Sorting
And Order Statistics

COMP2611: Data Structures

2019/2020

Outline

- Motivation
- Definitions
  - What is sorting?
  - Unstable vs stable sorting
  - Comparison based vs non-comparison based sorting
  - Offline vs Online
- Linearithmetic sorting algorithms:
  - Mergesort
  - Quicksort
- Linear sorting
  - Counting sort and Bucket sort
- Order statistics
  - Finding the $\mathbf{n}^{th}$ smallest/largest element

The Problem (Informally)

- Turn this

10	19	7	4	3	21	10	23	24	18	1	8	23	1	12

- Into this

1	1	3	4	7	8	10	10	12	18	19	21	23	23	24

- as efficiently as possible

The Problem

- Suppose that we have a sequence of items.
- These items adhere to one of these three properties:
  - The items have a partial ordering (i.e. can use comparison operators) defined on them, e.g. integers
  - The items have a key field that has a partial ordering defined on them e.g. student records with a student id as the key field
  - There is a key function that can be applied to the items that returns values that have a partial ordering defined on them, e.g. string length can be applied to strings

The Problem

- We want to re-order (find permutation) of items such that we produce a sequence with all the items in either ascending or descending order
- Relatively easy to swap between ascending and descending case
  - Will focus on ascending case
  - But results generalise (need to edit pseudocode)

The Problem (Formally)

- Consider that we are given a sequence of items $\mathtt{a_1,\ a_2,\ a_3,\ \ldots\ a_k}$
  - $\{1, 2, 3, \ldots, k\} = I \subset N$
  - Typically items are stored in the array
- Items have a partial ordering defined on them (depending on context)
  - Items can be structs, objects, or records that either have a key field or some key function that whose output acts as key (e.g. string length)
- Find bijection, $S: I \to I$, such that
  - For every a_k in our sequence, we have $a_k = b_{(S(k))}$
  - $b_1 \leq b_2 \leq b_3 \leq \ldots \leq b_k$
- We typically return the sorted sequence rather than the bijection itself

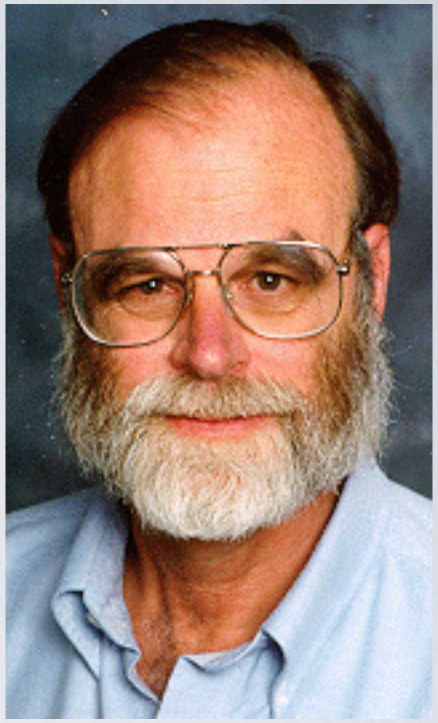
Sorting is Serious!

Microsoft Research team shatters data sorting record, wrenches trophy from Yahoo

Alexis Santos
05.22.12

59
Shares

Sorting Competition



- Sort Benchmark (sortbenchmark.org)
- Started by Jim Gray
  - Research scientist at Microsoft Research
  - Winner of 1998 Turing Award for contributions to databases
- Tencent Sort from Tencent Corp. (2016)
  - 100 TB in 134 seconds
  - 37 TB in 1 minute

Why?

- Why do we care so much about sorting?
- Rule of thumb:
  - "good things happen when data is sorted"
  - we can find things faster (e.g., using binary search)

Sorting Algorithms

- There are many ways to sort arrays
  - Iterative vs. recursive
  - in-place vs. not-in-place
  - comparison-based vs. non-comparative
  - Stable vs unstable
- In-place algorithms
  - transform data structure w/ small amount of extra storage (i.e., `O(1)` space complexity)
  - For sorting: array is overwritten by output instead of creating new array

Pseudocode

- Sorting algorithms are used in a wide variety of circumstances.
- Pseudocode will capture important intuitions, but you would need to adjust uses of comparisons to suite depending on situation in real code!
- We will use **`key_func`** to abstract away the process of computing or extracting the key from data
  - Conventions: returns
    - `0`, if they are "equal"
    - `1`, if the first argument is greater than the second
    - **`-1`** if the first argument is less than the second

"In-Placeness"

- Reversing an array

```
function reverse(A):
  n = A.length
  B = array of length n
  for i = 0 to n – 1:
    B[n-1-i] = A[i]
  return B
```

Not in-place!

```
function reverse(A):
  n = A.length
  for i = 0 to n/2:
    temp = A[i]
    A[i] = A[n-1-i]
    A[n-1-i] = temp
```

in-place

Return statement not needed

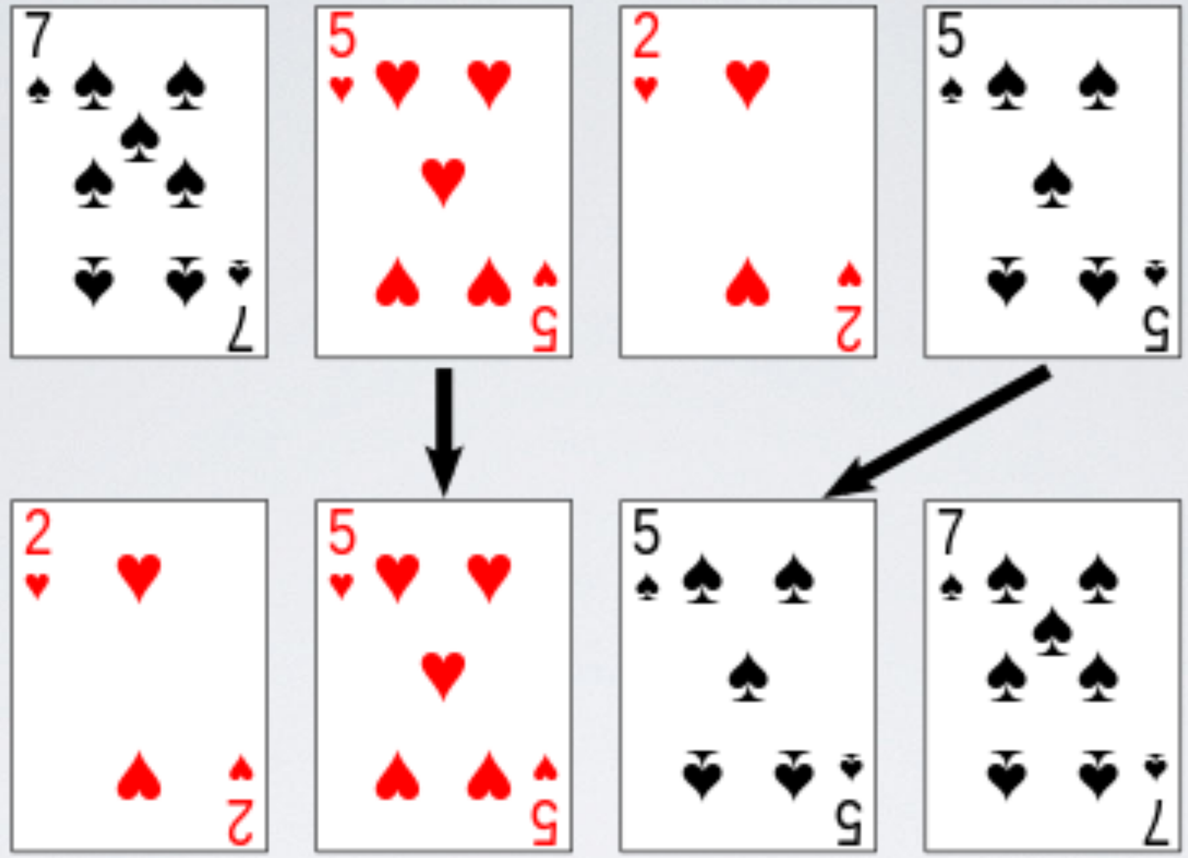
Properties of In-Place Solutions

- Harder to write **:-(**
- Use less memory **:-)**
- Even harder to write for recursive algorithms **:-(**
- Tradeoff between simplicity and efficiency

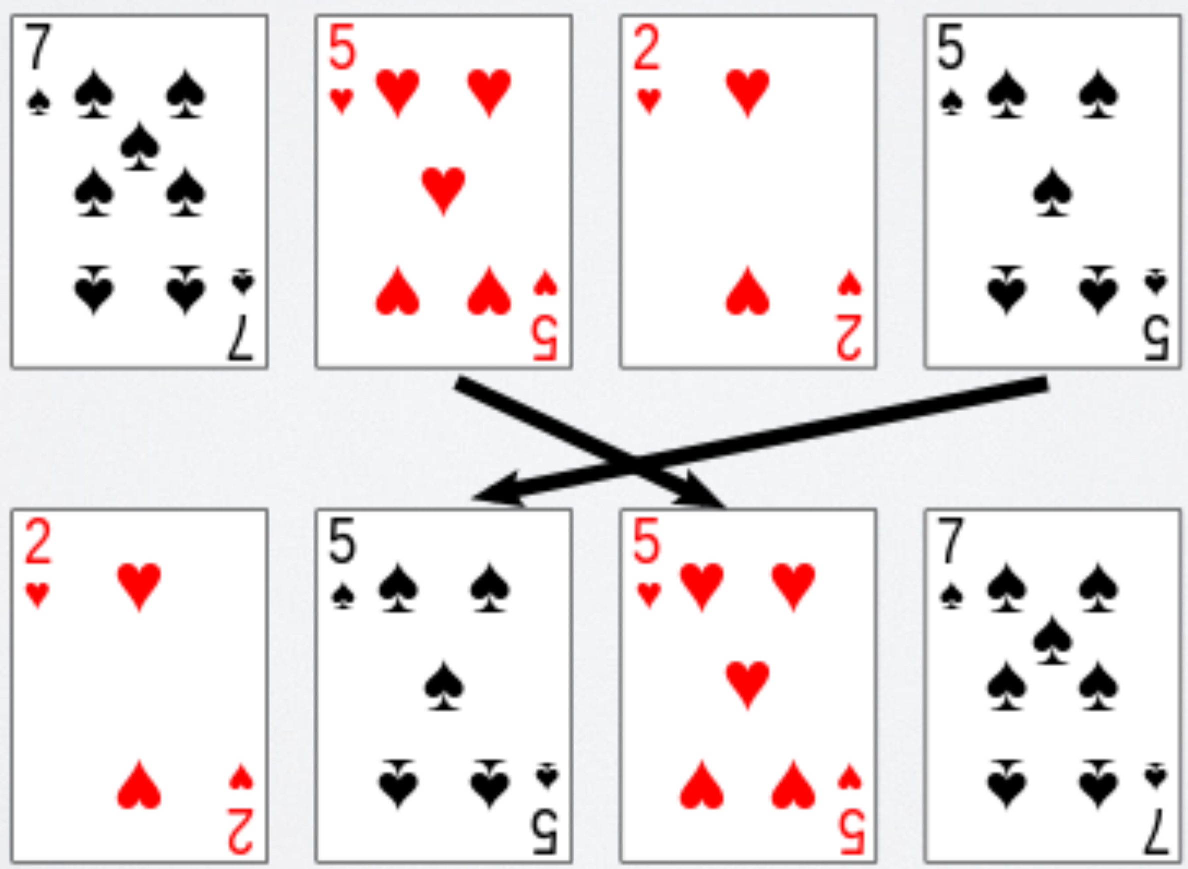
Stability

- A sorting algorithm that maintains the relative ordering of "equal" entries is considered stable
- Another example
  - Suppose that we have the array of strings: `[“foobaz”, “foobaz”, “baz”, “alice”, “bob”]`
  - Sort by string length
  - Stable sorting of array `[“baz”, “bob”, “alice”, “foobar”, “foobaz”]`
  - Unstable sorting of array: `[“bob”, “baz” ,“alice”, “foobar”, “foobaz”]`

Stable



Not stable



Stability

- Makes output more "predictable"
- Allows us to stack sorts together.
- Example: suppose that we have an array of student structs with first_name and last_name
- Want to sort by last_name and then first_name
- Using stable sorts, we can sort using the last_name as the key, and then sort the result using first_name as the key

Comparison-Based Sorting

- Even though a valid sort of a sequence of data adheres to an ordering
- We don't need to use the ordering to sort items all of the time
- Sorts that use comparisons: comparison-based sorting algorithms
- Sorts that don't use comparisons: non-comparison based sorting algorithms

Offline vs Online

- Offline Algorithm: batch processes data.
- Online Algorithm: serially processes data.
  - Can update solution as new data arrives
- Suppose that new data enters our array after sorting:
  - Offline sorting algorithm: need to sort entire array again (e.g. most sorting algorithms)
  - Online sorting algorithm: sort only portion of array (e.g. insertion sort)

Linearithmetic Sorting

- Most efficient (comparison based) sorting algorithms are O(nlogn)
- Consider a sort as series of yes/no decisions
  - Can be modelled as a binary tree
  - Each leaf is a permutation
  - Sorting algorithm travels to leaves
  - We have `n!` Leaves

Linearithmetic Sorting

- Let tree have height `h`

$$\begin{aligned} 2^h &= n! \\ h &= log_2 n! \\ &= log_2(1 \times 2 \times 3 \cdots \times n) \\ &= log_2 1 + log_2 2 + log_2 3 + \cdots + log_2 n \\ &= \sum_{i=1}^{n} log_2 i \\ &\leq \sum_{i=1}^{n} log_2 n \\ &= n log_2 n \end{aligned}$$

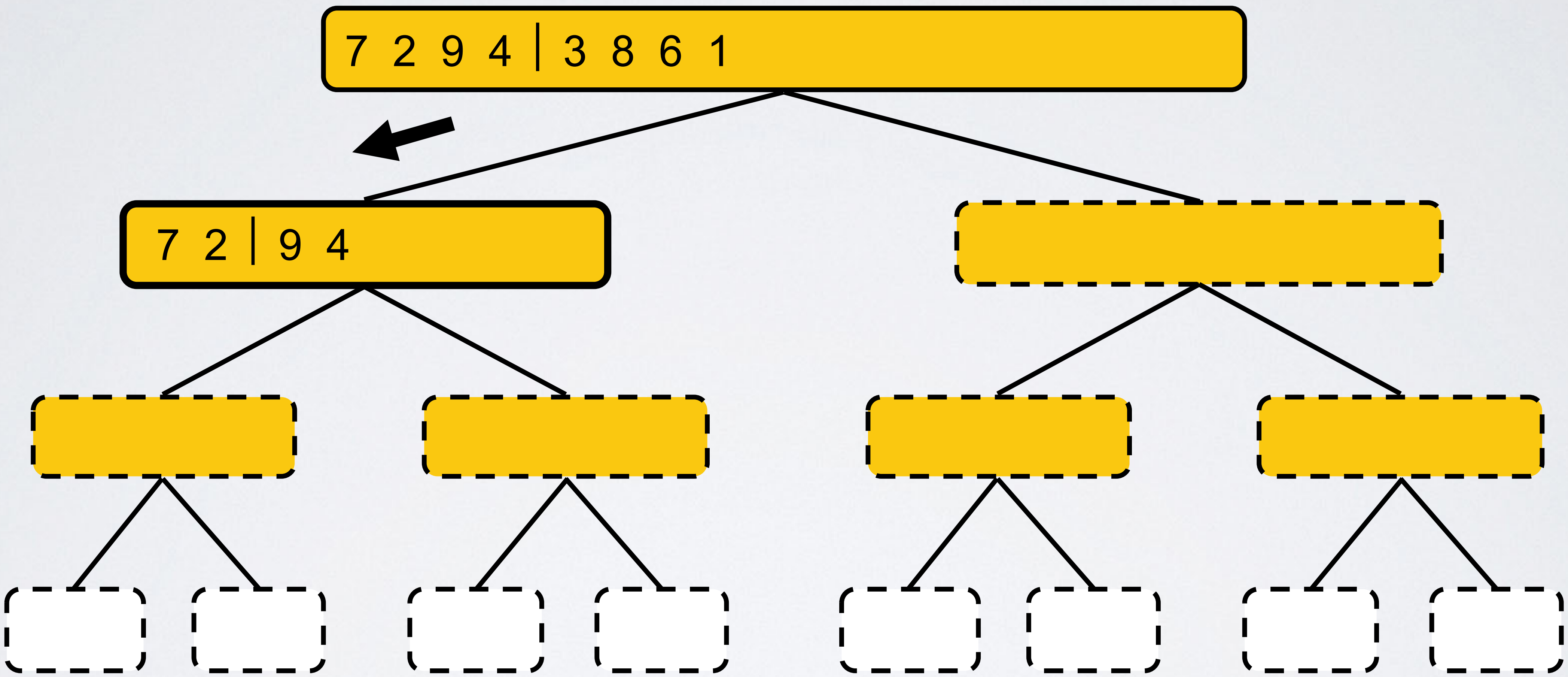
Main examples

- Heapsort: covered last class
- Divide and conquer algorithms:
  - Have base case
  - Recursively divide larger problem into smaller problems, solve smaller problems, and then combine them to solution of larger problem
  - For sorting: recursively divide array into pieces, sort smaller pieces, combine into solution for larger array
  - Two main examples: Mergesort and Quicksort

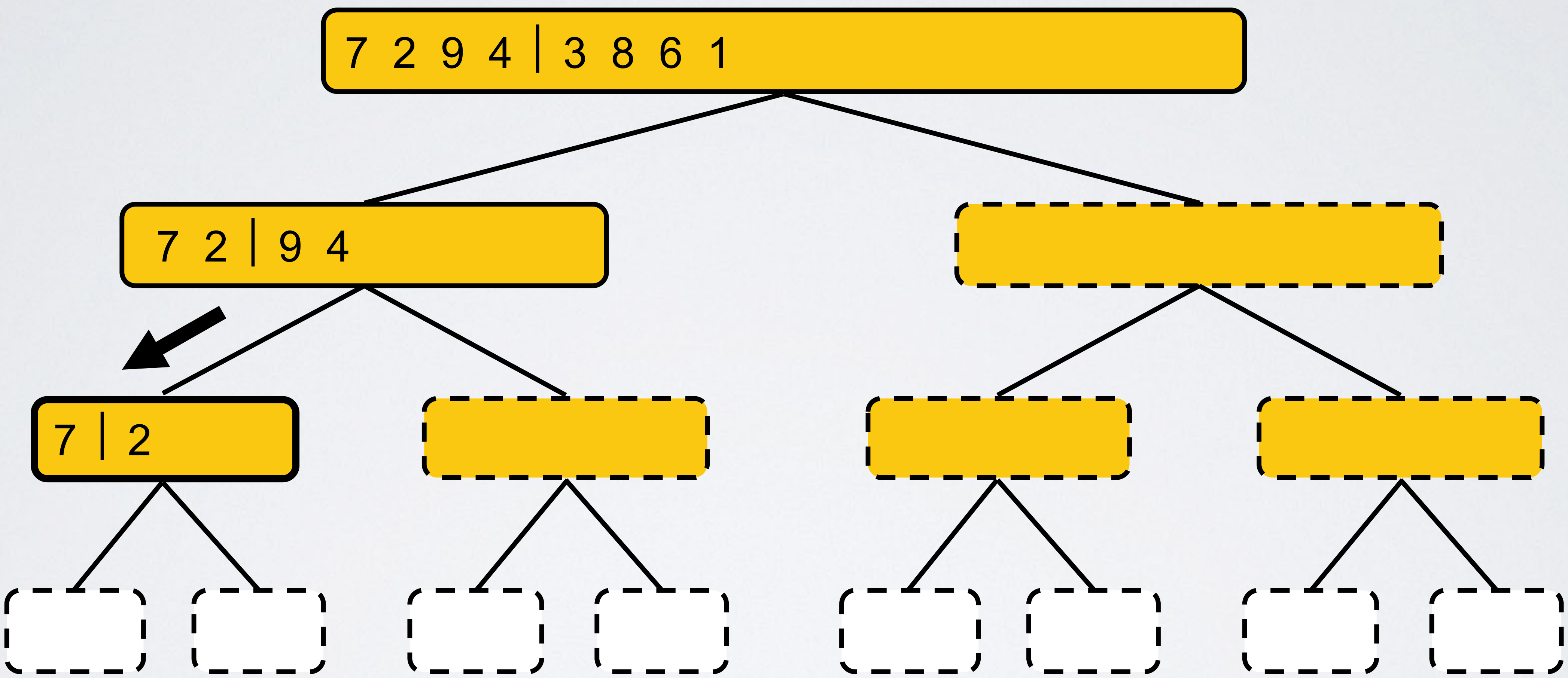
Merge Sort

- Sorting algorithm based on divide & conquer
- Like quadratic sorts
  - comparative
- Unlike quadratic sorts
  - recursive
  - linearithmic `O(nlog n)`

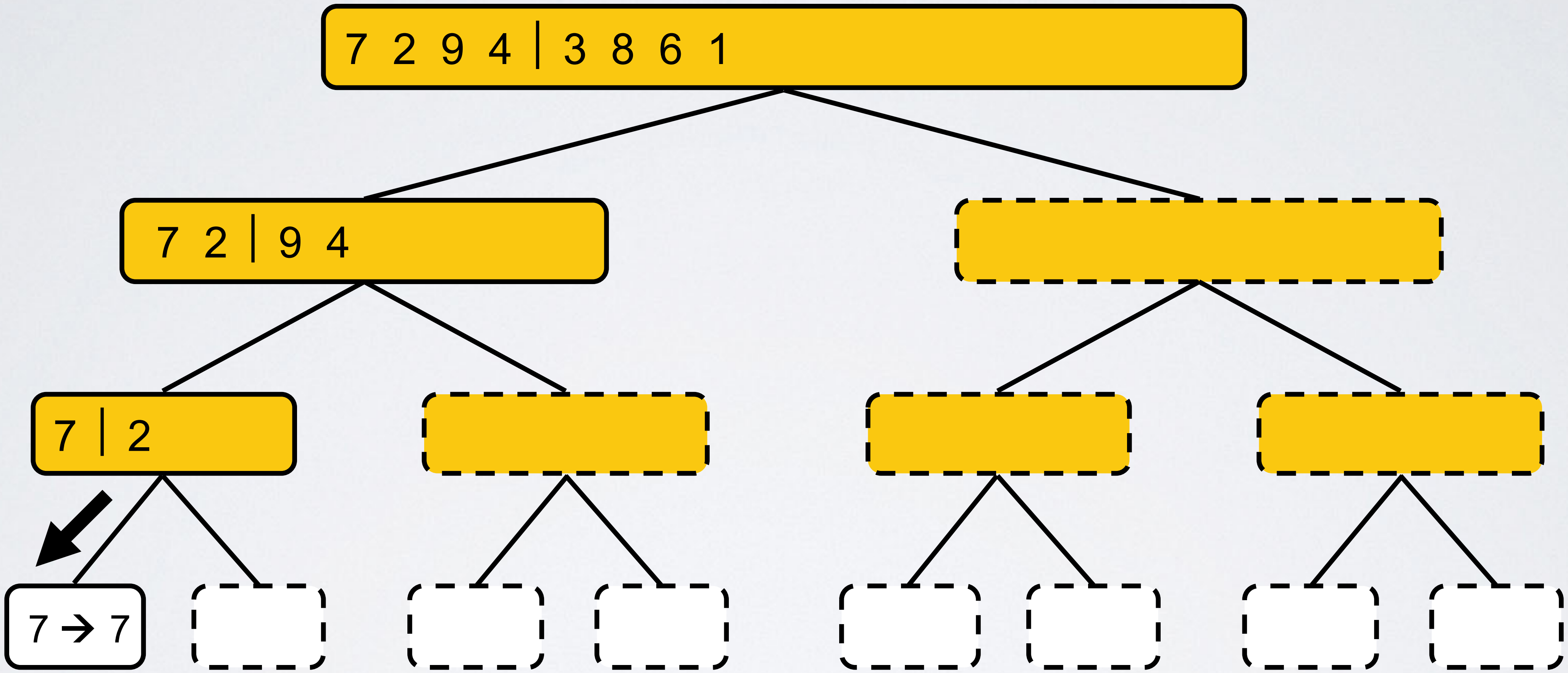
Merge Sort Recursion Tree



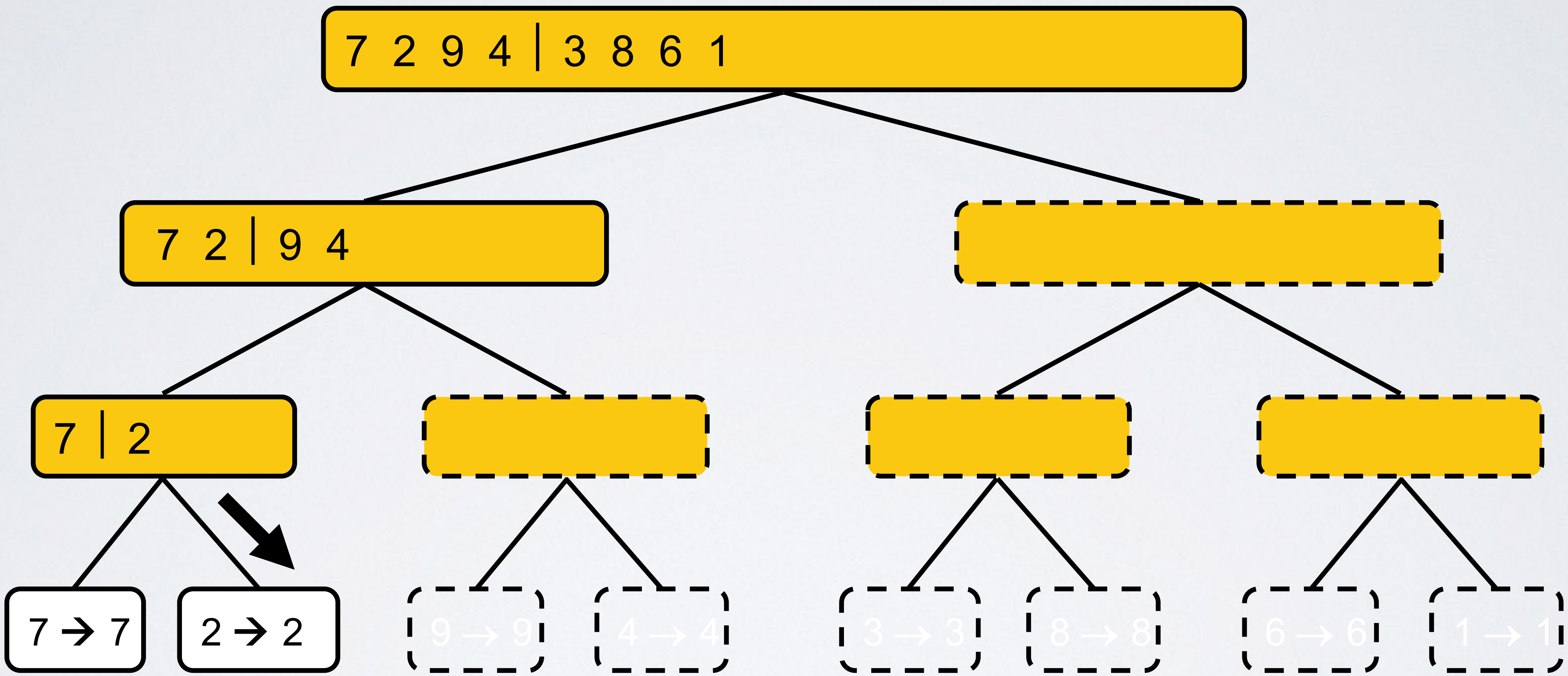
Merge Sort Recursion Tree



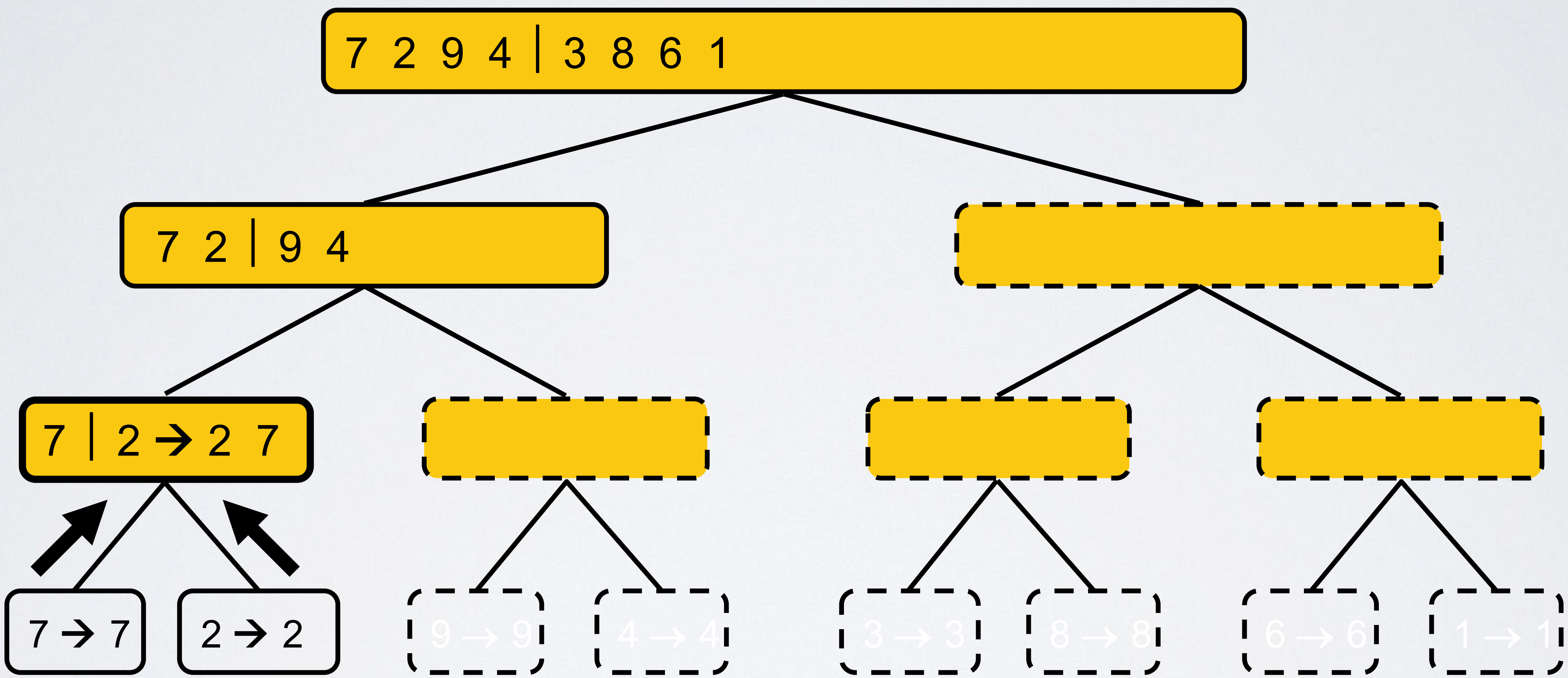
Merge Sort Recursion Tree



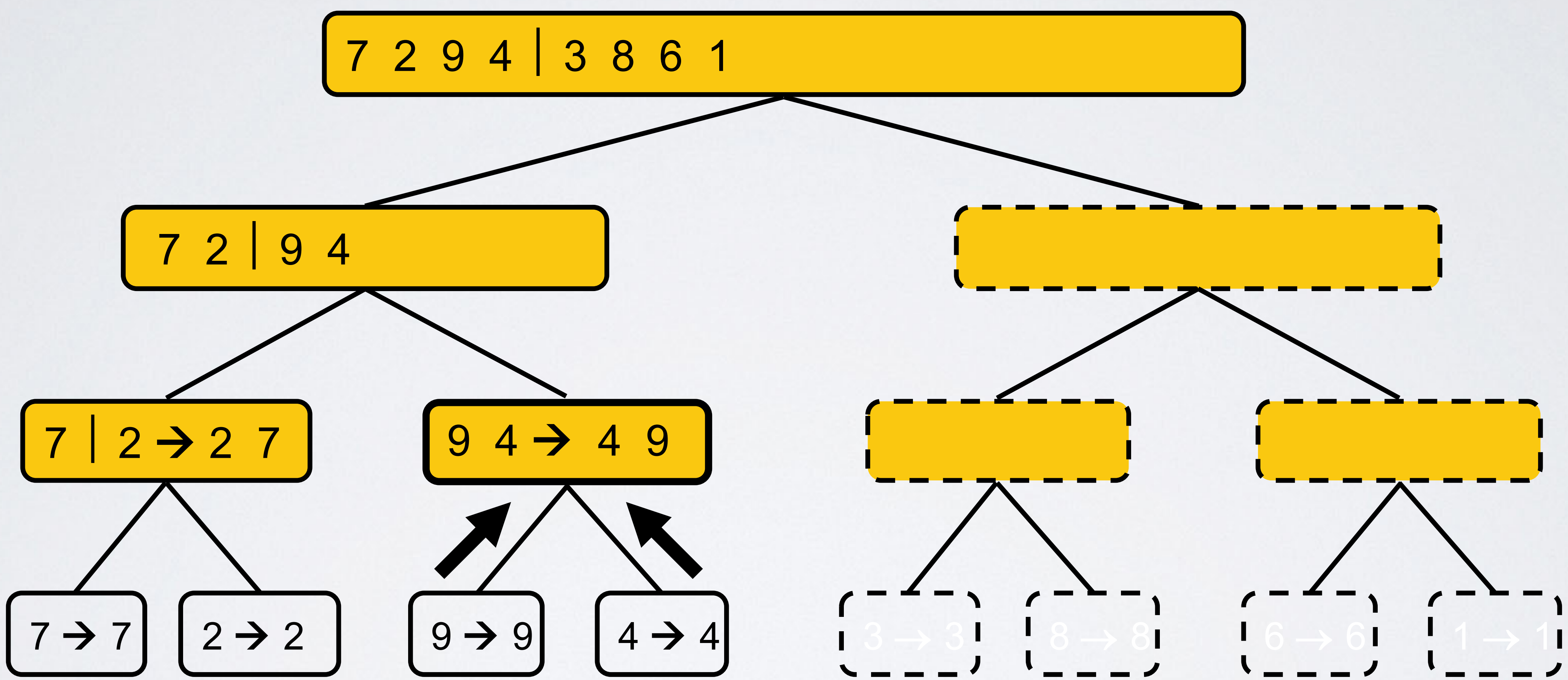
Merge Sort Recursion Tree



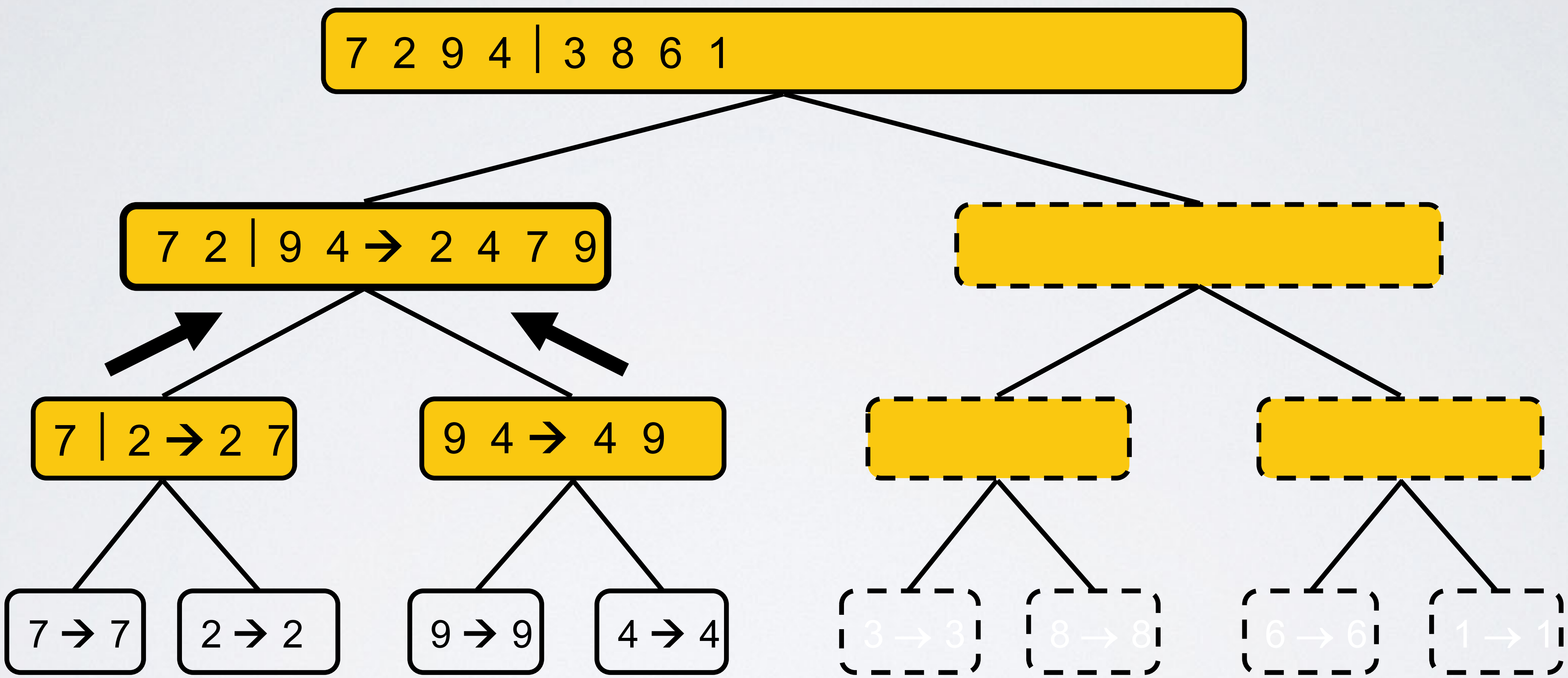
Merge Sort Recursion Tree



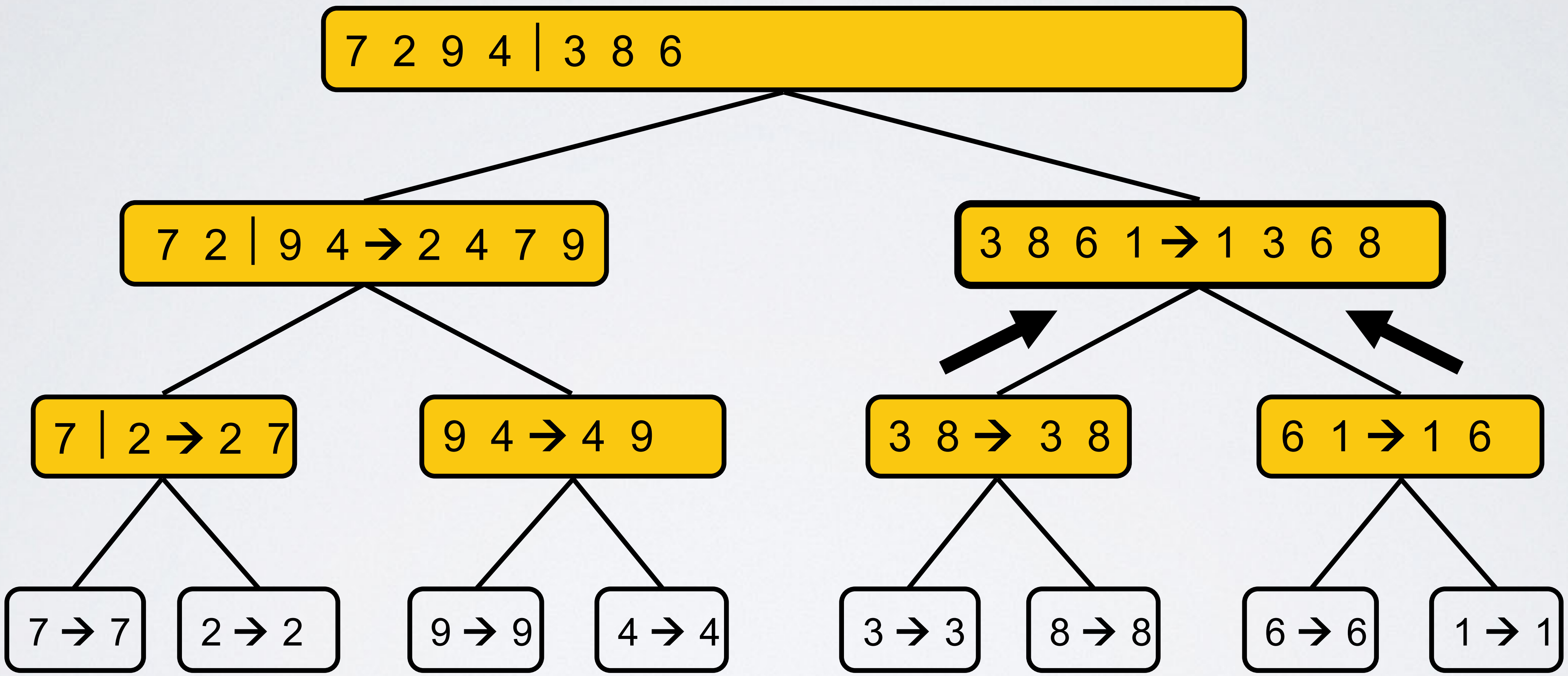
Merge Sort Recursion Tree



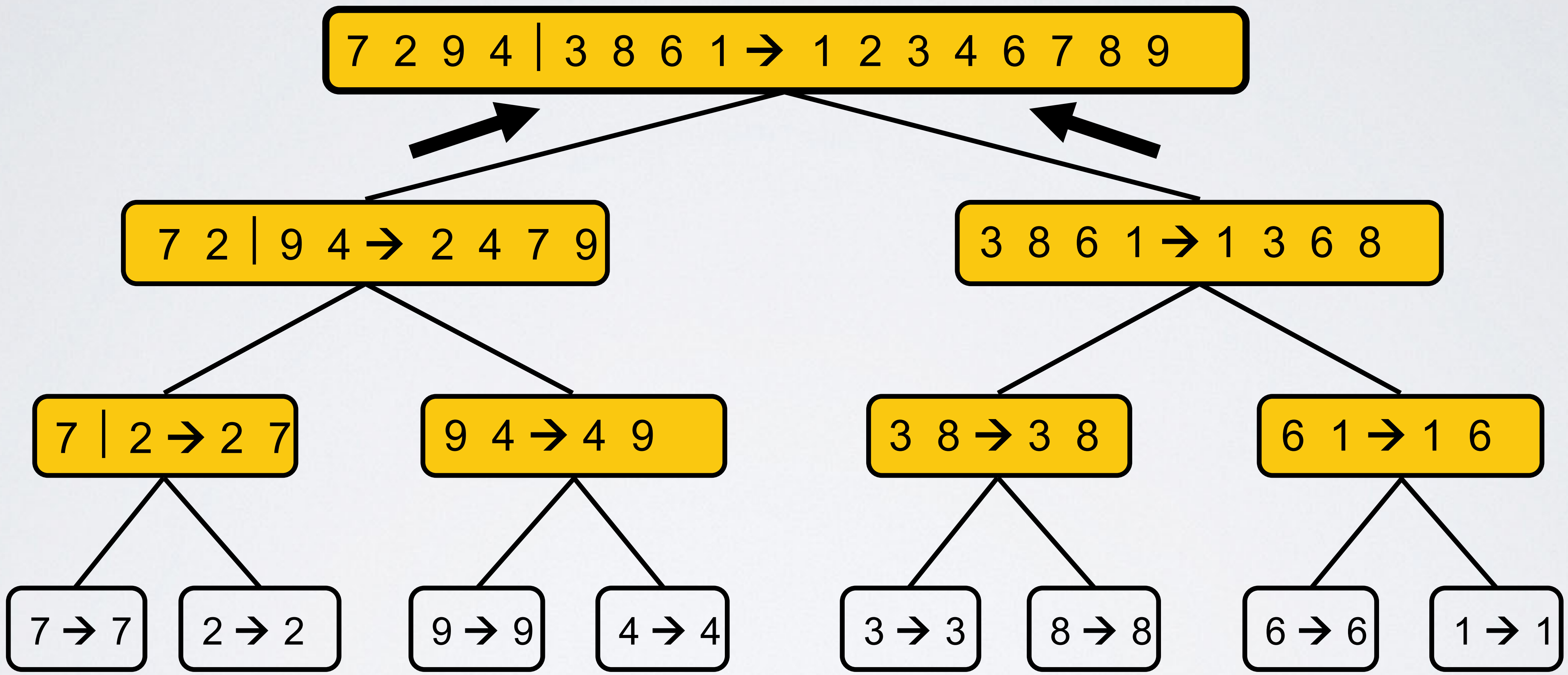
Merge Sort Recursion Tree



Merge Sort Recursion Tree



Merge Sort Recursion Tree

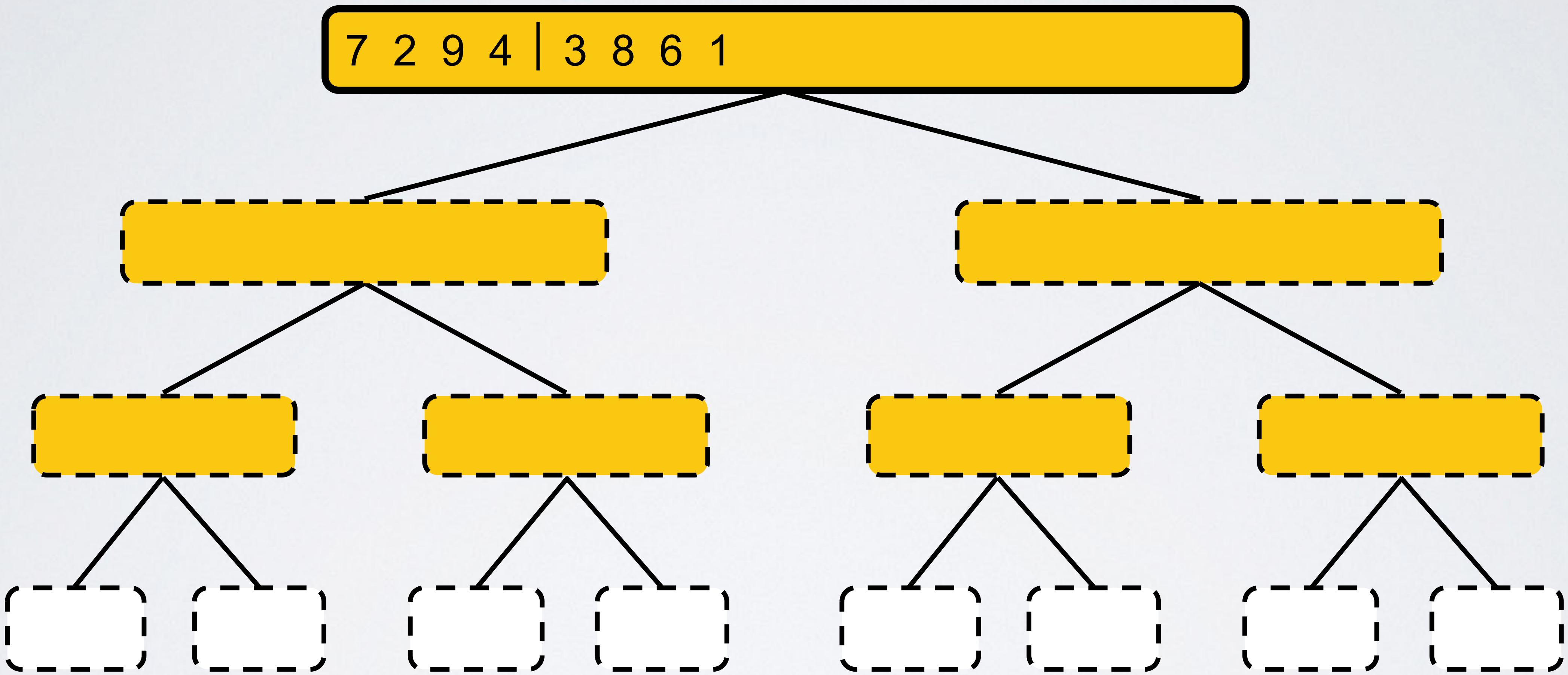


Merge Sort Pseudo-Code

```
function mergeSort(A, lo, hi, key_func):
  if (hi != lo):
    mid = n/2
    mergeSort(A, 0, mid - 1, key_func)
    mergeSort(A, mid , hi, key_func)
    merge(A, lo, mid-1, mid, hi, key_func)
```

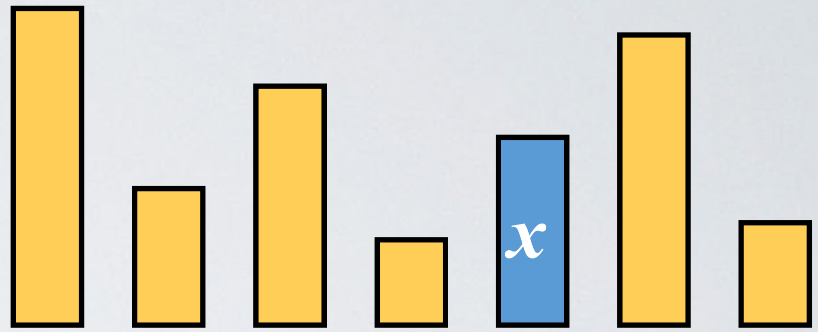
Merge pseudo-code in folder!

Merge Sort Recursion Tree

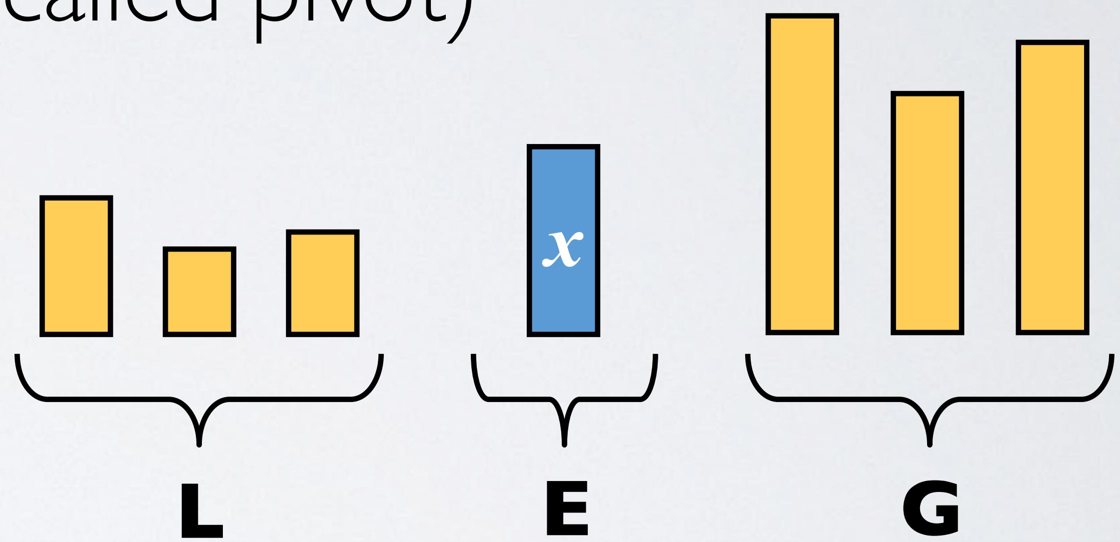


Quicksort

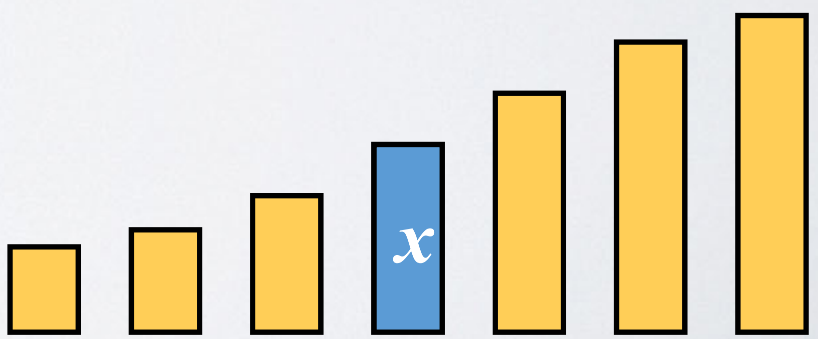
- Randomized sorting algorithm
- Based on divide-and-conquer



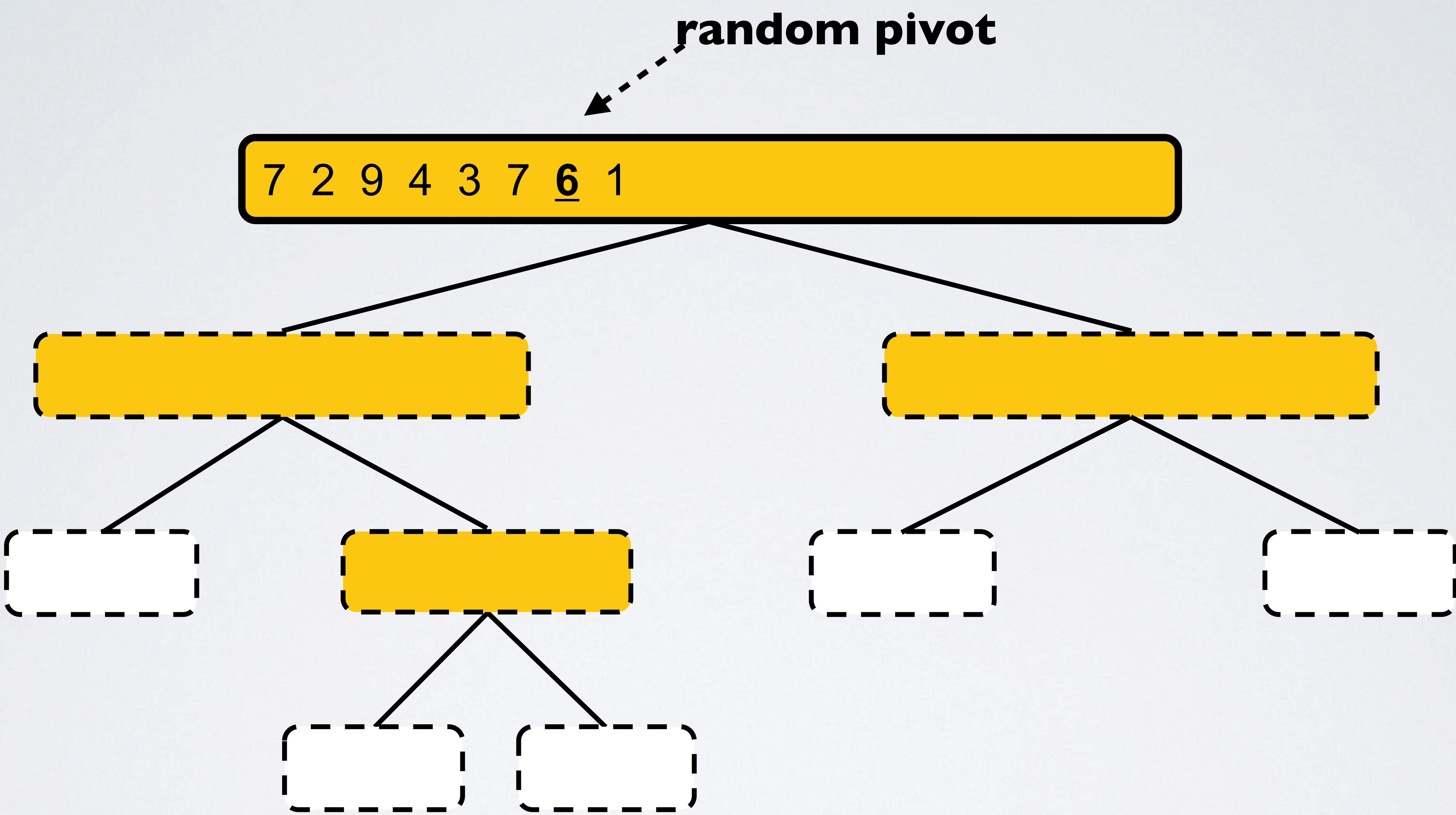
  - divide: pick random element (called pivot) and partition set into
    - **L:** elements less than x
    - **E:** elements equal to x
    - **G:** elements larger than x



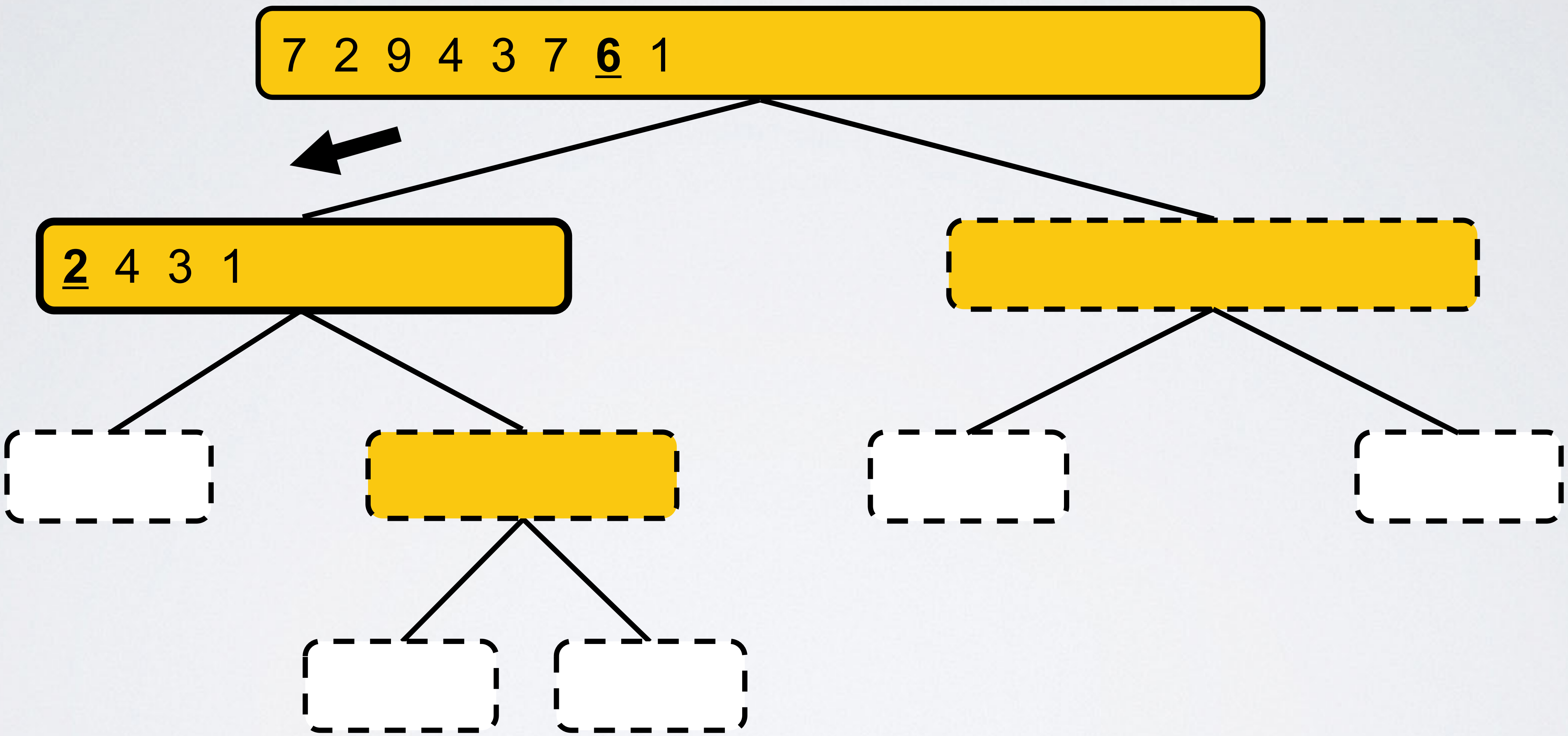
  - recur: quicksort L and G
  - conquer: join L, E and G



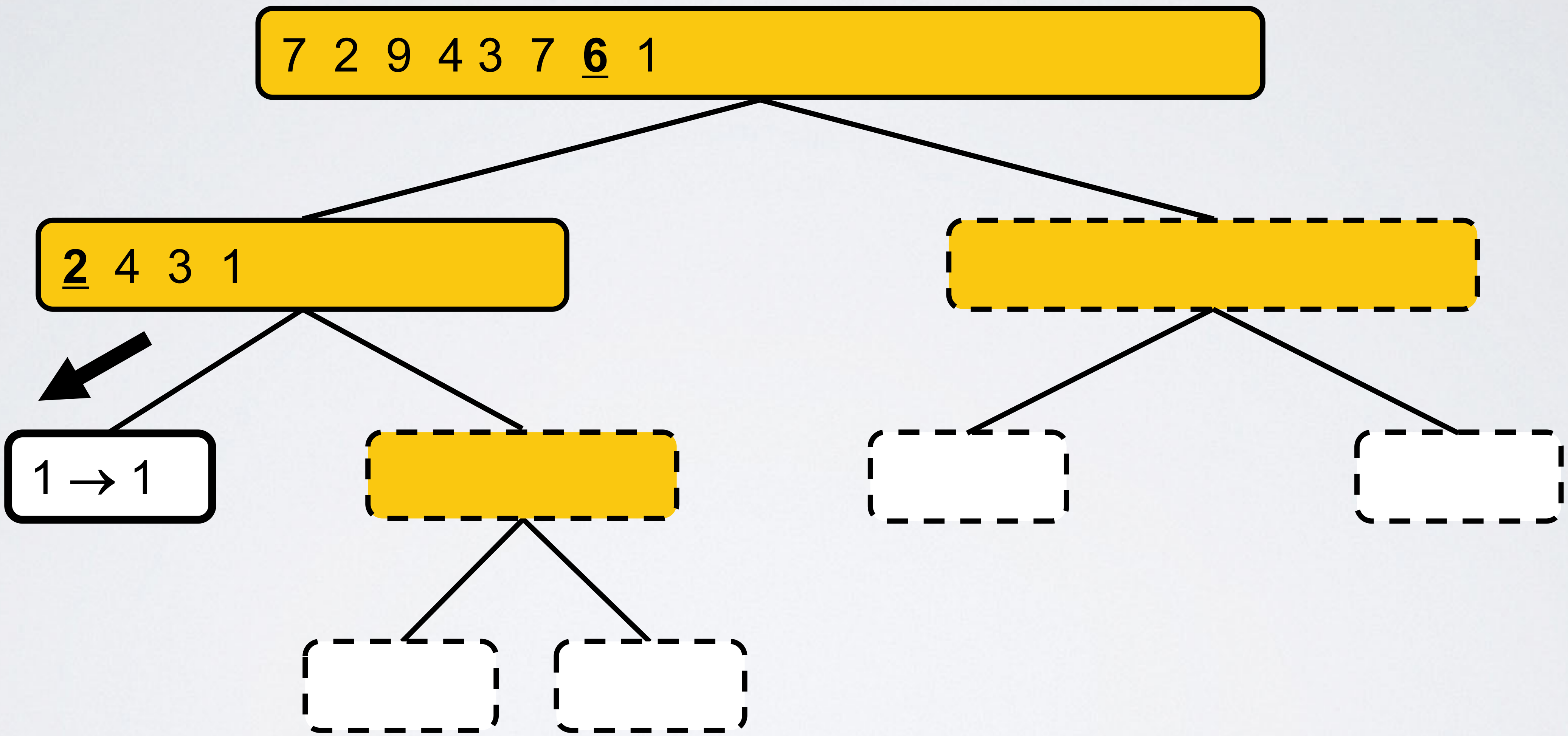
Quicksort Example



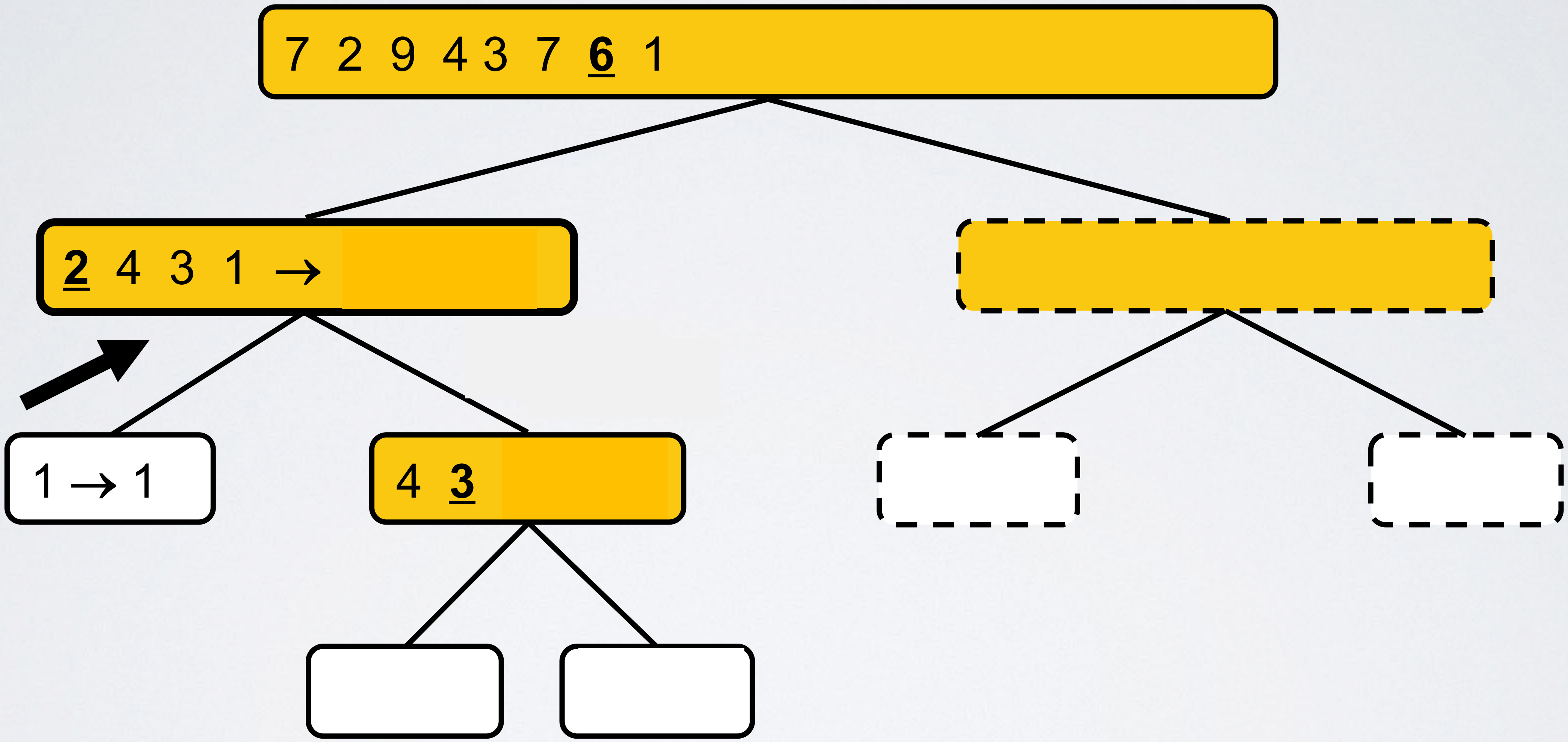
Quicksort Example



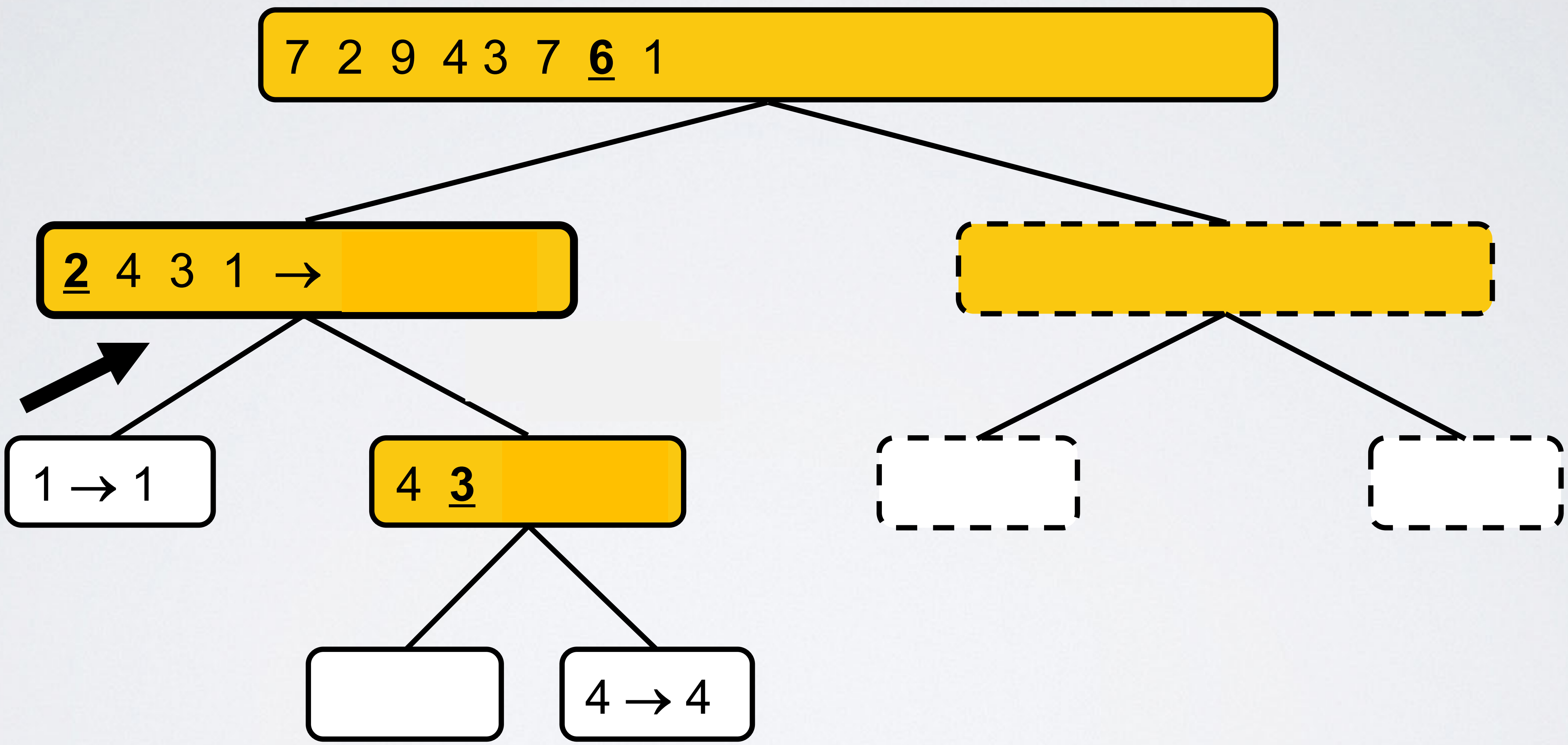
Quicksort Example



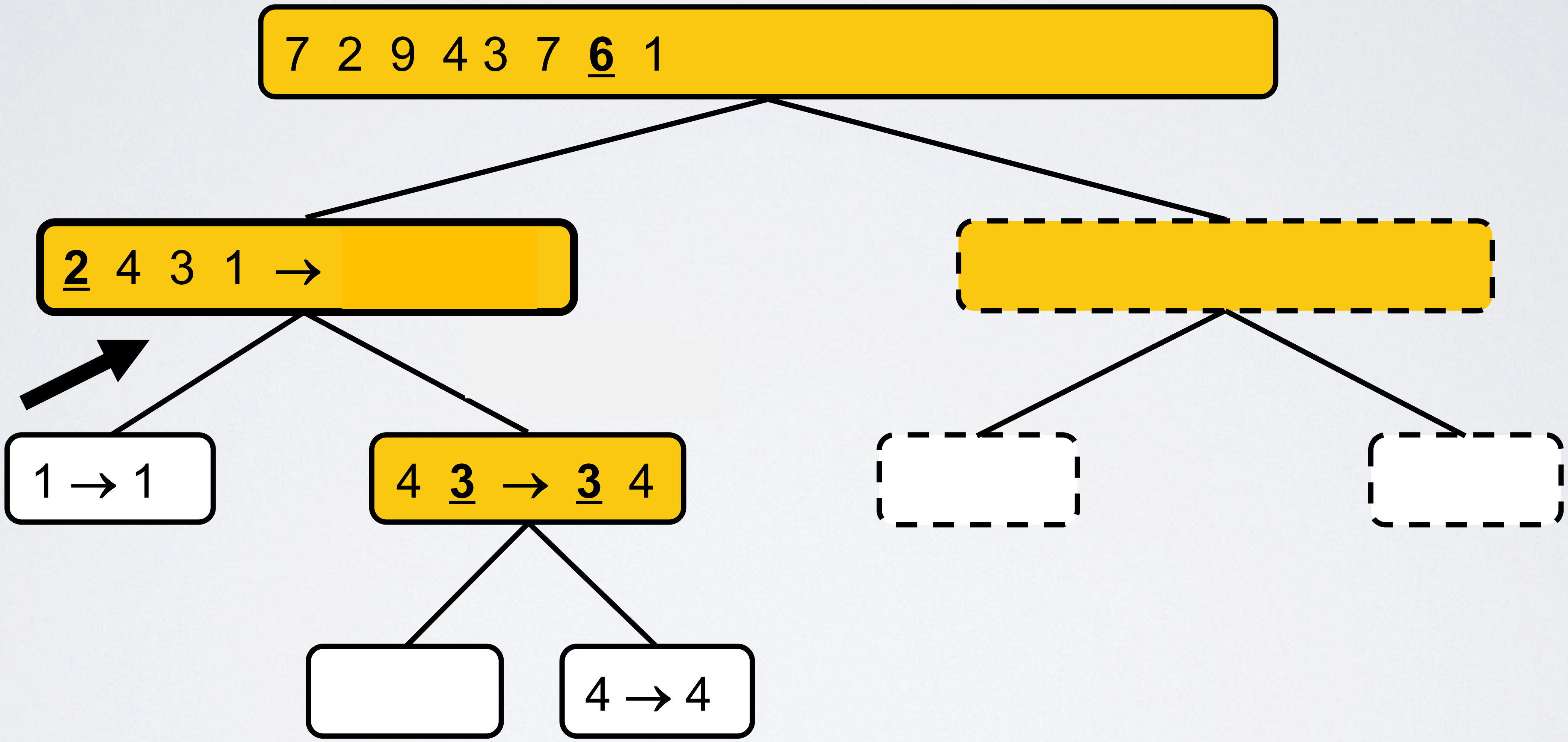
Quicksort Example



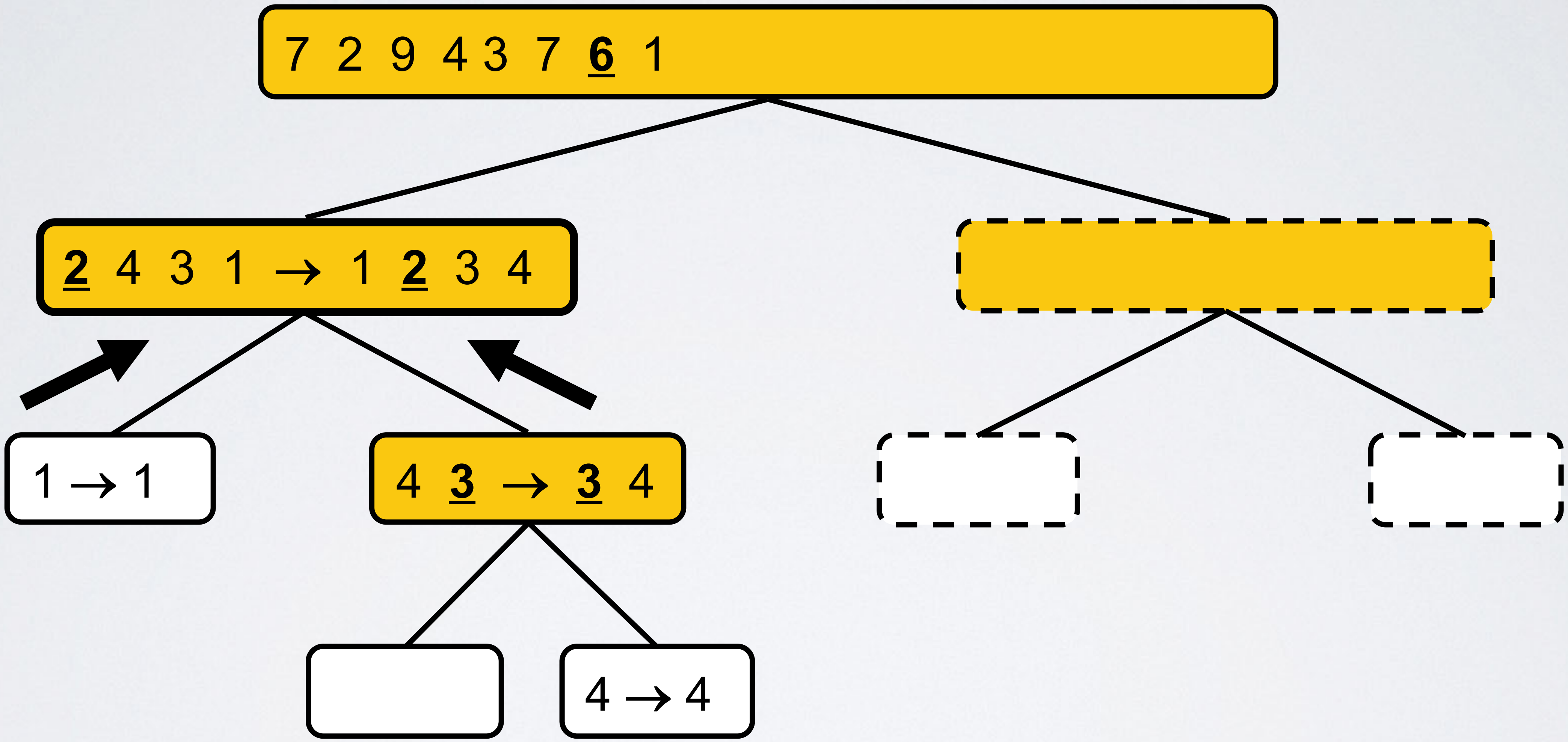
Quicksort Example



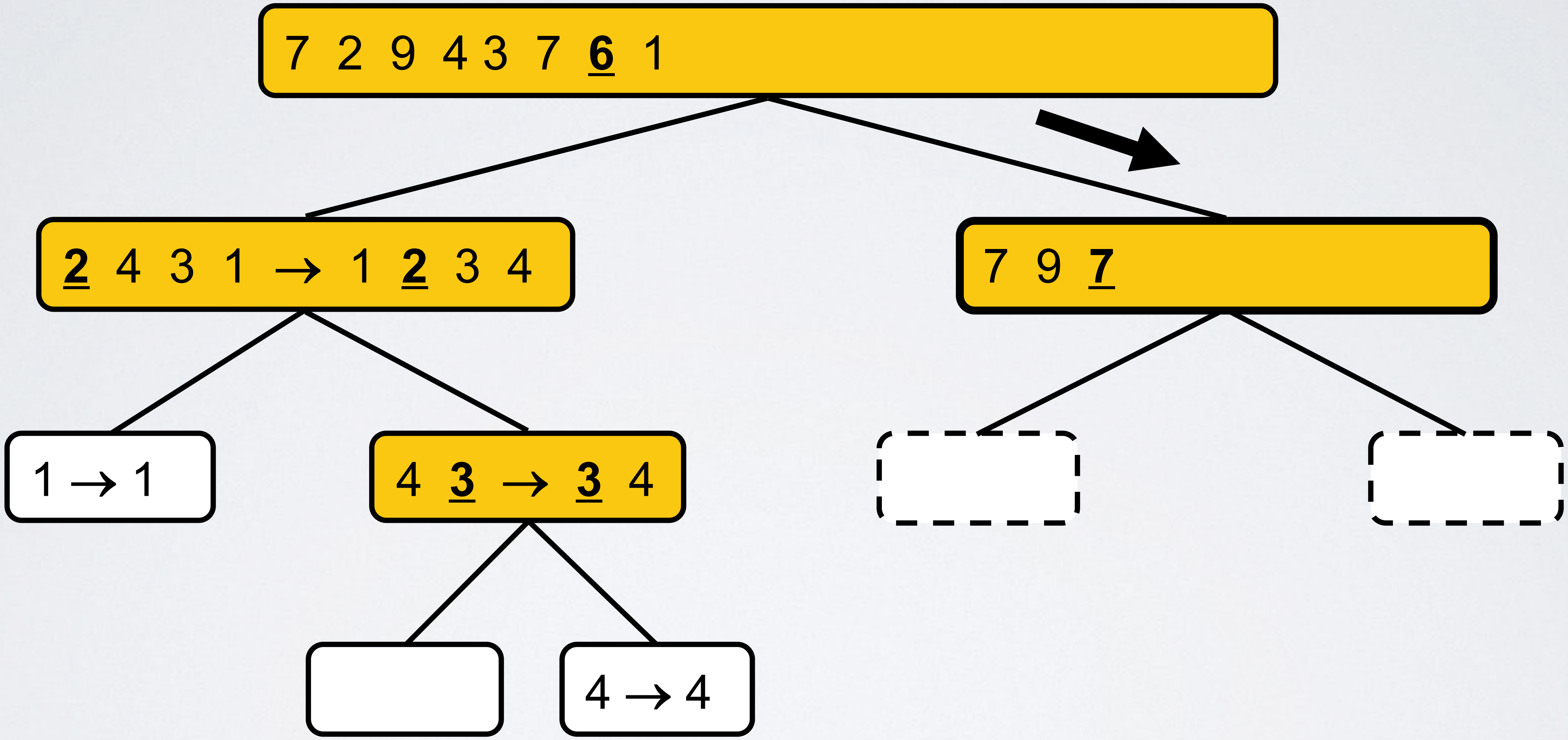
Quicksort Example



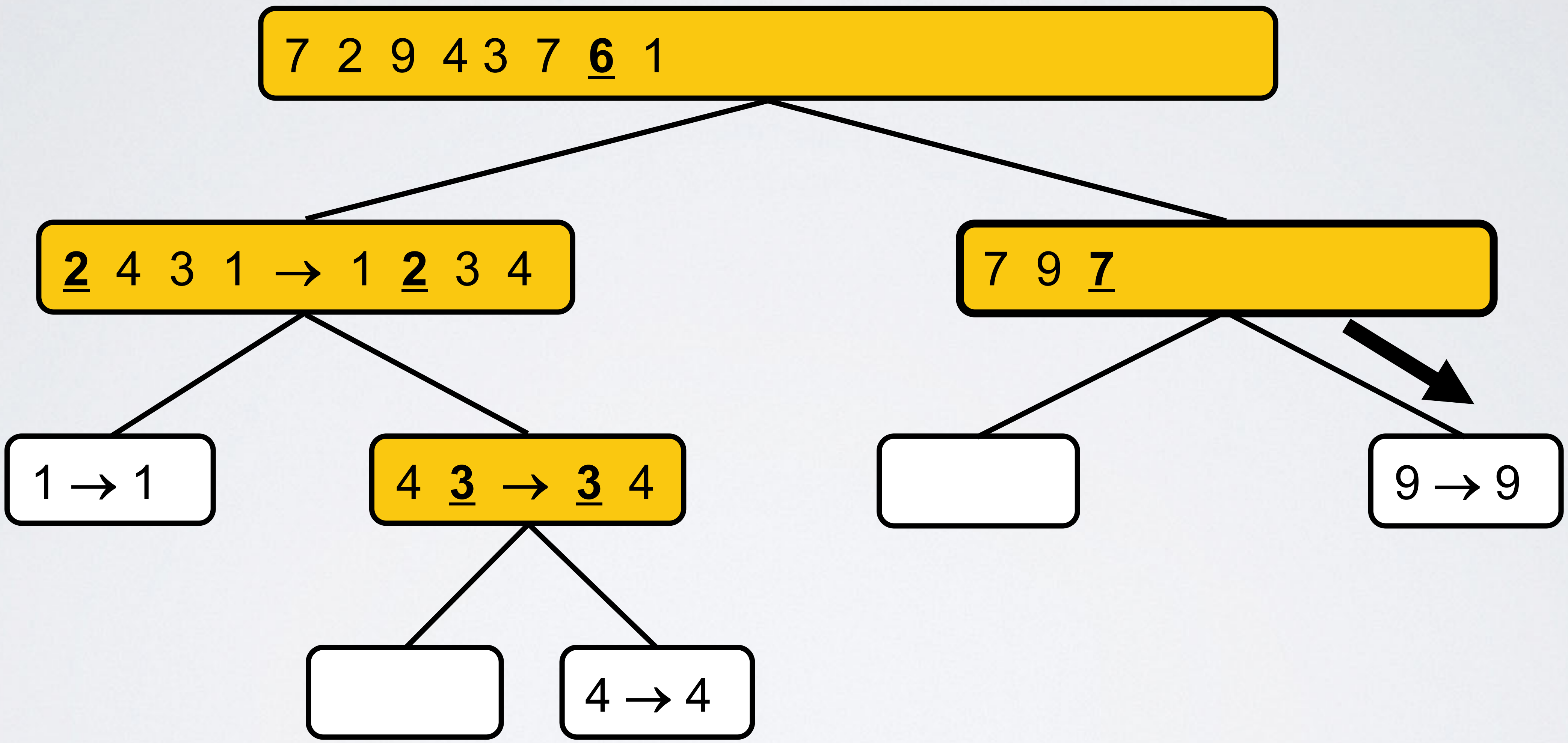
Quicksort Example



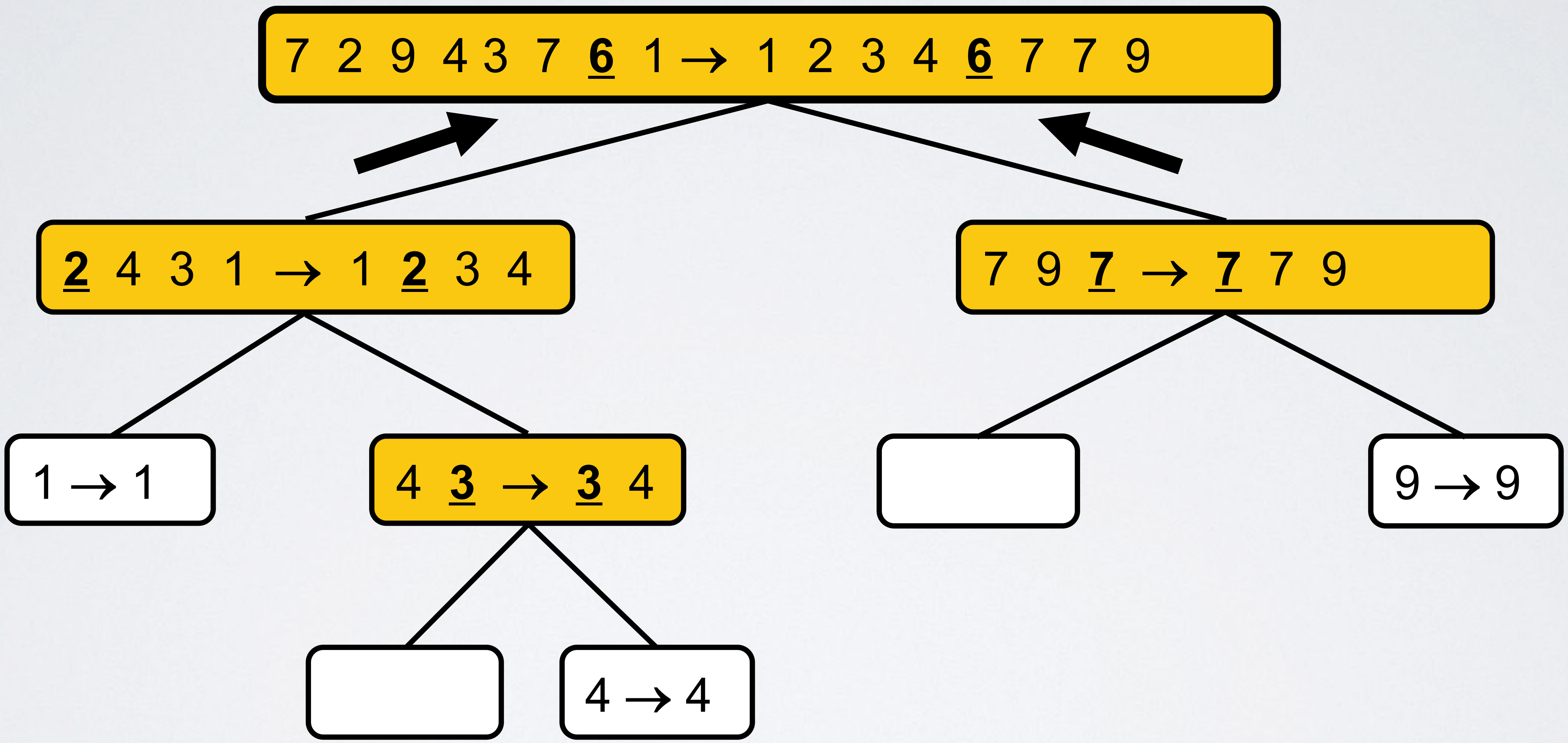
Quicksort Example



Quicksort Example



Quicksort Example



In-Place Quicksort

```
function partition(A, low, high, key_func):
  pivotIndex = random index between low and high
  pivotValue = A[pivotIndex]
  swap(A, pivotIndex, high) # move pivot to end
  currIndex = low
  for i from low to high – 1:
    if key_func(A[i], pivotValue) < 0:
      swap(A, i, currIndex)
        currIndex++
  swap(A, currIndex, high)   # move the pivot back
  return currIndex
```

In-Place Quicksort

```
function quicksort(A, low, high, key_func):
  if low < high:
    pivotIndex = partition(A, low, high, key_func)
    quicksort(A, low, pivotIndex – 1, key_func)
    quicksort(A, pivotIndex + 1, high, key_func)
```

Merge Sort vs. Quicksort

- Merge sort is worst-case `O(n log n)`
- Quicksort is expected `O(n log n)`
- Which is better?
- In practice quicksort is faster!
  - it also uses less space
  - constants are better

Non-Comparative Sorting

- Sorting functions are used on different types of inputs
  - Integers, floats, strings, arrays, other objects…
  - As long as we can compare the inputs we can use comparative sorting algorithms
- But for certain kinds of inputs, we can sometimes do better
  - example: for positive integers we can use Counting sort

Counting Sort

- Suppose that our input data comprises of integers between `m` to `n` (inclusive)
- Store array of counts of each item
- Pass through initial array, incrementing counts
- Iterate over count array to construct sorted array

Counting Sort

```
function counting_sort(A, m, n):
  counts = create_array(n - m + 1)
  fill_array(counts, 0)
  for x in A:
    counts[x - m] += 1
  j = 0
  for i in 0 to (n - m + 1):
    while counts[i] > 0:
      A[j] = i + m
      j += 1
      counts[i] -= 1
```

Bucket Sort

- A variant of counting sort
- For any given item, the key field or return value of the key function is between 0 and n
- Operate similar to counting sort, except instead of incrementing a count, we append to a list or dynamic array

Bucket Sort

```
function bucket_sort(A, m, n, key_func):
  buckets = create_array(n - m + 1)
  fill_array(buckets, [])
  for x in A:
    buckets[key_func(x) - m].append(x)
  j = 0
  for i in 0 to (n - m + 1):
    while not is_empty(buckets[i]):
      A[j] = buckets[i].pop(0)
      j += 1
```

Order Statistics

- Order statistics: revolve around finding the nth smallest or largest element in a set
  - Example: median
- Naive approach:
  - Sort the elements of the set
  - Retrieve nth smallest/largest entry by random access
  - **`O(nlogn)`**
  - Can we do better?

Partition revisited

- Remember core idea of quick sort:
  - Find global placement of element, sort before the element and then after the element
  - Combine results

Partition revisited

- Remember core idea of quick sort:
  - Find **global placement** of element, sort before the element and then after the element
  - Combine results

Partition returns global placement
We can use partition function!

Intuition

- Use partition on array representation of set
- get placement of first pivot
- If that placement is the nth slot, then we finish :-)
  - Else, we need to keep looking…
  - Start searching after placement if placement is before n
  - Start searching before placement if placement is after n

Find the $\mathbf{n}^{\mathtt{th}}$ smallest item

```
function smallest(A, n, key_func):
  lo = 0
  hi = length(A) - 1
  p = partition(A, lo, hi, key_func)
  while (p != (n - 1)):
    if p < (n - 1):
      lo = p + 1
    else:
      hi = p - 1
    p = partition(A, lo, hi, key_func)
  return A[n - 1]
```

Find the n^{th} smallest item

How would you adjust to find the n^{th} largest

```
function smallest(A, n, key_func):
  lo = 0
  hi = length(A) - 1
  p = partition(A, lo, hi, key_func)
  while (p != (n - 1)):
    if p < (n - 1):
      lo = p + 1
    else:
      hi = p - 1
    p = partition(A, lo, hi, key_func)
  return A[n - 1]
```