Week 2
Recursive Algorithms
Sorting Algorithms

Algorithms and Data Structures

COMP3506/7505

Week 2 – Recursion & Sorting

1. Recursion
2. Analysis of recursive algorithms
3. Divide-and-conquer paradigm
4. Mergesort
5. Quicksort

Recursion Pattern

- Recursion: when a method calls itself
- Classic example – factorial function:
  - $n! = 1 \times 2 \times 3 \times 4 \times \cdots \times (n-1) \times n$
- Recursive definition:
  - $f(n) = \begin{cases} 1, \text{ if } n = 0 \\ n \cdot f(n-1), \ else \end{cases}$

Linear Recursion

- Test for **base cases**
  - Begin by testing for a set of base cases
  - Every possible chain of recursive calls *must* eventually reach a **base case**

Linear Recursion

- Perform a single recursive call

```python
def recursive_algo(n):
  if n <= 0:
    return 0
  elif n % 2 == 0:
    return 1 + recursive_algo(n/2)
  else:
    return 1 + recursive_algo(n-1)
```

Example: Reverse a List

reverse_list(A, i, j)

```
def reverse_list(my_list, i, j):
  if i < j:
    tmp = my_list[i]
    my_list[i] = my_list[j]
    my_list[j] = tmp
    reverse_list(my_list, i+1, j-1)
```

Defining Arguments for Recursion

- Recursive methods may require **additional parameters**

- We defined array reversal as reverse_list(A, i, j) not reverse_list(A)

Tail Recursion

- Recursive call as the **last step**
  - Result of the call must be used immediately and directly, or it is ***not*** a tail recursion

```
def reverse(S, start, stop):
  if start < stop-1:
    S[start], S[stop-1] = S[stop-1], S[start]
    reverse(S, start+1, stop-1)
```

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

Tail Recursion

- Recursive call as the **last step**
  - Result of the call must be used immediately and directly, or it is ***not*** a tail recursion
- Easily converted into iterative forms

```
def reverse(S, start, stop):
  if start < stop-1:
    S[start], S[stop-1] = S[stop-1], S[start]
    reverse(S, start+1, stop-1)
```

Tail Recursion

- Recursive call as the **last step**
  - Result of the call must be used immediately and directly, or it is ***not*** a tail recursion
- Easily converted into iterative forms

```
def reverse(S, start, stop):
  if start < stop-1:
    S[start], S[stop-1] = S[stop-1], S[start]
    reverse(S, start+1, stop-1)
```

```
def reverse_iterative(S):
  start, stop = 0, len(S)
  while start < stop-1:
    S[start], S[stop-1] = S[stop-1], S[start]
    start, stop = start+1, stop-1
```

Binary Recursion

- Two calls for each non-base case

```
def binary_sum(S, start, stop):
  if start >= stop:
    return 0
  elif start == stop-1:
    return S[start]
  else:
    mid = (start + stop) // 2
    return binary_sum(S, start, mid) + binary_sum(S, mid, stop)
```

Binary Recursion

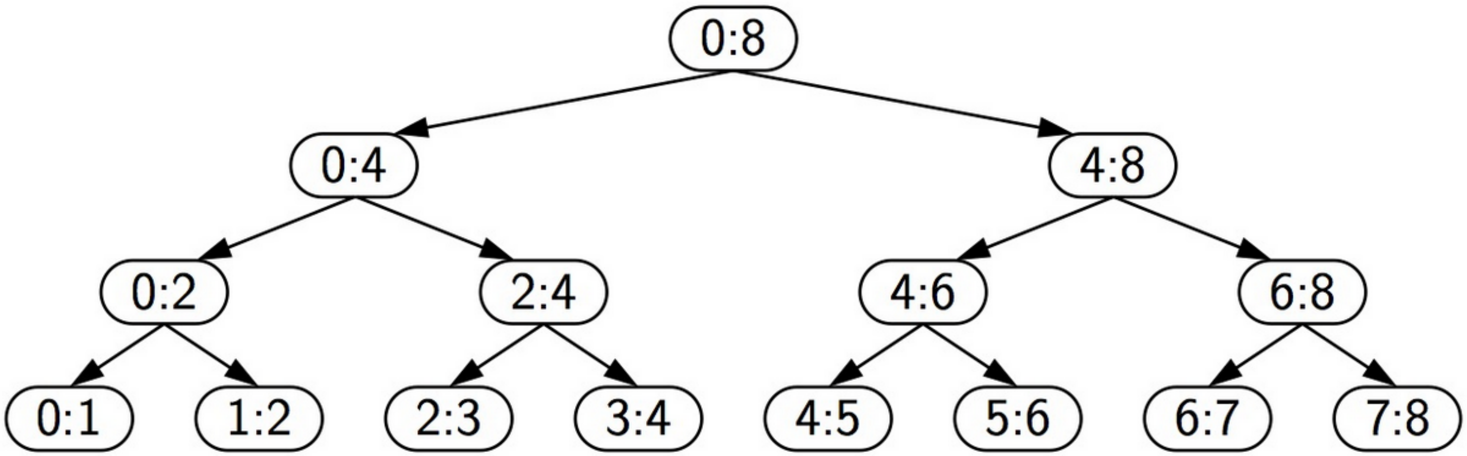


Figure 4.13: Recursion trace for the execution of binary_sum(0, 8).

Multiple Recursion

- Multiple recursion makes potentially many recursive calls

```
def directory_tree(path):
  if os.path.isdir(path): # is this a directory?
    for thing in os.listdir(path):
       childpath = os.path.join(path, thing)
       directory_tree(path) # note – this could be called many times
  else: # nope, we've bottomed out – let's just print this file/path
    print (path)
```

Recursion Activity

- Use recursion to design an algorithm that sorts an array of *n* integers
- We will call this ***selectionSort***

Base Case

- $n = 1$
  - Single-element input
  - Nothing to sort!

Recursive Case

- Scan each element of the array – find the largest (e_{max})

7	**5**	**3**	**5**	**7**	**8**	**6**	**5**	**1**	**2**

- Swap e_{max} with the last element of the array

7	**5**	**3**	**5**	**7**	**2**	**6**	**5**	**1**	**8**

- Repeat this process on the first n - 1 elements

7	**5**	**3**	**5**	**7**	**2**	**6**	**5**	**1**

Pseudocode

```
Algorithm selectionSort(A, n)
    if n > 1 then
        maxIndex ← 0
        for i ← 1 to n - 1 do
            if A[i] > A[maxIndex] then
                maxIndex ← i
        swap(A[maxIndex], A[n – 1])
        selectionSort(A, n - 1)
```

Week 2 – Recursion & Sorting

1. Recursion
2. Analysis of recursive algorithms
3. Divide-and-conquer paradigm
4. Mergesort
5. Quicksort

Running time of *selectionSort*

```
Algorithm selectionSort(A, n)
    if n > 1 then
        maxIndex ← 0                          } O(1)
        for i ← 1 to n - 1 do
            if A[i] > A[maxIndex] then        } O(n)
                maxIndex ← i
        swap(A[maxIndex], A[n – 1])           } O(1)
        selectionSort(A, n - 1)
```

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) + T(n-1) & \text{if } n > 1 \end{cases}$$

Running time of *selectionSort*

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

$= O(n) + O(n - 1) + T(n - 2)$

$= O(n) + O(n - 1) + O(n - 2) + T(n - 3)$

$\ldots$

n decreases by one each call, so there will be *n* recursive calls in total

$T(n) = 1 + 2 + 3 + \ldots + n - 1 + n$

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

$T(n)$ is $O(n^2)$

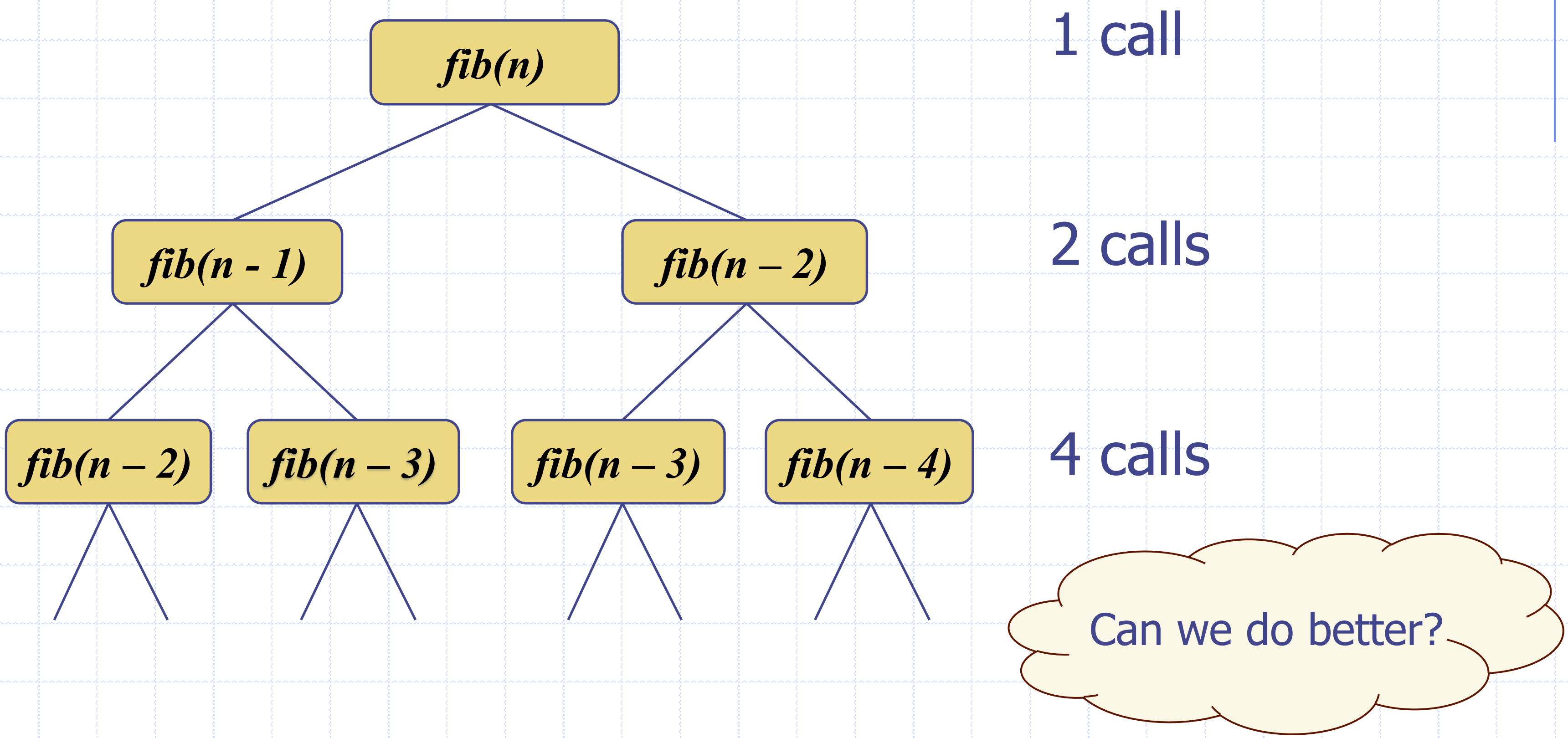
Fibonacci algorithm

```
Algorithm fib(n)
    for i
    if n < 2 then
        return 1
    return fib(n - 1) + fib(n - 2)
```

$$\left.\begin{array}{l}\text{for } i\\ \text{if } n < 2 \text{ then}\end{array}\right\} O(1)$$

$$T(n) = \begin{cases} O(1) & \text{if } n < 2 \\ T(n-1) + T(n-2) & \text{if } n \geq 2 \end{cases}$$

Analysis of Fibonacci algorithm



The number of calls doubles at each level in the recursion tree

Therefore the total number of calls will be less than or equal to 2^n

We can say this algorithm is $O(2^n)$

EXERCISE: Is this a tight bound? Why or why not?

Learn from my Mistakes

Question: Which of the following options characterizes the behavior of the given recurrence?

$$T = \begin{cases} 1, & n < 2 \\ T(n-1) + T(n-2), & n \geq 2 \end{cases}$$

- ○ $\Theta(n\ log_3\ n)$
- ○ $\Theta(n\ log\ n)$
- ○ $\Theta(n^3\ log\ n)$
- ○ $\Theta(3^n)$
- ○ $\Theta(n^2)$
- ◉ $\Theta(2^n)$

Week 2 – Recursion & Sorting

1. Recursion
2. Analysis of recursive algorithms
3. Divide-and-conquer paradigm
4. Mergesort
5. Quicksort

Search Algorithm

```
Algorithm search(A, n, k)
    Input: a sorted array A containing n elements
    Output: whether A contains k
    for i ← 1 to n do
        if A[i] = k then
            return true
    return false
```

- The running time: $O(n)$

Binary Search

- Compare the middle element, *m*, with *k*

2	4	5	7	7	8	9	10	14	15	20

  - If k < m, search the left half

2	4	5	7	7

  - If k > m, search the right half

9	10	14	15	20

  - If k = m, return true

Binary Search

```
Algorithm binarySearch(A, left, right, k)
    Input: a sorted array, A
    Output: whether k is in the subarray between A[left] and A[right]
    if right < left then
        return false
    m ← ⌊ (left + right) ÷ 2 ⌋
    if A[m] > k then
        return binarySearch(A, left, m – 1, k)
    else if A[m] < k then
        return binarySearch(A, m + 1, right, k)
    else
        return true
```

Analysis of Binary Search

- We can express the running time of *binarySearch* as

$$T(n) = \begin{cases} O(1) & \text{if } n \le 1 \\ T(n/2) + O(1) & \text{if } n > 1 \end{cases}$$

Analysis of Binary Search

- We can express the running time of *binarySearch* as

$$T(n) = \begin{cases} O(1) & \text{if } n \le 1 \\ T(n/2) + O(1) & \text{if } n > 1 \end{cases}$$

- The running time is $O(\lg n)$

Computing Powers

Algorithm *power*(x, n)
  Input: A number x and integer $n \geq 0$
  Output: The value x^n
  if $n = 0$ then
    return 1
  else
    $y \leftarrow x \cdot power(x, n-1)$
    return y

- ***Runtime*** complexity is $O(n)$

Computing Powers by Squaring

Algorithm *power*(x, n)

Input: A number x and integer $n \geq 0$

Output: The value x^n

```
if n = 0 then
    return 1
if n is odd then
    y ← power(x, (n - 1) ÷ 2)
    return x · y · y
else
    y ← power(x, n ÷ 2)
    return y · y
```

$$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$$
$$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$$
$$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$$
$$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128$$

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n \text{ is odd} \\ p(x,n/2)^2 & \text{if } n \text{ is even} \end{cases}$$

- ***Runtime*** complexity is $O(\lg n)$ ☺

Week 2 – Recursion & Sorting

1. Recursion
2. Analysis of recursive algorithms
3. Divide-and-conquer paradigm
4. Mergesort
5. Quicksort

Sorting Intuition

- Sorting is a fundamental operation…

- If you were asked to sort a list, how would you do it?

- Probably bubble sort, insertion sort, or selection sort… Let’s have a look.

Sorting Intuition

- In 2007, Barack Obama was at Google and the CEO at the time, Eric Schmidt, asked him *"what is the most efficient way to sort a million 32-bit integers?"*

- Obama's reply?

Sorting Intuition

- In 2007, Barack Obama was at Google and the CEO at the time, Eric Schmidt, asked him *"what is the most efficient way to sort a million 32-bit integers?"*
- Obama's reply? "*I think the bubble sort would be the wrong way to go"*
- And he's right! Why?

https://youtu.be/koMpGeZpu4Q

Sorting Intuition

- In 2... e and the... ed him *"wh... a mil...*

- Oba... *rt would be ...*
- And he’s right! Why?

https://youtu.be/koMpGeZpu4Q

Merge-Sort

- Sorting algorithm based on the divide-and-conquer paradigm
- **Guaranteed** $O(n \log n)$ running time
- This is **as good as it gets for worst case sorting**
  - Based on the "comparison" model
  - More on this next week

Merge-Sort

- Sort input sequence A with n elements
  - Divide: partition A into two halves
    Recur: recursively sort each half
  - Conquer: merge the two halves

Algorithm mergeSort(A, l, r)
  Input an array A
  Output A sorted between indices l and r

  if $l < r$
    $m \leftarrow \lfloor (l + r) \div 2 \rfloor$
    mergeSort(S, l, m)
    mergeSort(S, $m + 1$, r)
    merge(S, l, m, r)

Merging Two Sorted Sequences

- Conquer step Merging two sorted sequences, each with $n \div 2$ elements, takes $O(n)$ time

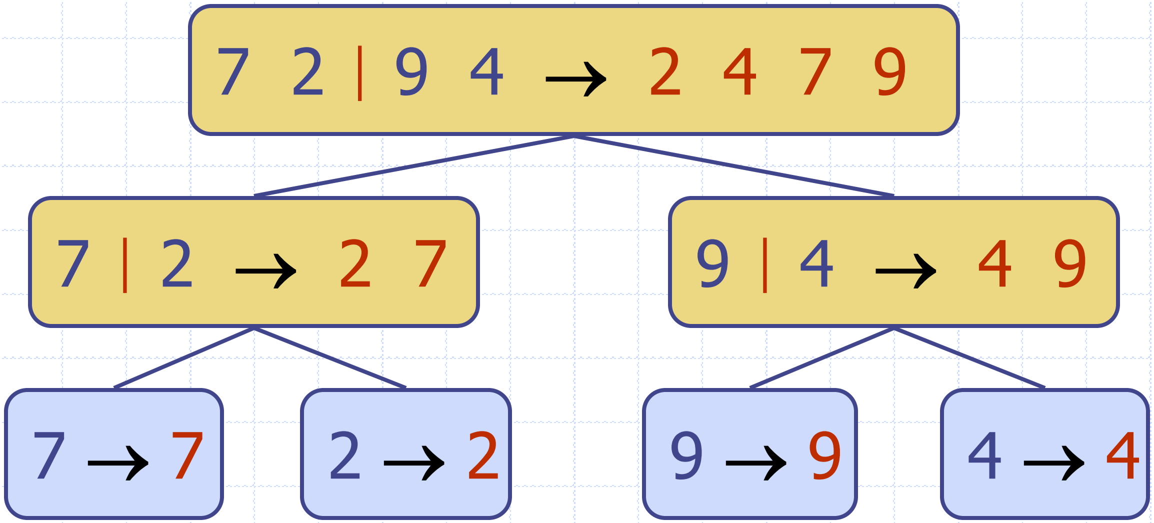
Algorithm ***merge***$(A, l, m\ r)$

Input an array A with two sorted halves
Output sorted union of $A[l..m]$ and $A[m..r]$

$n_1 \leftarrow m - l + 1$ // size of first half of A
$n_2 \leftarrow r - m$ // size of second half of A
$L \leftarrow$ copy of $A[l..m]$, $R \leftarrow$ copy of $A[m..r]$
$i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow l$
while $i < n_1$ and $j < n_2$ do // merge into A
  if $L[i] <= R[j]$ then
    $A[k{+}{+}] = L[i{+}{+}]$
  else
    $A[k{+}{+}] = R[j{+}{+}]$
while $i < n_1$ do // copy rest of L into A
  $A[k{+}{+}] = L[i{+}{+}]$
while $j < n_2$ do // copy rest of R into A
  $A[k{+}{+}] = R[j{+}{+}]$

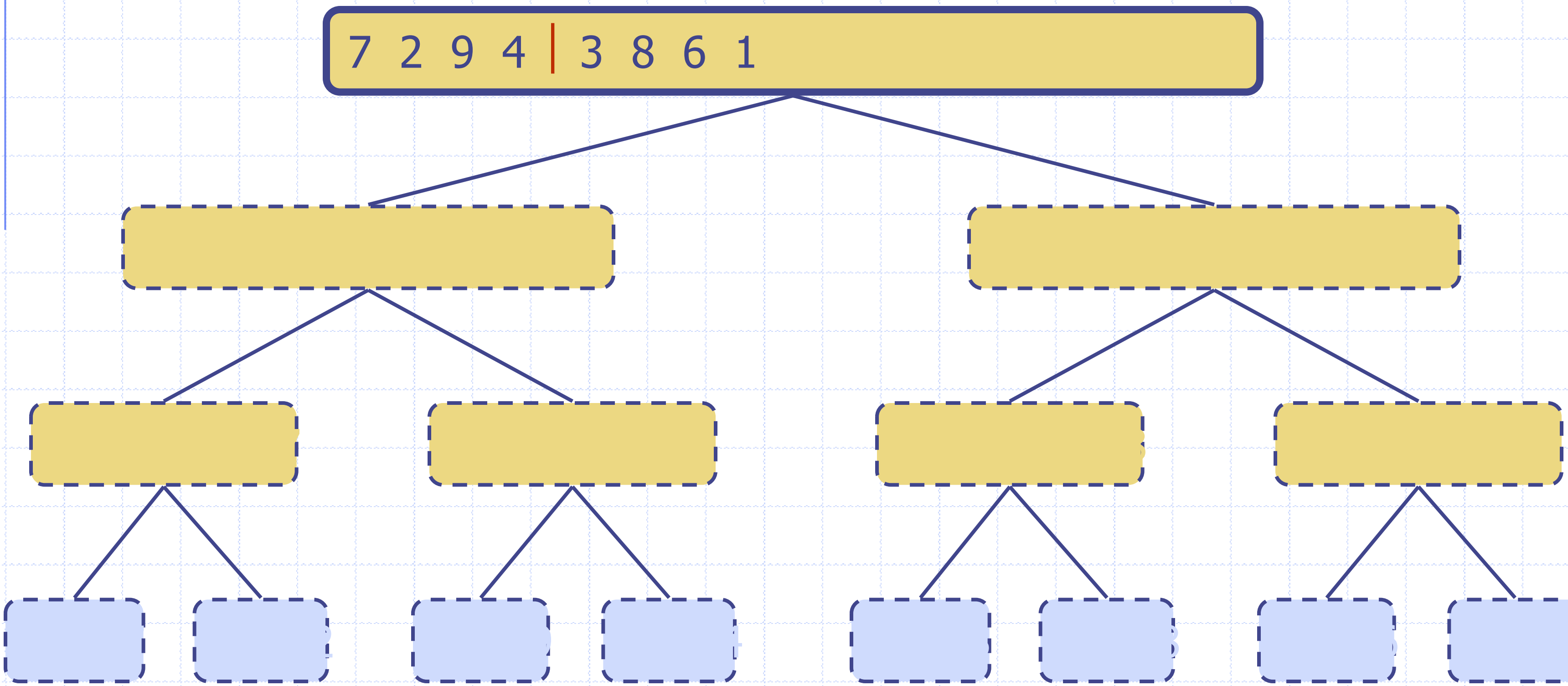
Merge-Sort Recursion Tree

- Execution of merge-sort is depicted by a recursion tree
  - each node represents a recursive call of merge-sort



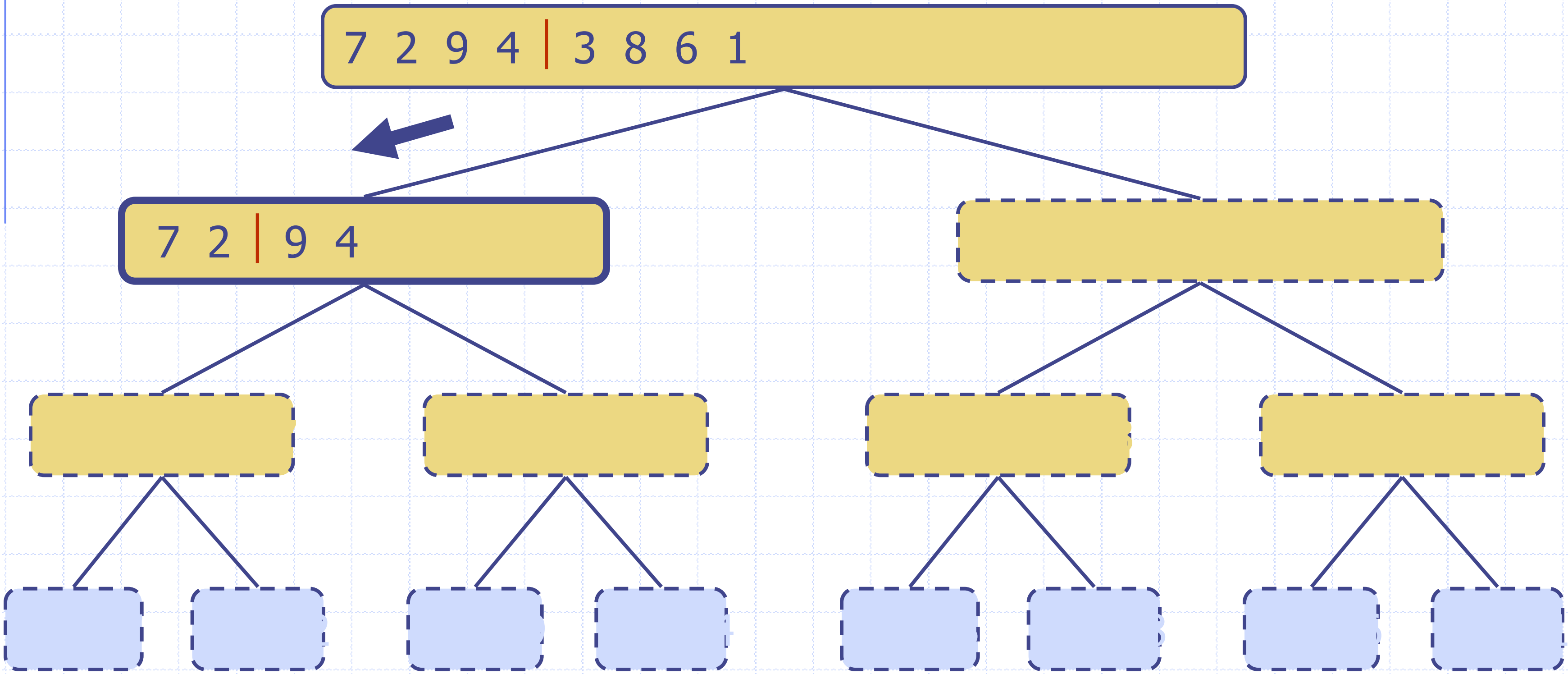
Execution Example

- Partition



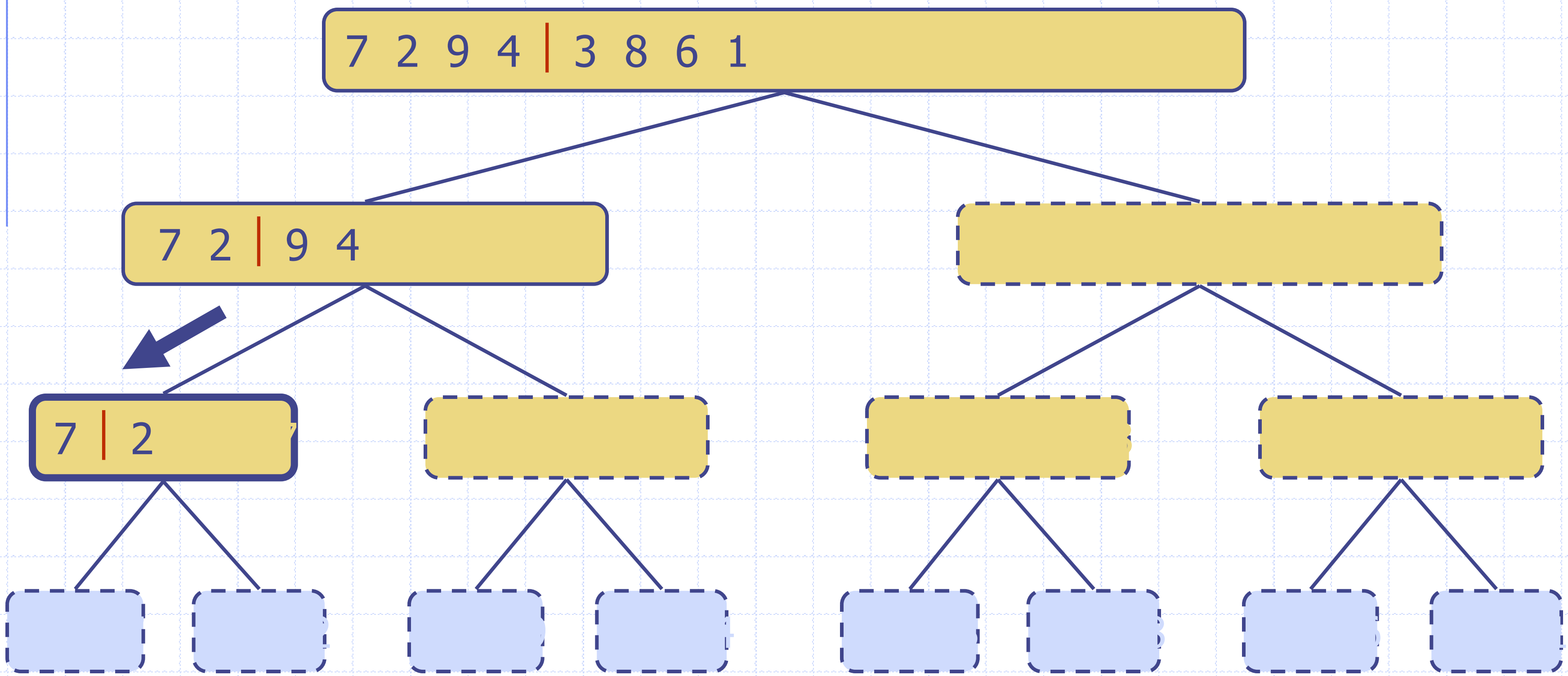
Execution Example (cont.)

- Recursive call, partition



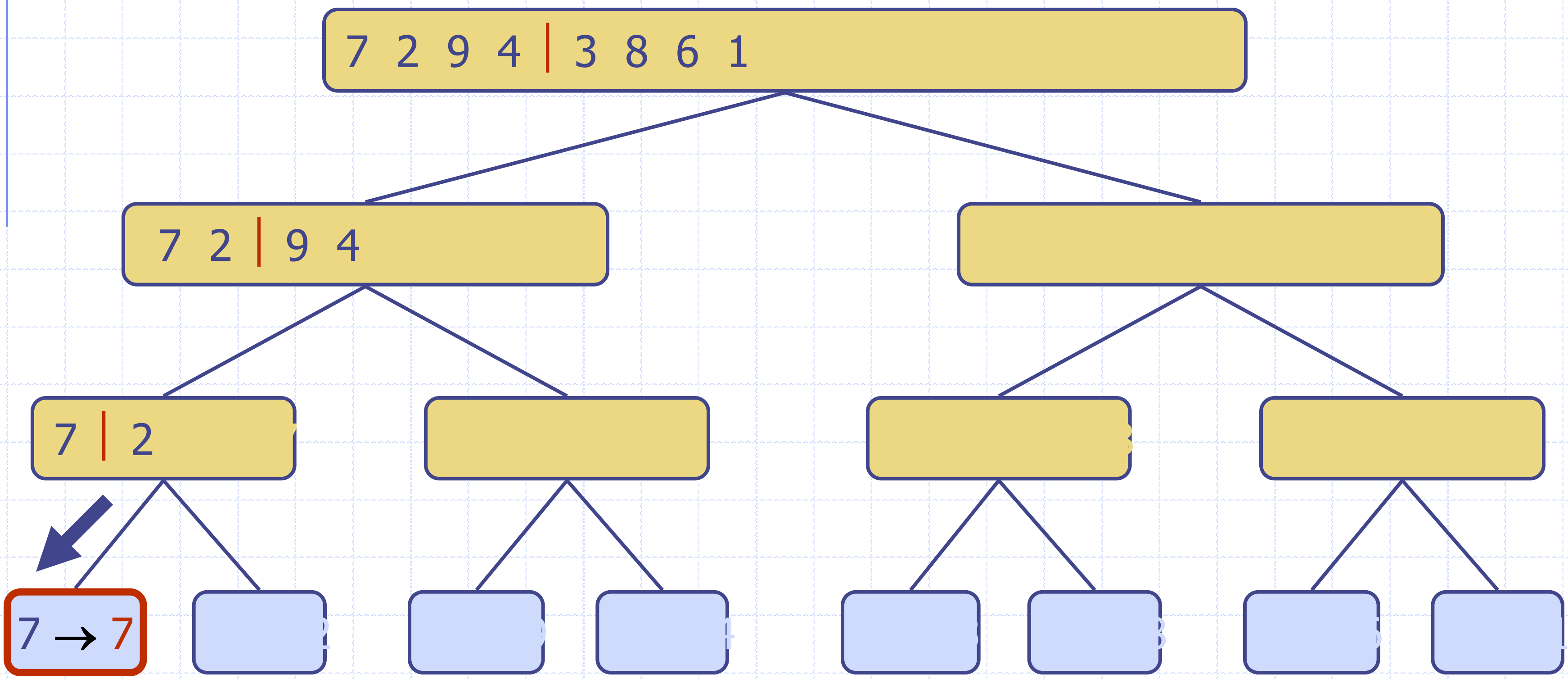
Execution Example (cont.)

- Recursive call, partition



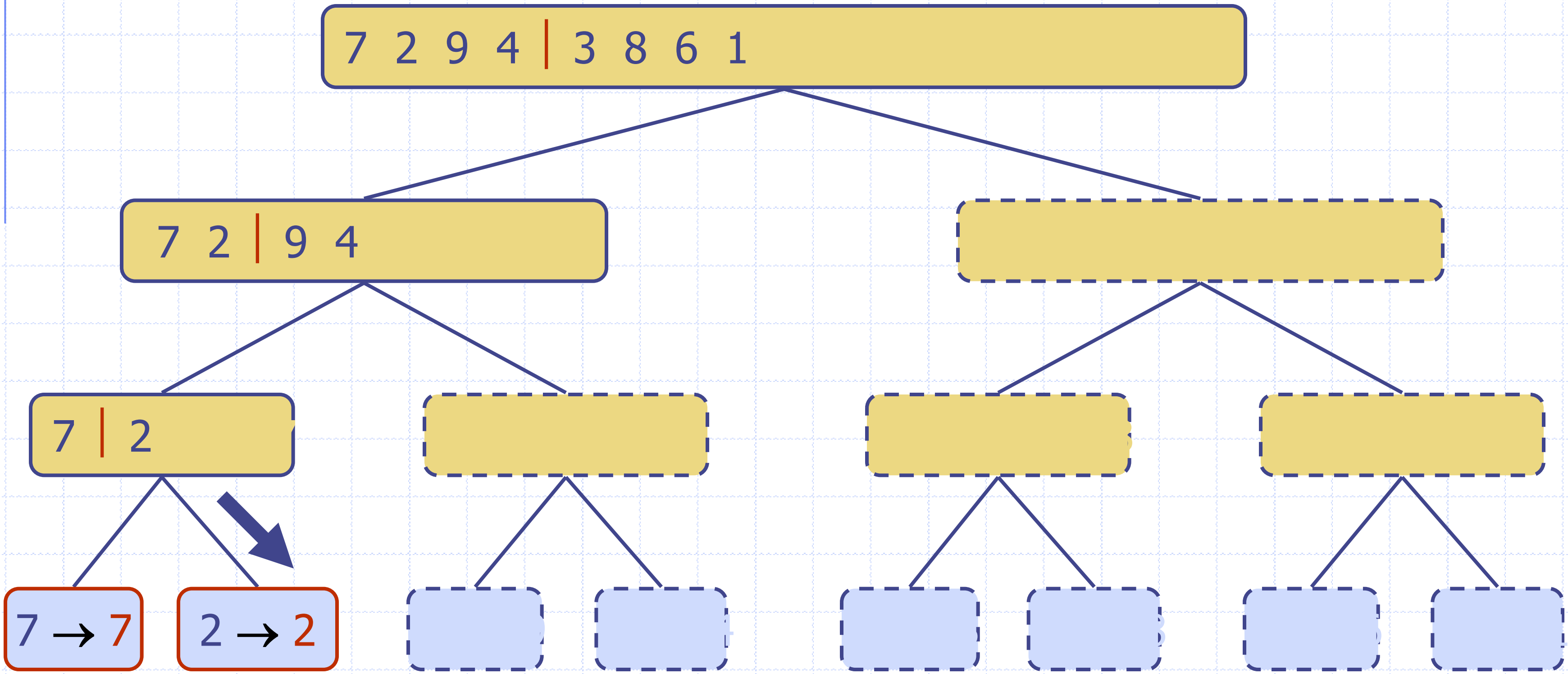
Execution Example (cont.)

- Recursive call, base case



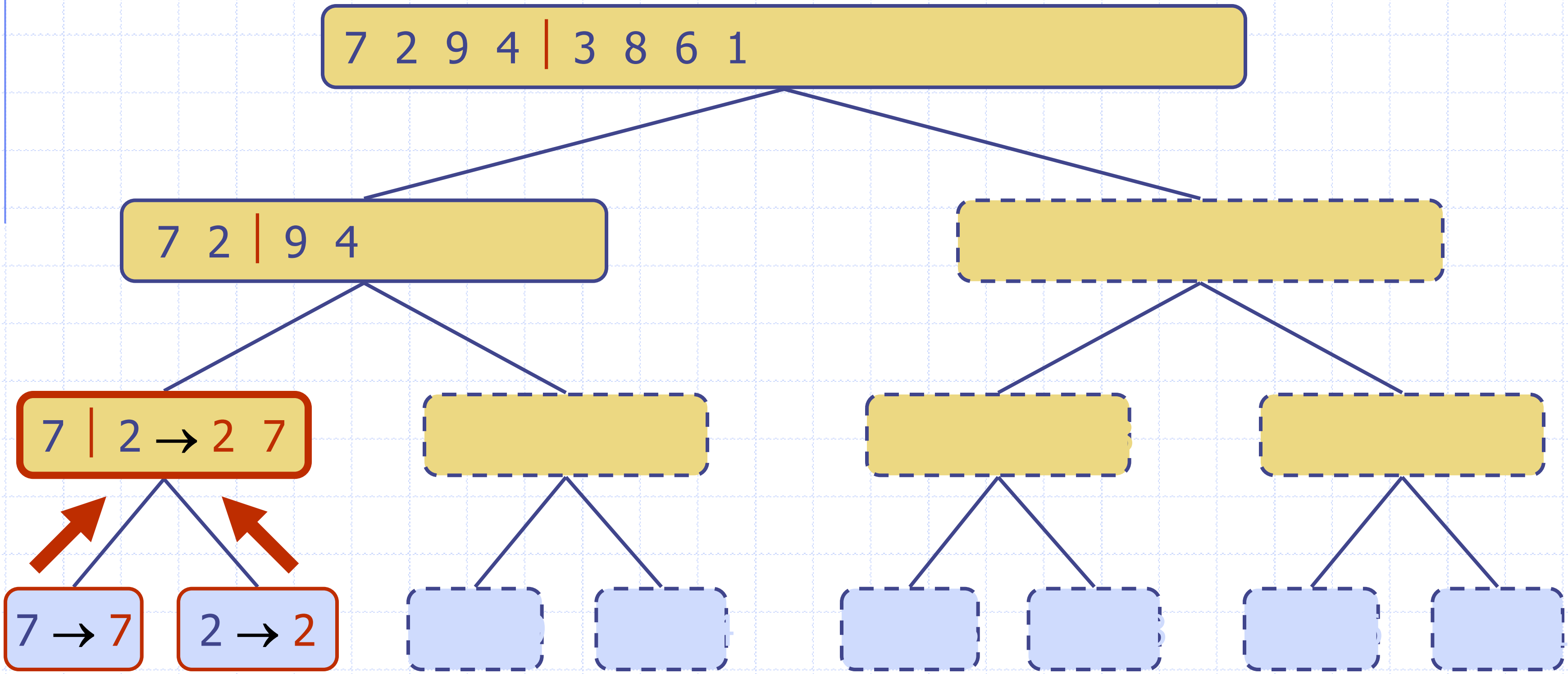
Execution Example (cont.)

- Recursive call, base case



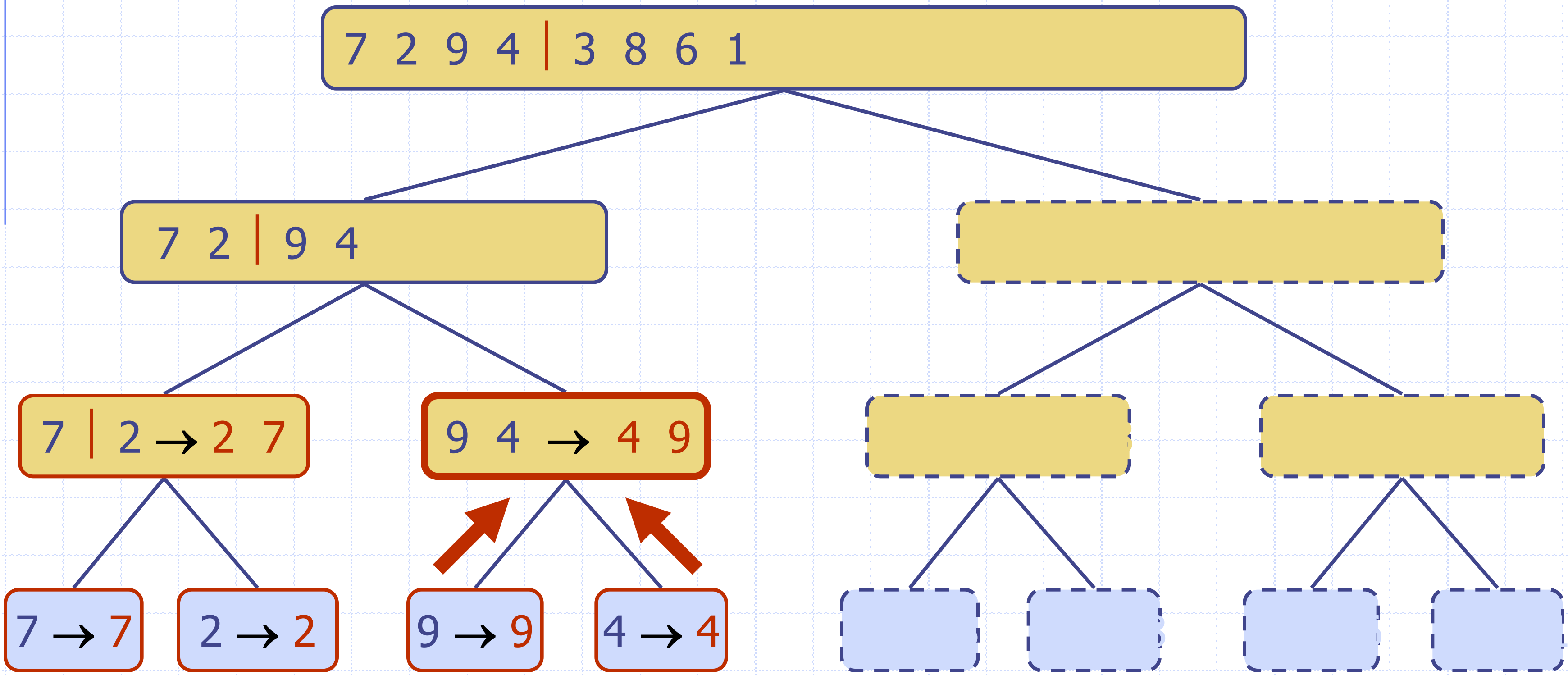
Execution Example (cont.)

- Merge



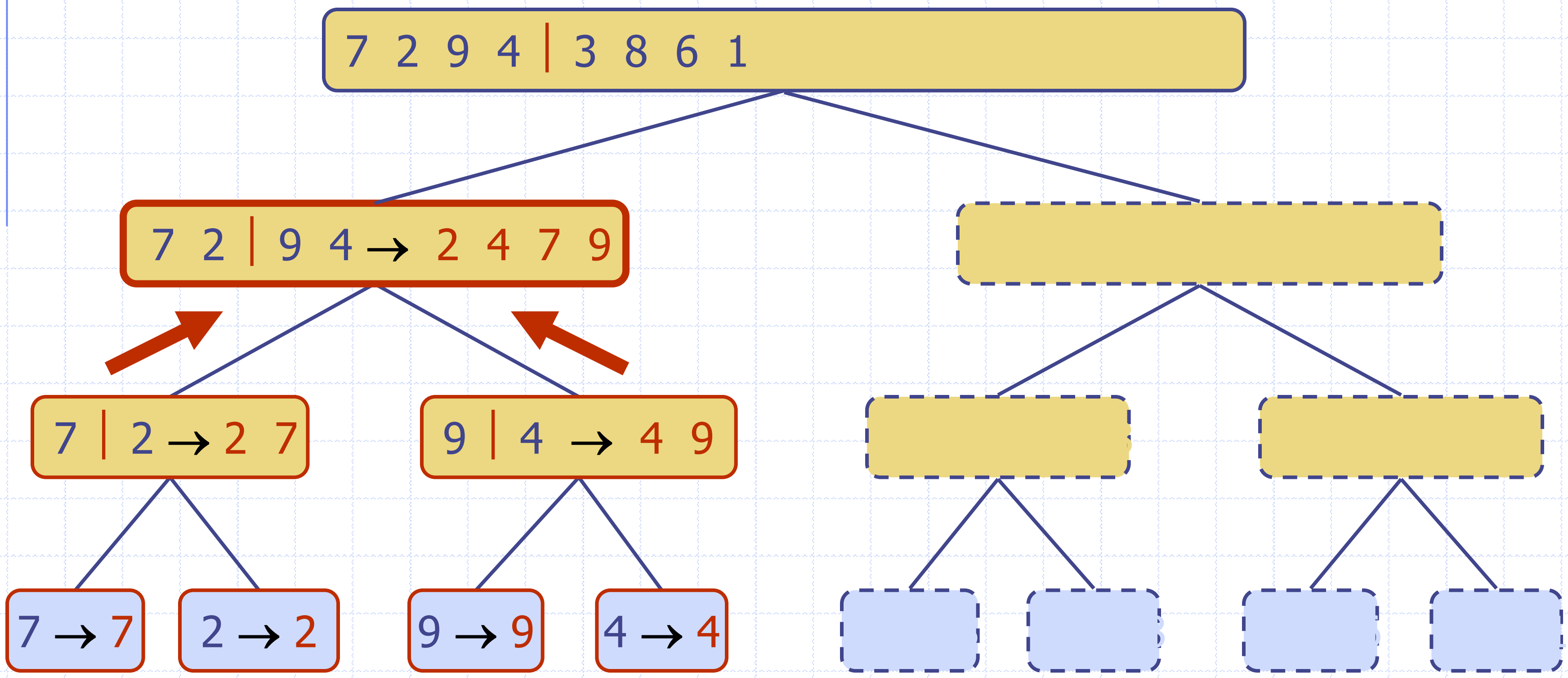
Execution Example (cont.)

- Recursive call, …, base case, merge



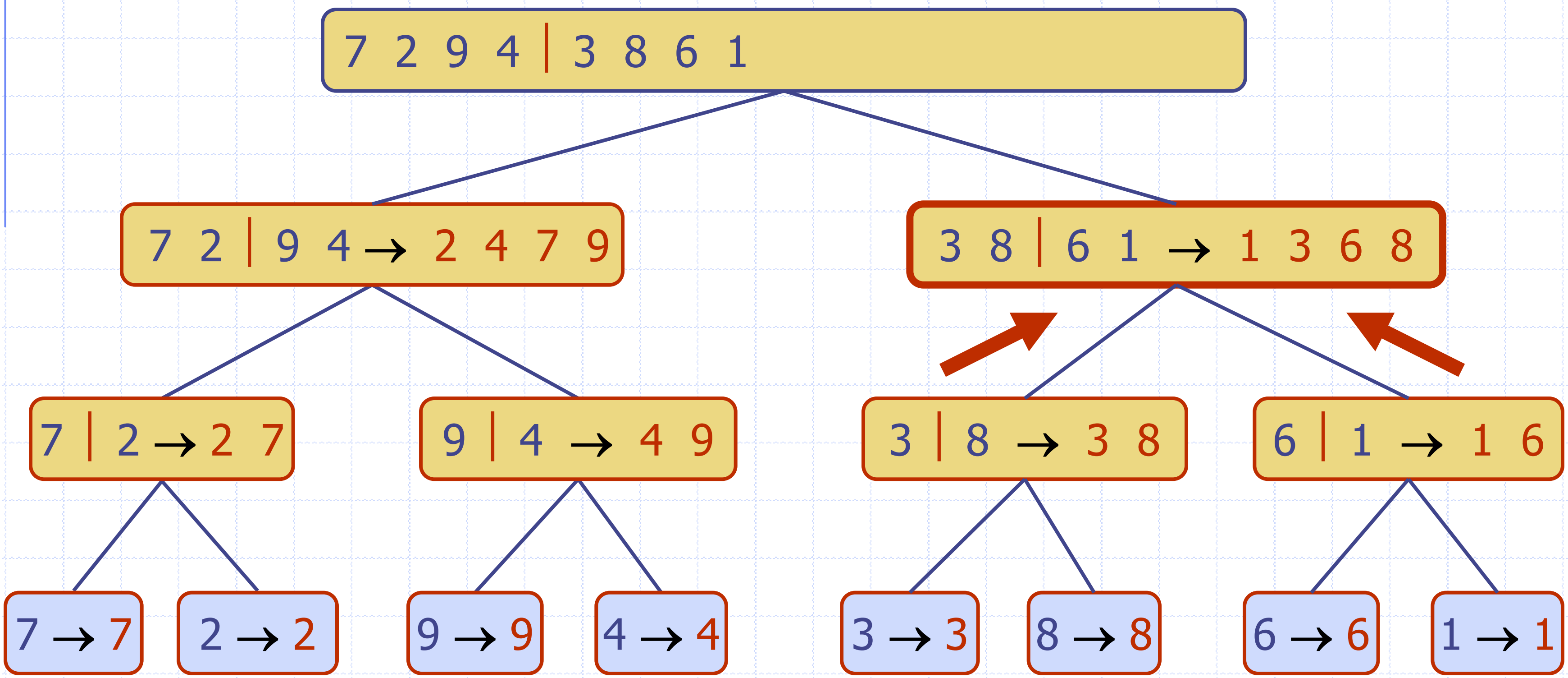
Execution Example (cont.)

- Merge



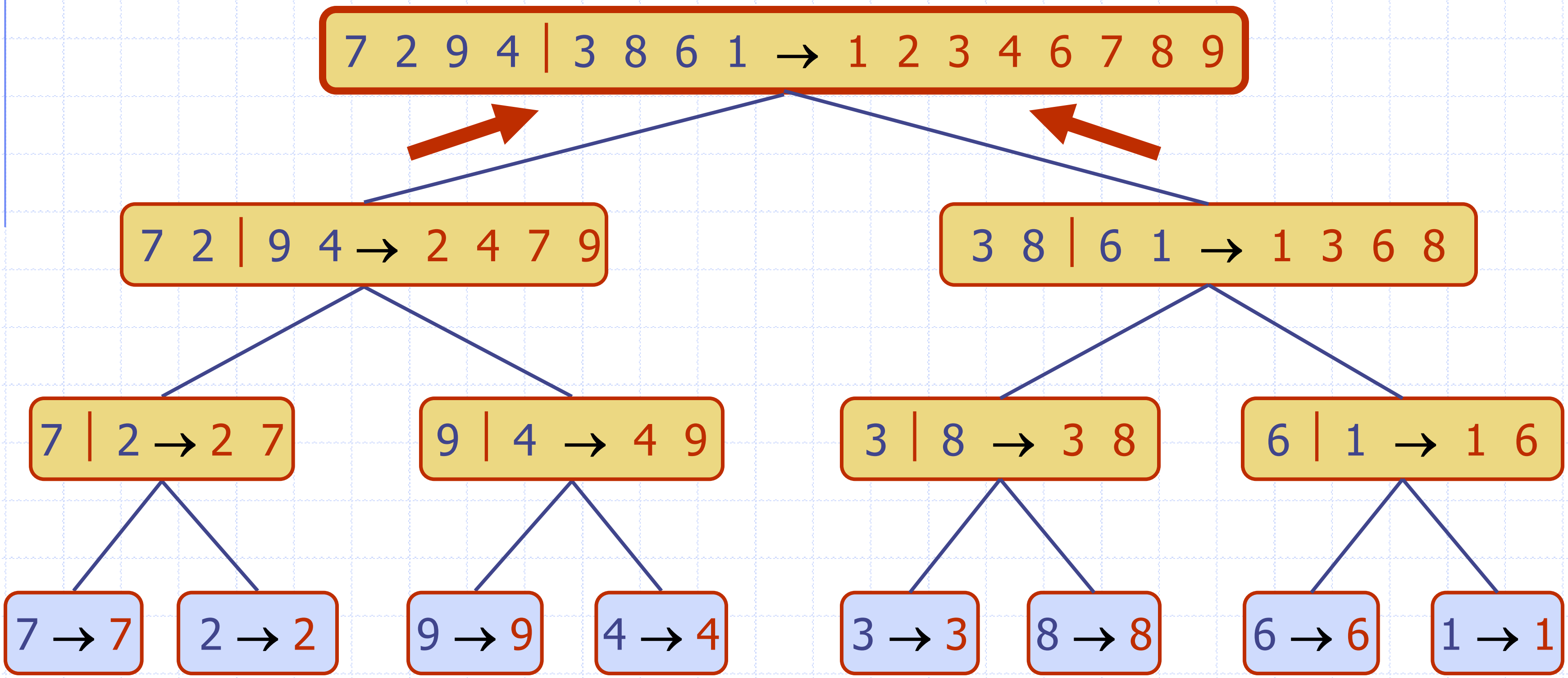
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge

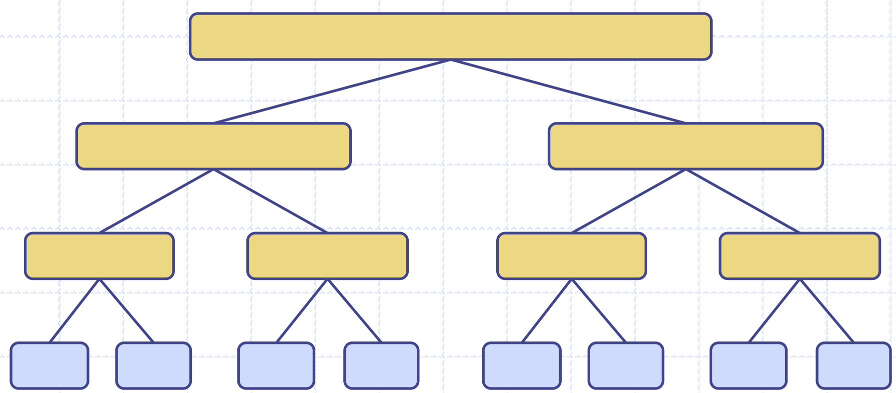


Analysis of Merge-Sort

- We can write the running time of merge sort as

$$T(n) = \begin{cases} O(1) & \text{if } n < 2 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

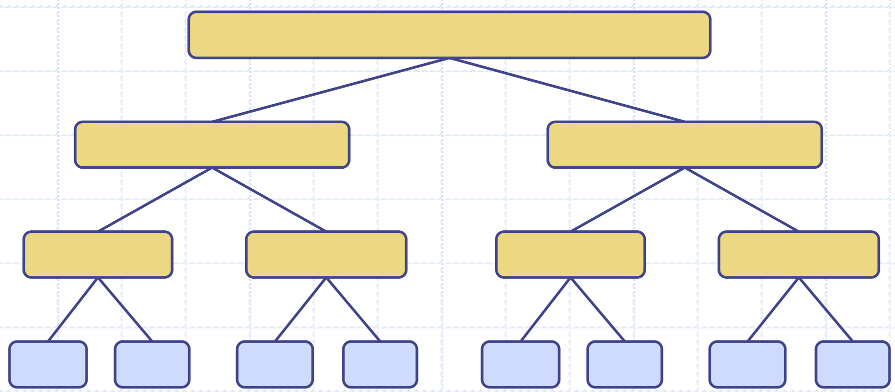
depth	#seqs	size
0	1	n
1	2	$n \div 2$
i	2^i	$n \div 2^i$
…	…	…



Analysis of Merge-Sort

- Depth h of the merge-sort tree is $O(\lg n)$
- Overall amount of work done at nodes of depth i is $O(n)$
- Thus, the total running time of merge-sort is $O(n \lg n)$
- $1 = n \div 2^h$
- $n = 2^h$
- $h = \lg n$

depth	#seqs	size
0	1	n
1	2	$n \div 2$
i	2^i	$n \div 2^i$
…	…	…

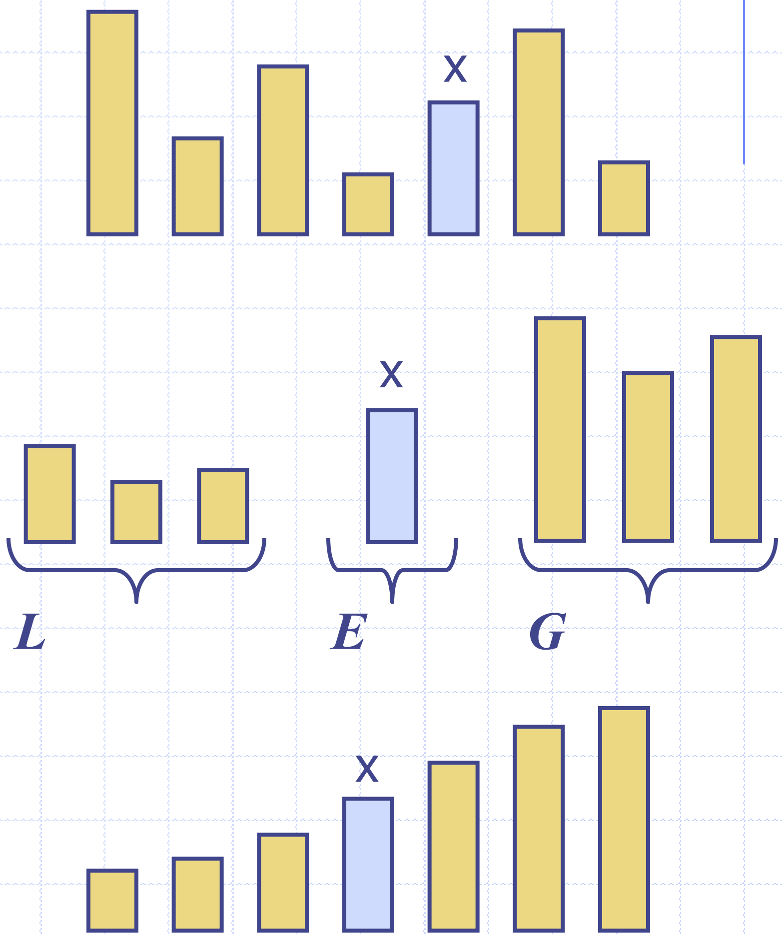


Week 2 – Recursion & Sorting

1. Recursion
2. Analysis of recursive algorithms
3. Divide-and-conquer paradigm
4. Mergesort
5. Quicksort

Quick-Sort

- **Randomised** divide-and-conquer sorting algorithm
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal to x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



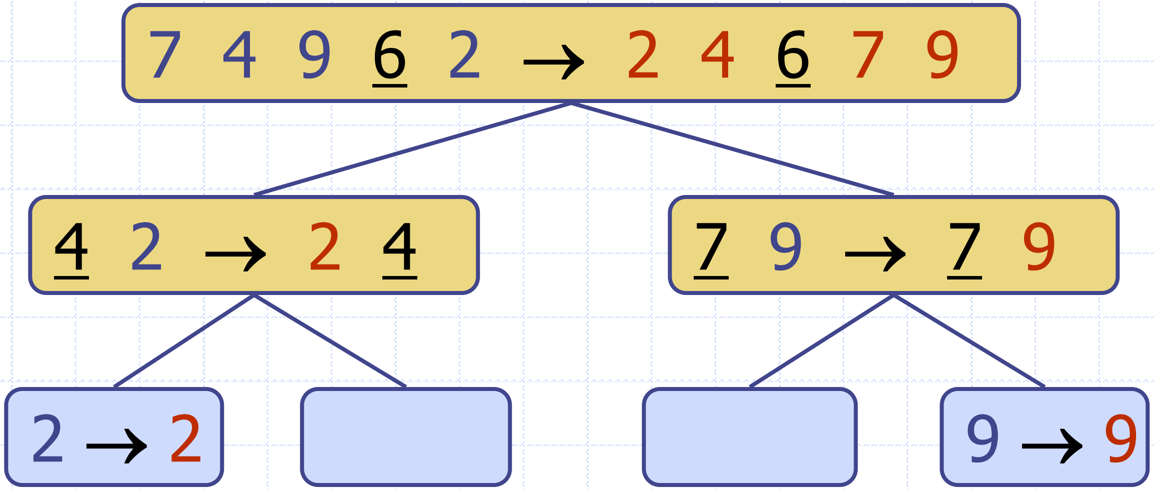
Partition

- Partition input sequence
  - remove, in turn, each element y from S, and
  - insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or end of a sequence
  - hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot.
    $L, E, G \leftarrow$ empty sequences
    $x \leftarrow S$.remove(p)
    while $\neg S$.isEmpty()
        $y \leftarrow S$.remove(S.first())
        if $y < x$
            L.add(y)
        else if $y = x$
            E.add(y)
        else $\{ y > x \}$
            G.add(y)
    return L, E, G

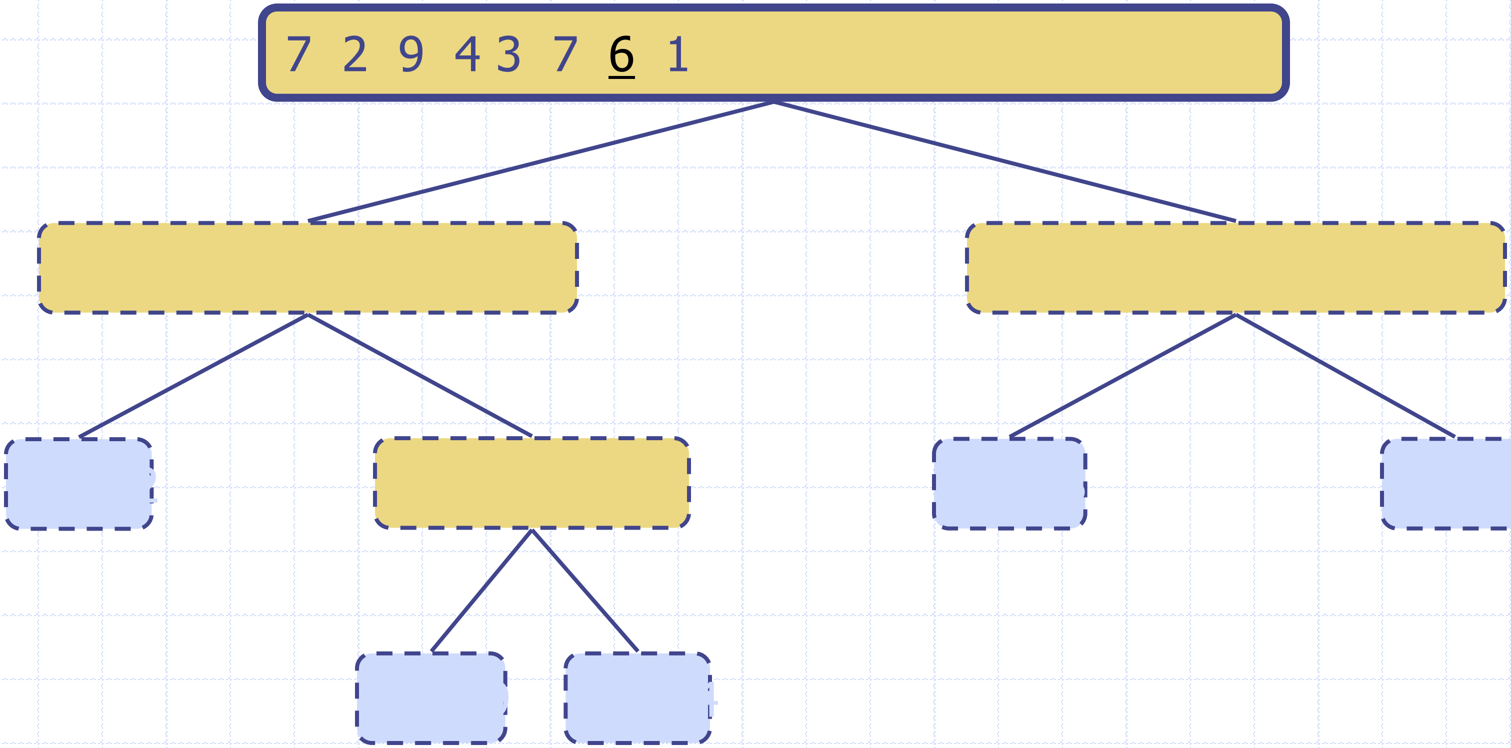
Quick-Sort Recursion Tree

- Execution of quick-sort is depicted by a recursion tree
  - each node represents a recursive call of quick-sort and stores



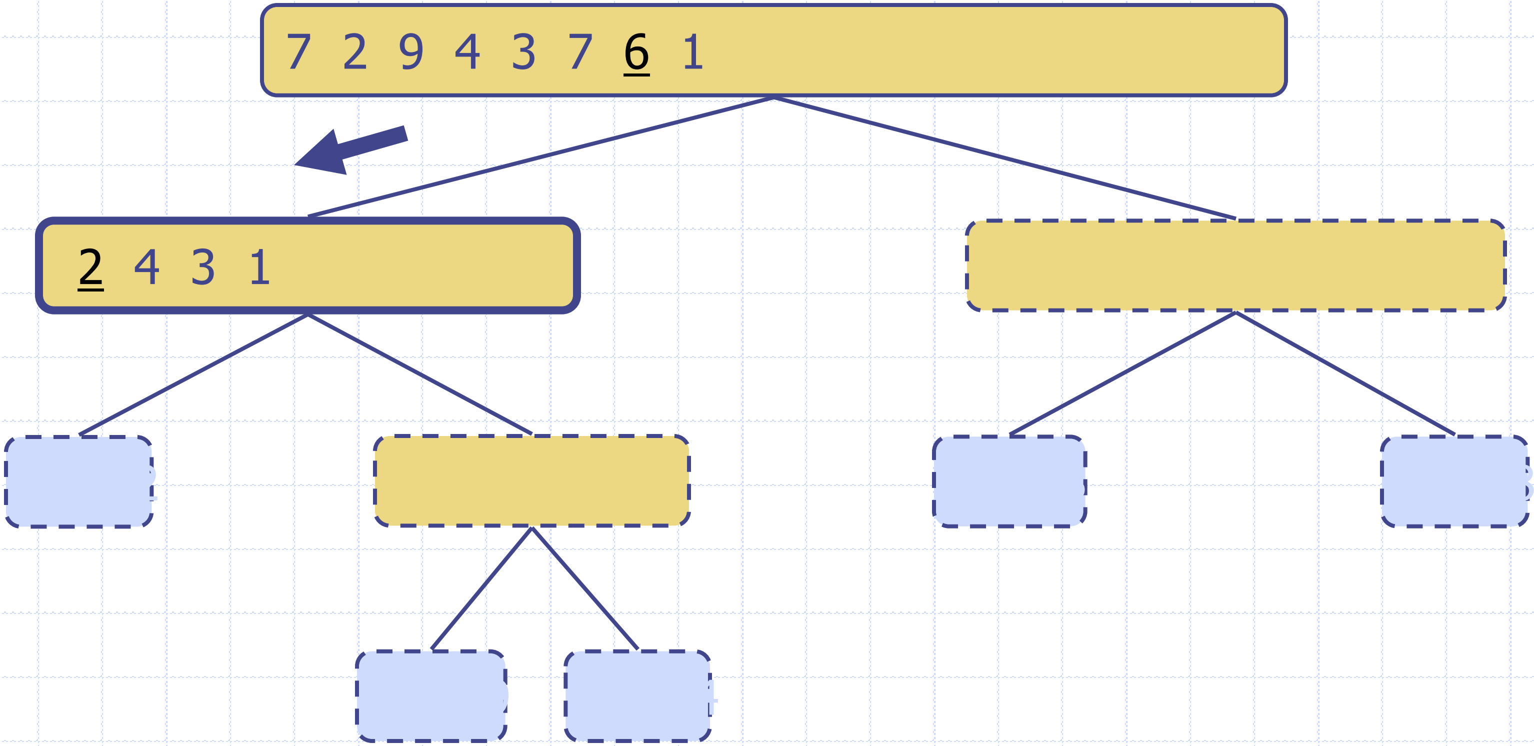
Execution Example

- Pivot selection



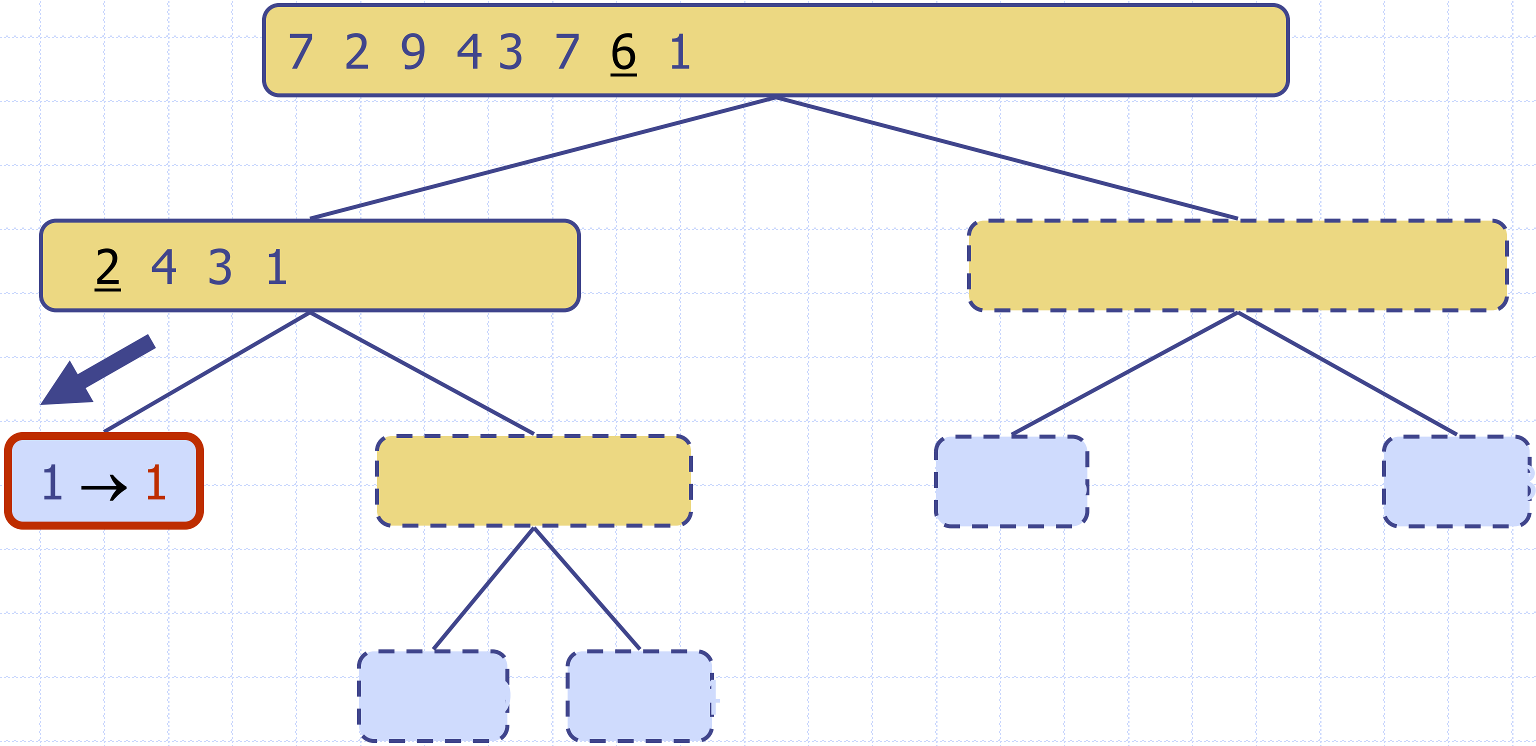
Execution Example (cont.)

- Partition, recursive call, pivot selection



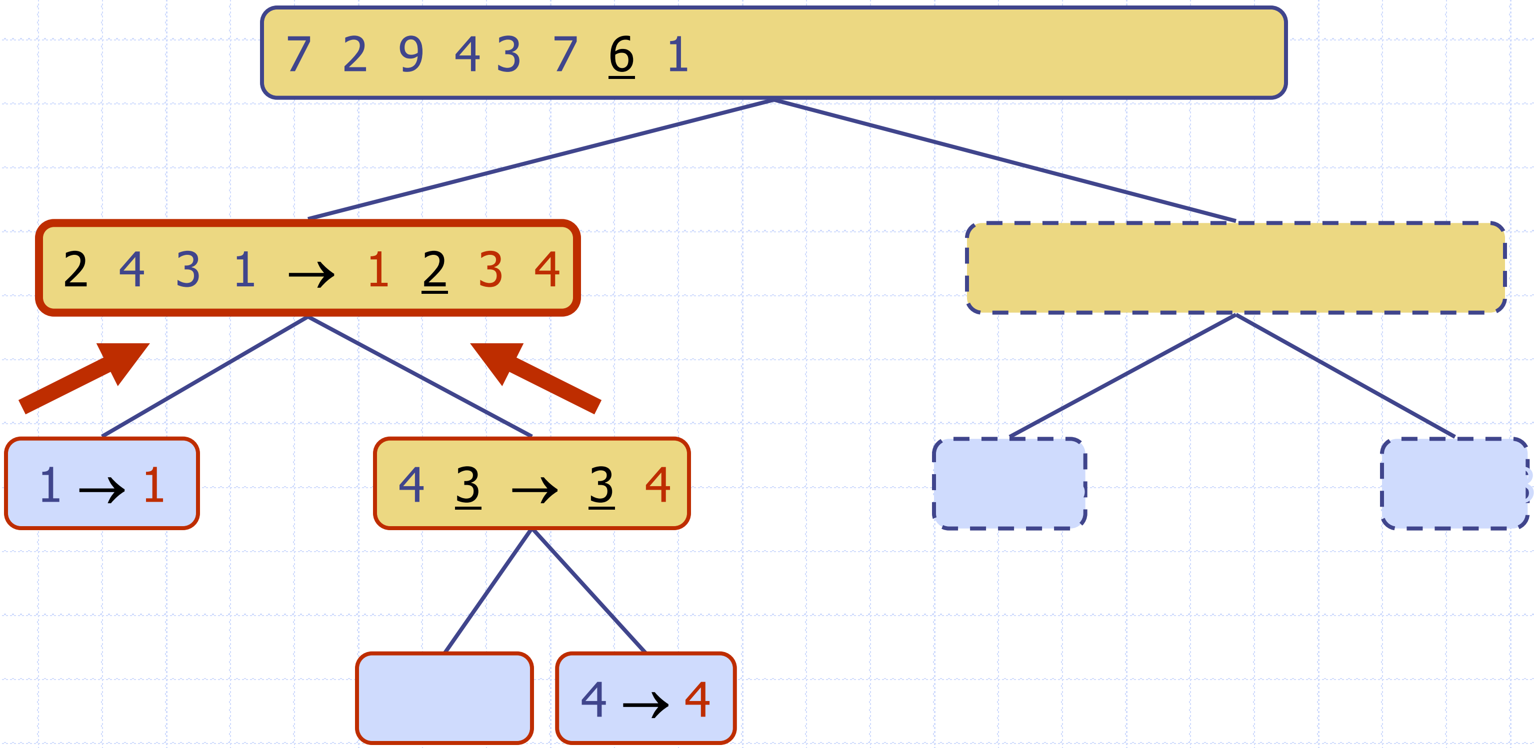
Execution Example (cont.)

- Partition, recursive call, base case



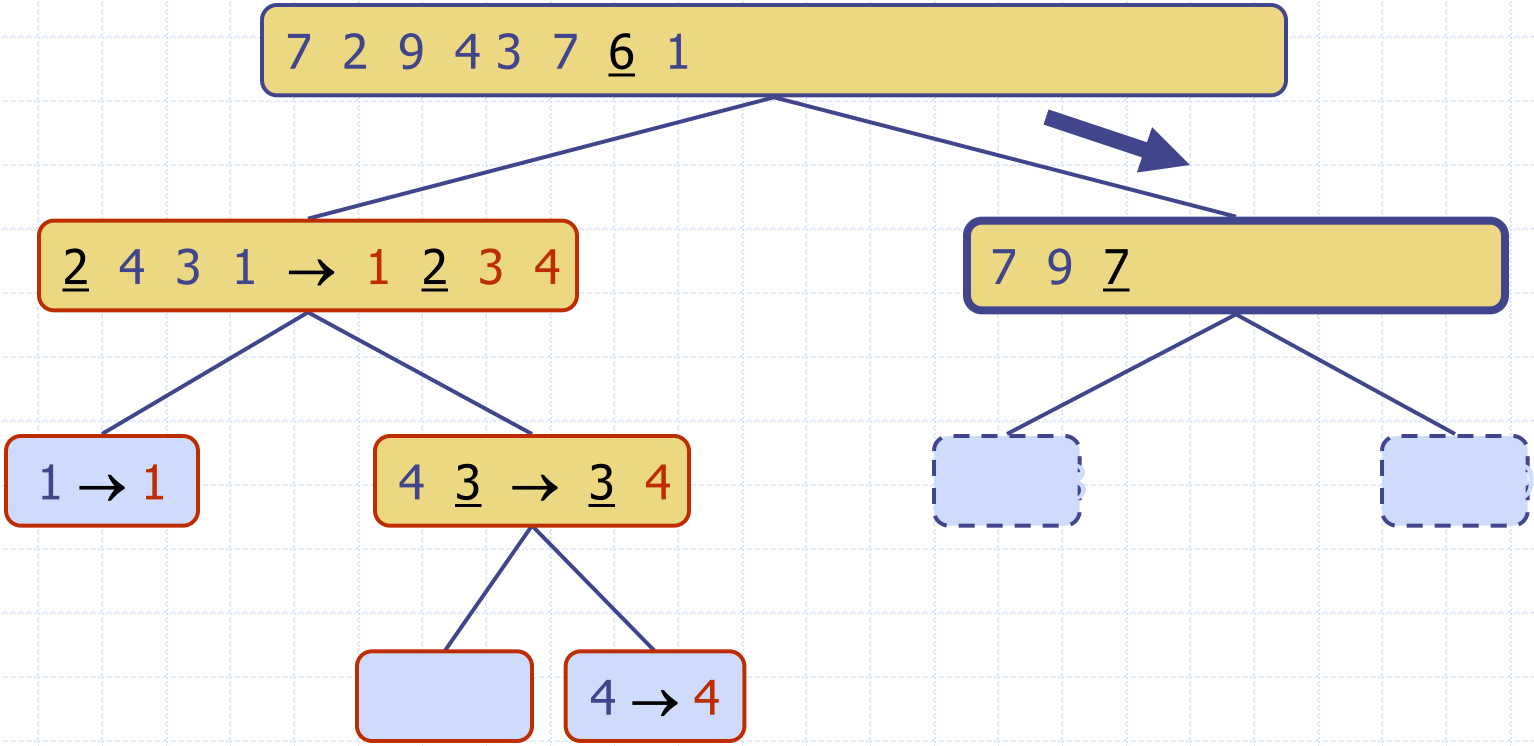
Execution Example (cont.)

- Recursive call, …, base case, join



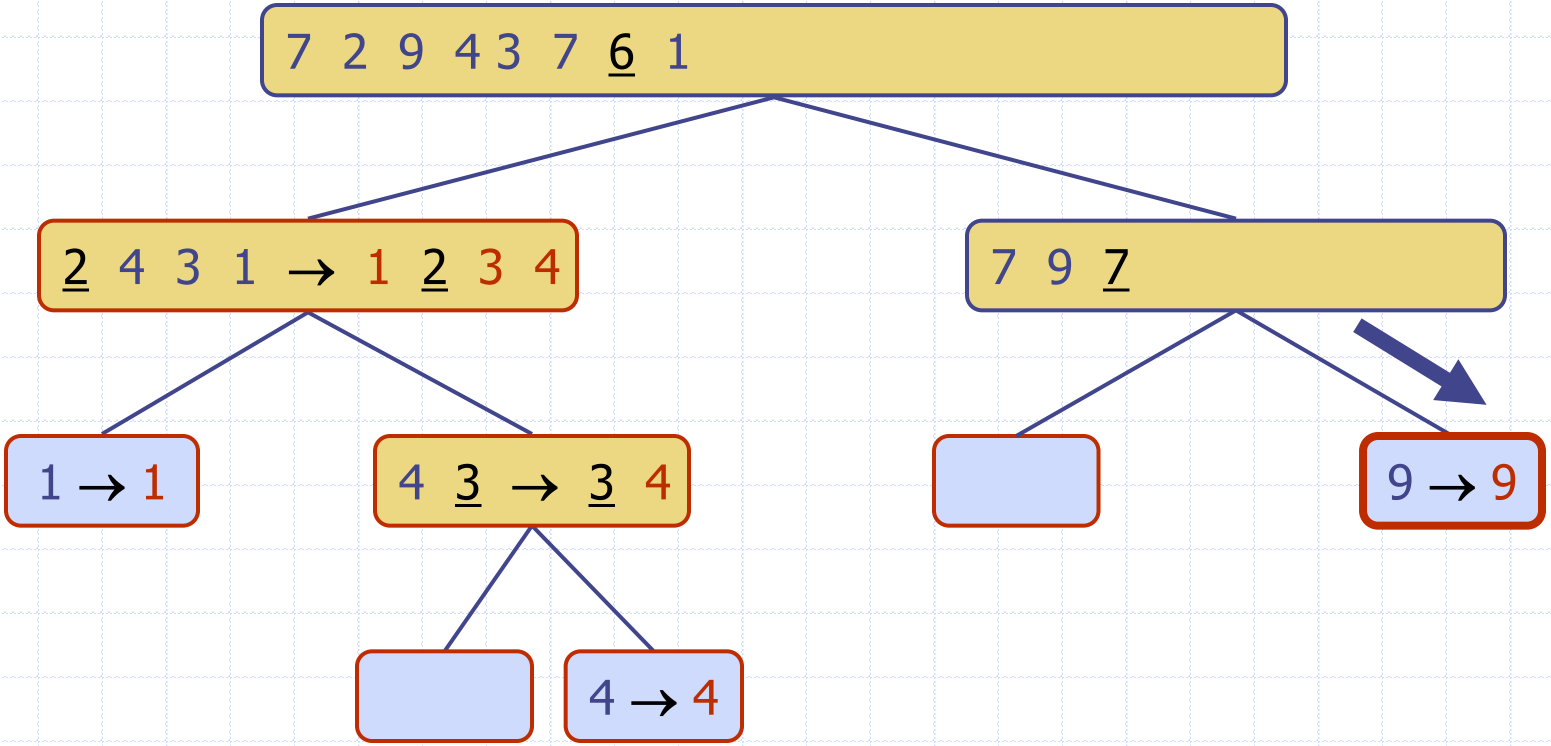
Execution Example (cont.)

- Recursive call, pivot selection



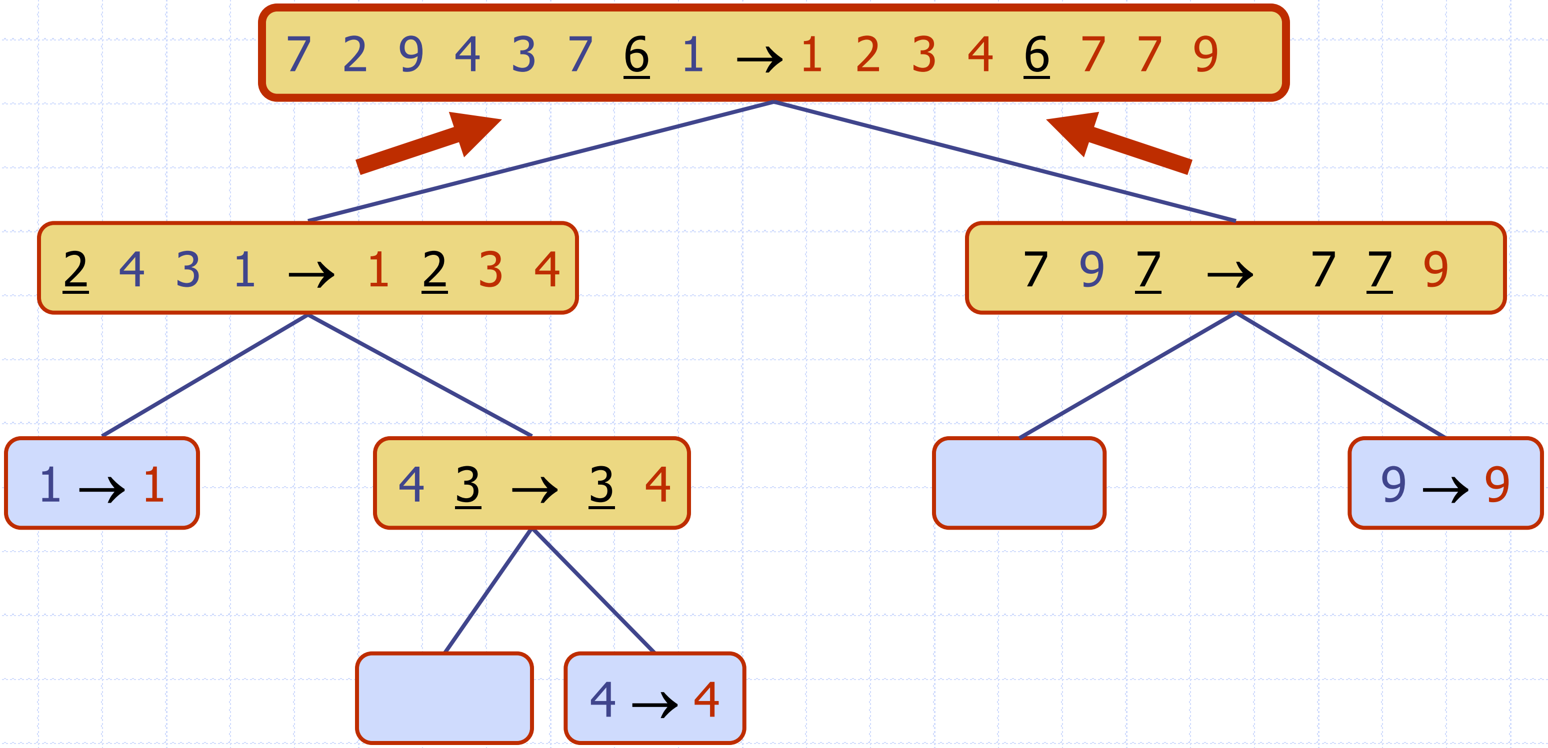
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

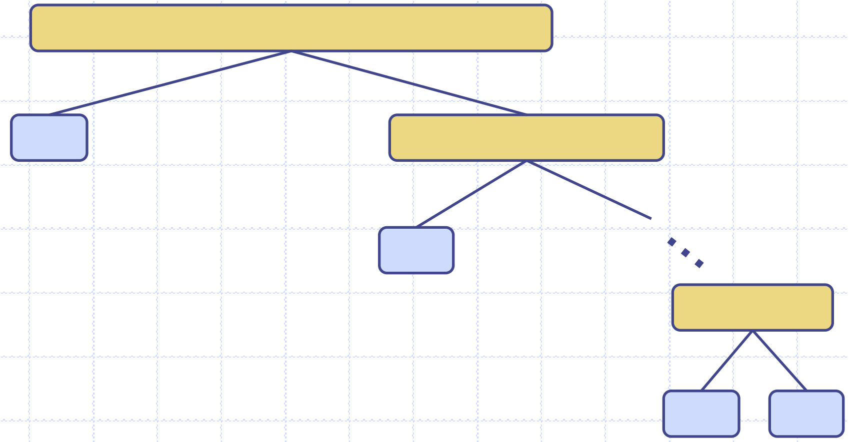
- Join, join



Worst-Case Running Time

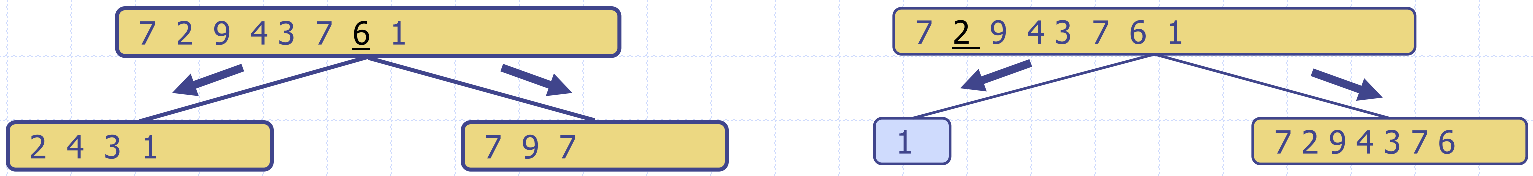
- Worst case for quick-sort occurs when the pivot is the minimum or maximum element
- Running time is proportional to the sum: $n + (n-1) + \ldots + 2 + 1$
- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



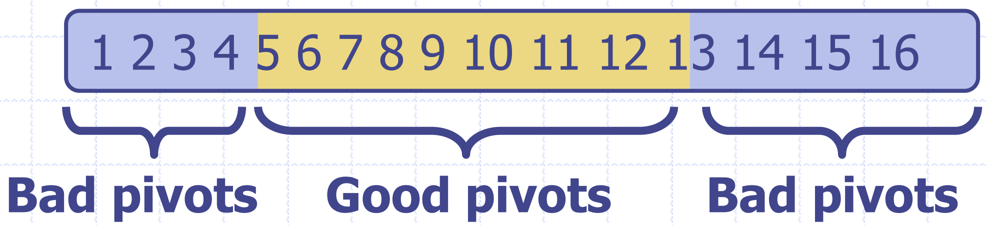
Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$



Good call **Bad call**

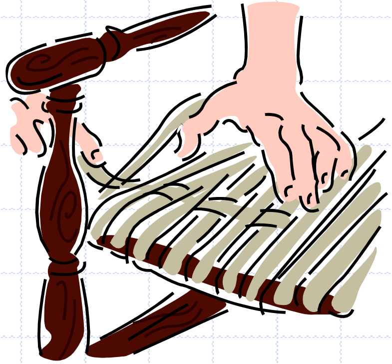
- Good calls have a probability of $1/2$
  - ½ of the possible pivots cause good calls



Expected Running Time, Part 2

- Probabilistic Fact: Expected number of coin tosses required in order to get k heads is $2k$
  - expected height of the quick-sort tree is $O(\log n)$
  - Amount of work done at nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$

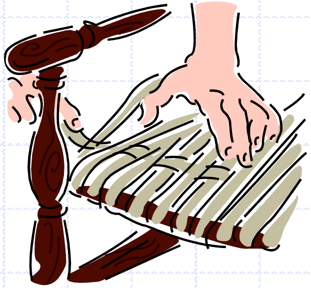
In-Place Quick-Sort



- In partition step, use replace operations to rearrange the elements of the input sequence
  - elements less than the pivot have index less than h
  - elements equal to the pivot have index between h and k
  - elements greater than the pivot have index greater than k
- Recursive calls consider
  - elements with index less than h
  - elements with index greater than k

Algorithm inPlaceQuickSort(A, l, r)
  Input array A, indices l and r
  Output array A with the elements of index between l and r rearranged in increasing order
  if $l \geq r$
    return
  $i \leftarrow$ random integer between l and r
  $x \leftarrow A[i]$
  $(h, k) \leftarrow$ inPlacePartition(A, x)
  inPlaceQuickSort(A, l, $h - 1$)
  inPlaceQuickSort(A, $k + 1$, r)

In-Place Partitioning



- Perform partition using two indices to split S into L and E & G (a similar method can split E & G into E and G).

j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9 (pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element ≥ x.
  - Scan k to the left until finding an element < x.
  - Swap elements at indices j and k

→ j k ←

3 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 **6** 9

A nice example of Big Omega

- What if I instead did something like this?

```
def permutation_sort(my_list):
  for perm in gen_permutations(my_list):
    if is_sorted(perm):
      return perm
```

- Worst Case Analysis, please?

A nice example of Big Omega

- What if I instead did something like this?

```
def permutation_sort(my_list):
  for perm in gen_permutations(my_list):
    if is_sorted(perm):
      return perm
```

- Worst Case Analysis, please?
  1. I do a call to some algorithm gen_permutations()
  2. I iterate over all of those, and I check is_sorted() function

A nice example of Big Omega

- What if I instead did something like this?

```
def permutation_sort(my_list):
  for perm in gen_permutations(my_list):
    if is_sorted(perm):
      return perm
```

- Worst Case Analysis, please?
  1. I do a call to some algorithm gen_permutations()
  2. I iterate over all of those, and I check is_sorted() function

We know how to do is_sorted() in $O(n)$ time

What do we know about permutations?

Fun with Sorting

```
def bogo_sort(my_list):

  while is_sorted(my_list) == False:
    random_shuffle(my_list)
```

Worst case?

Average case?

Best case?

Fun with Sorting

```
def bogo_sort(my_list):
  while is_sorted(my_list) == False:
    random_shuffle(my_list)
```

Worst case? Unbounded (!!)

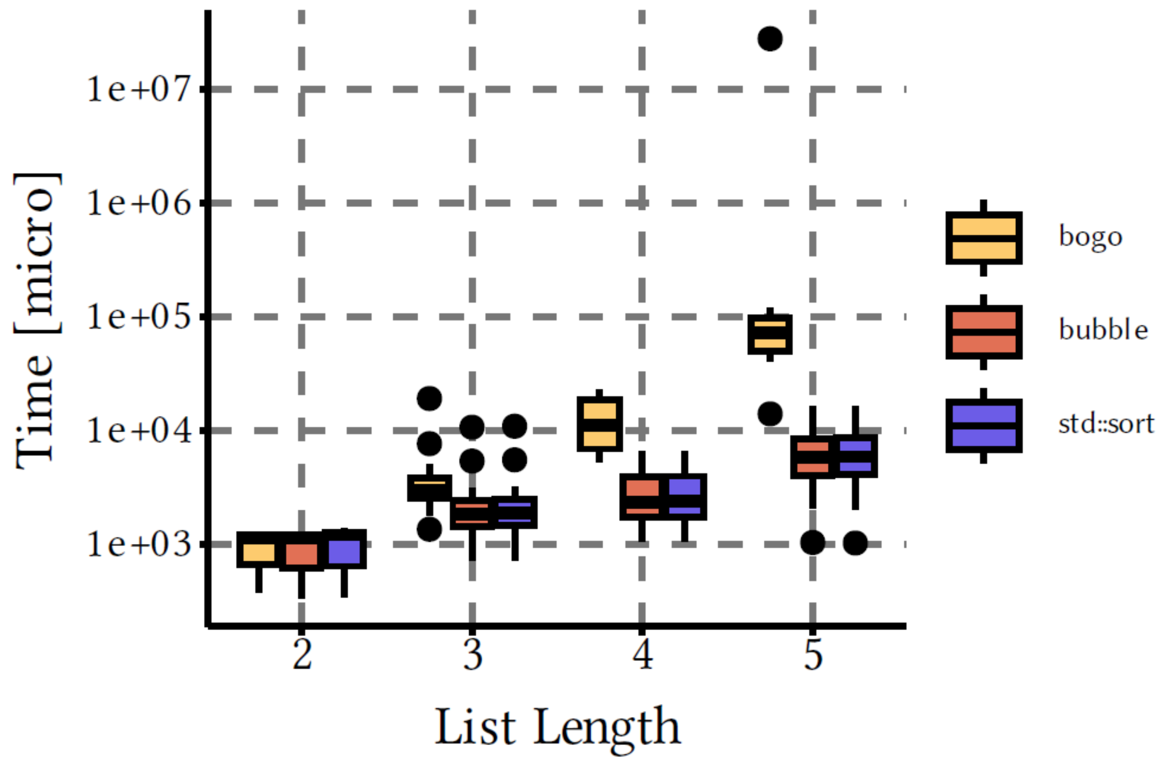
Average case? $\Theta(n * n!)$

Best case? $\Omega(n)$

Gruber, H.; Holzer, M.; Ruepp, O., "Sorting the slow way: an analysis of perversely awful randomized sorting algorithms", 4th International Conference on Fun with Algorithms, Castiglioncello, Italy, 2007

In the real world…

- Application: Lots of sort calls to very short lists [within a prototype search system]



Next Week

- More sorting (advanced techniques)
- Lists and arrays
- Amortization
- Linked Structures

- More fun with sorting
  - https://www.youtube.com/watch?v=kPRA0W1kECg