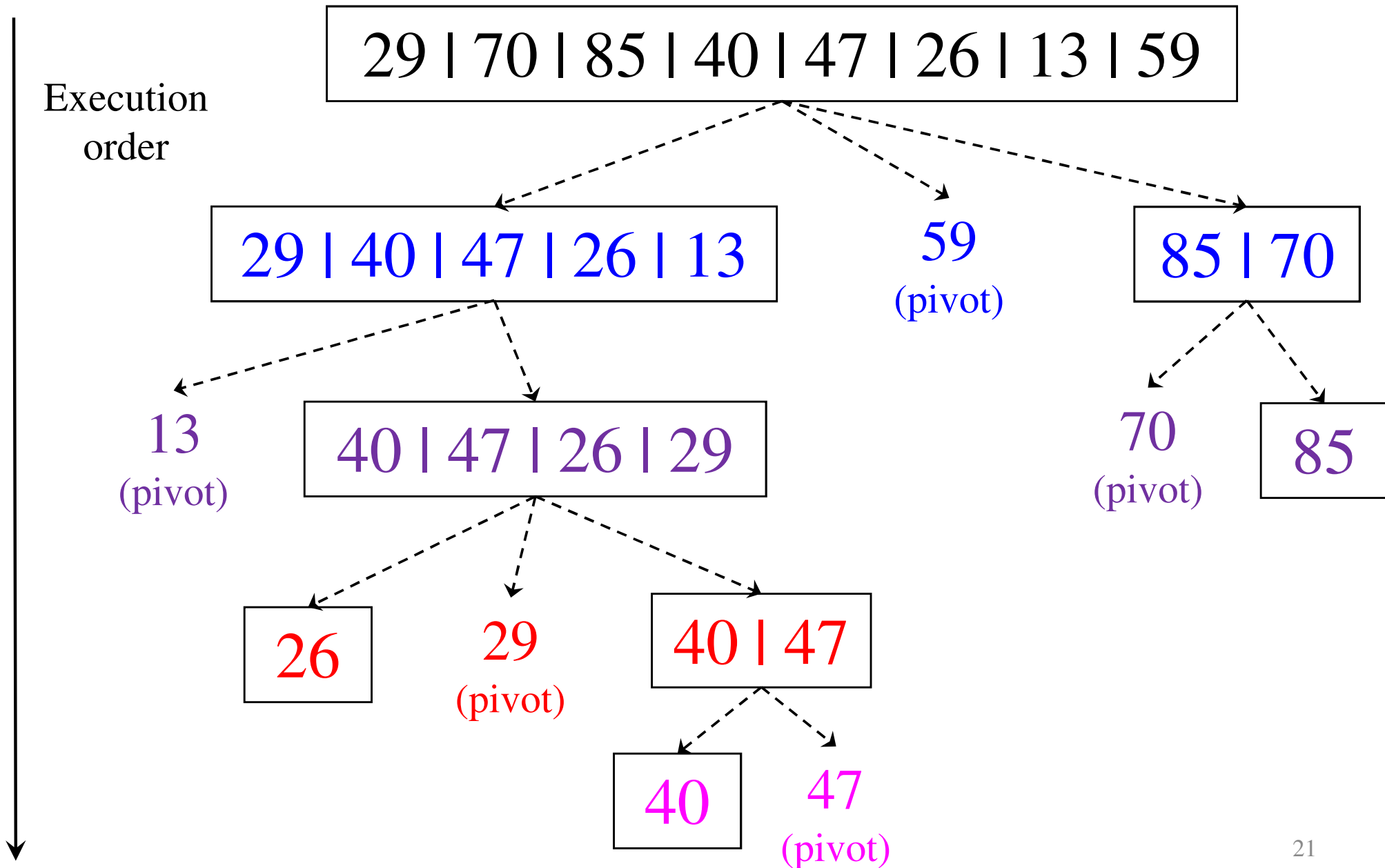
Before partition



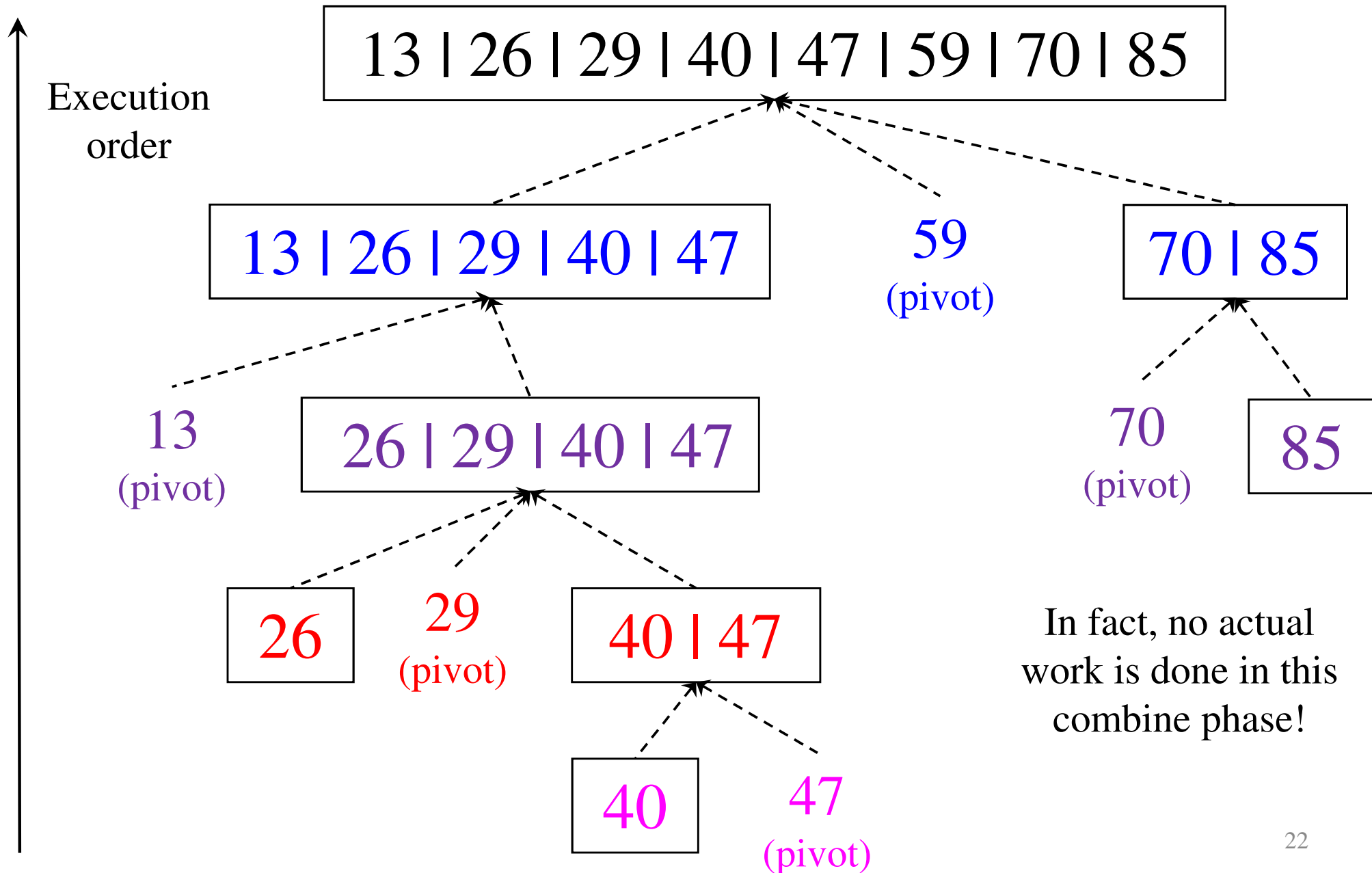
After partition



Quick Sort: Divide Phase



Quick Sort: Combine Phase



Our Roadmap

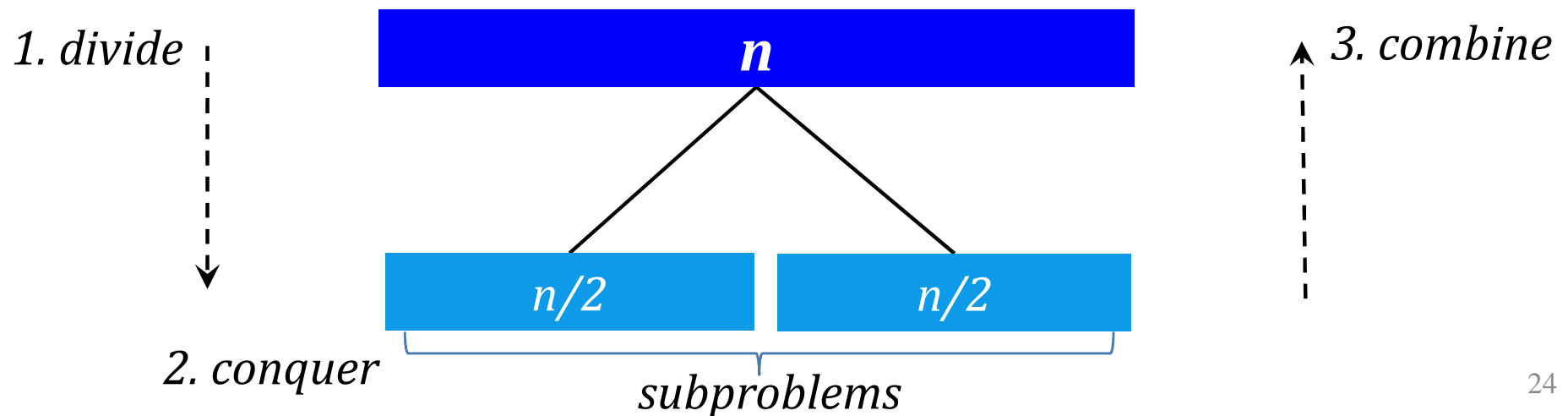
- ◆ Slow sorting algorithms
- ◆ Fast sorting algorithms (by *divide-and-conquer*)
 - ◆ Merge sort
 - ◆ Quick sort
- ◆ How to *analyze* the running time of these divide-and-conquer algorithms?
- ◆ Can we sort faster than $O(n \log n)$ in the worst case?



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:

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

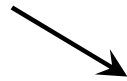
Running Time of D&C Algorithm

- ◆ To analyze the running time of a D&C algorithm, we do:
 - ◆ *Step 1.* Find the recurrence of the algorithm
 - ◆ *Step 2.* Solve the recurrence *[please refer to Lecture 3]*
- ◆ 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 $(i < n_L \text{ and } L[i] < R[j])$
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

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

1. if $n > 1$
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. $A[0..n-1] \leftarrow \text{Merge} (L, R)$

- ◆ 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)

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

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 ($A[0..n-1]$)

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

1. if $l < h$
2. $p \leftarrow \text{Partition} (A[l..h])$
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., $x = n-x-1 = (n-1)/2$
$$\begin{aligned} T(n) &= 2 T((n-1)/2) + c n \\ &\leq 2 T(n/2) + c n \end{aligned}$$
- ◆ 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 $n-x-1 = 0$

$$T(n) = T(n-1) + c n$$

Thus, we derive: $T(n)$

$$= c n + c (n-1) + c (n-2) + \dots + c (1)$$

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

$$= O(n^2)$$

<u>Input size</u>	<u>Time</u>
n	$c n$
$n-1$	$c (n-1)$
$n-2$	$c (n-2)$
$\dots$	$\dots$
1	$c (1)$

$\leq pivot$

$pivot$

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 $A[l..h]$)

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 $i + 1$

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:

$$\text{still } O(n^2)$$

but it occurs with very low chance

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

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 ($A[p+1..h]$)

Our Roadmap

- ◆ Slow sorting algorithms
- ◆ Fast sorting algorithms (by *divide-and-conquer*)
 - ◆ Merge sort
 - ◆ Quick sort
- ◆ How to *analyze* the running time of these divide-and-conquer algorithms?
- ◆ Can we sort faster than $O(n \log n)$ in the worst case?
 - ◆ NO/YES
 - ◆ Depending on the allowed operations, the domain size



Lower bound of comparison sorts

- ◆ An algorithm is *comparison sort algorithm* if
 - ◆ It determines the sorted order of an input array **only** by comparing elements
- ◆ Selection sort, Bubble sort, Insertion sort, Merge sort, Quick sort are comparison sort algorithms
- ◆ **[Theorem]** Any *comparison sort algorithm* requires $\Omega(n \log n)$ comparisons in the worst-case
 - ◆ Assuming all elements are distinct

Lower bound of comparison sorts

- ◆ To sort three elements a_1, a_2, a_3 , how many possible sorting orders?

- ◆ $3!$ (permutations)

- ◆ A binary decision tree

- ◆ We start from the root (the node at the top)
 - ◆ In a non-leaf node, “ $i:j$ ” means “compare a_i, a_j ”

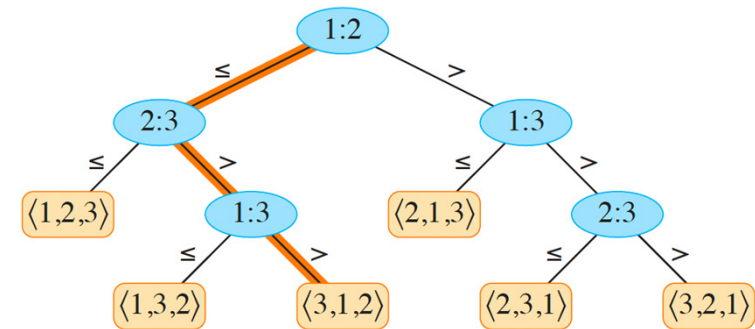
- two outcomes $\rightarrow$ two child nodes

- ◆ Each leaf node stores a sorted order

- ◆ A path from root to a leaf is a solution

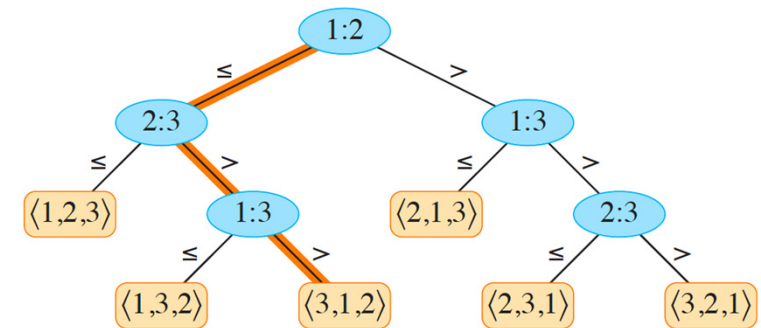
- ◆ The length of the path is the number of comparisons (i.e., the running time)

- ◆ The height h of the tree is the worst-case lower bound running time



Lower bound of comparison sorts

- ◆ To sort n elements $a_1, \dots, a_n$, how many possible sorting orders?
 - ◆ $n!$
- ◆ The height h of the tree is the worst-case lower bound running time



- ◆ Let's find a lower bound of h
- ◆ A complete binary tree with height h have at most 2^h leaf nodes
- ◆ $n! \leq 2^h$
- ◆ $h \geq \log_2 n!$
- ◆ Since $\log_2 n! = \Theta(n \log_2 n)$,

$$h = \Omega(n \log_2 n)$$

Sort with additional information

- ◆ Sort not only by comparison but also **additional information**
- ◆ *Example: Sort n elements that are integers in range 1 to k*
 - ◆ Counting sort runs in $\Theta(n + k)$ time, which is $\Theta(n)$, if $k = O(n)$

Sort with additional information

- ◆ Sort n elements *that are integers in range 1 to k*
 - ◆ How to sort this array of integers in $[1,4]$?

A

1	2	4	1	4	2	1
---	---	---	---	---	---	---

- ◆ Count the frequency of each integer

C

1	2	3	4
0	0	0	0

 $\longrightarrow$ C

1	2	3	4
3	2	0	2

- ◆ How to get the sorted array?
 - ◆ place the integers in A according to the frequencies in C

Sort n elements *that are integers in range 1 to k*

A

1	2	4	1	4	2	1
---	---	---	---	---	---	---

1 2 3 4

C

3	2	0	2
---	---	---	---

$i=1, l=0, m=3$; fill at positions 1,2,3

A

1	1	1	1	4	2	1
---	---	---	---	---	---	---

$i=2, l=3, m=2$; fill at positions 4,5

A

1	1	1	2	2	2	1
---	---	---	---	---	---	---

$i=3, l=5, m=0$; do nothing in A

A

1	1	1	2	2	2	1
---	---	---	---	---	---	---

$i=4, l=5, m=2$; fill at positions 6,7

A

1	1	1	2	2	4	4
---	---	---	---	---	---	---

Counting-Sort(A[1.. n], max value k)

1. create a new array C[1.. k] // frequency

2. for $i \leftarrow 1$ to k

3. C[i] $\leftarrow 0$

4. for $j \leftarrow 1$ to n

5. C[A[j]] $\leftarrow$ C[A[j]] + 1

6. $l \leftarrow 0$

7. for $i \leftarrow 1$ to k

8. $m \leftarrow$ C[i]

9. for $q \leftarrow 1$ to m

10. A[$l+q$] $\leftarrow i$

11. $l \leftarrow l+m$

Time complexity: $O(n+k)$

Summary

- ◆ Worst-case input of a sorting algorithm
- ◆ Divide-and-conquer sorting algorithms:
Merge sort and Quick sort
- ◆ Running time analysis of sorting algorithms
- ◆ Lower bound of comparison sort algorithms
- ◆ When can we sort faster than $O(n \log n)$?
- ◆ Please read Chapters 2.3, 7, 8.1, 8.2 in the textbook