# Sorting & Master Theorem

CS16: Introduction to Data Structures & Algorithms

Spring 2019

# Outline

- Motivation
- Quadratic Sorting
  - Selection sort
  - Insertion sort
- Linearithmic Sorting
  - Merge Sort
  - Master Theorem
  - Quick Sort
- Comparative sorting lower bound
- Linear Sorting
  - Radix Sort

# The Problem

- 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

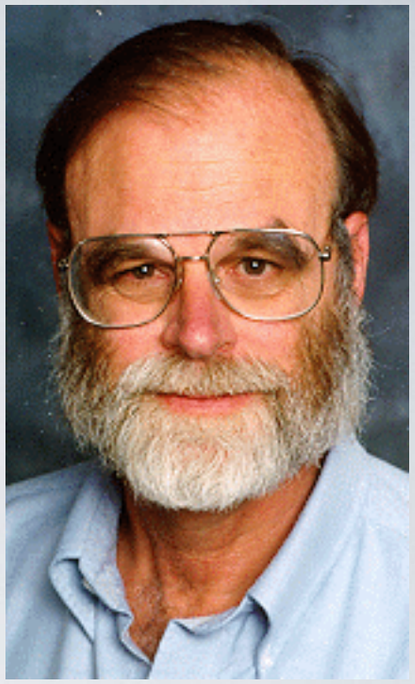
# Sorting is Serious!

## 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
- In-place algorithms
  - transform data structure w/ small amount of extra storage (i.e., `O(1)`)
  - For sorting: array is overwritten by output instead of creating new array
- Most sorting algorithms in 16 are comparison-based
  - main operation is comparison
  - but not all (e.g., Radix sort)

# "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 an efficiency

# Outline

- Motivation
- Quadratic Sorting
  - **Selection sort**
  - Insertion sort
- Linearithmic Sorting
  - Merge Sort
  - Master Theorem
  - Quick Sort
- Comparative sorting lower bound
- Linear Sorting
  - Radix Sort

# Selection Sort

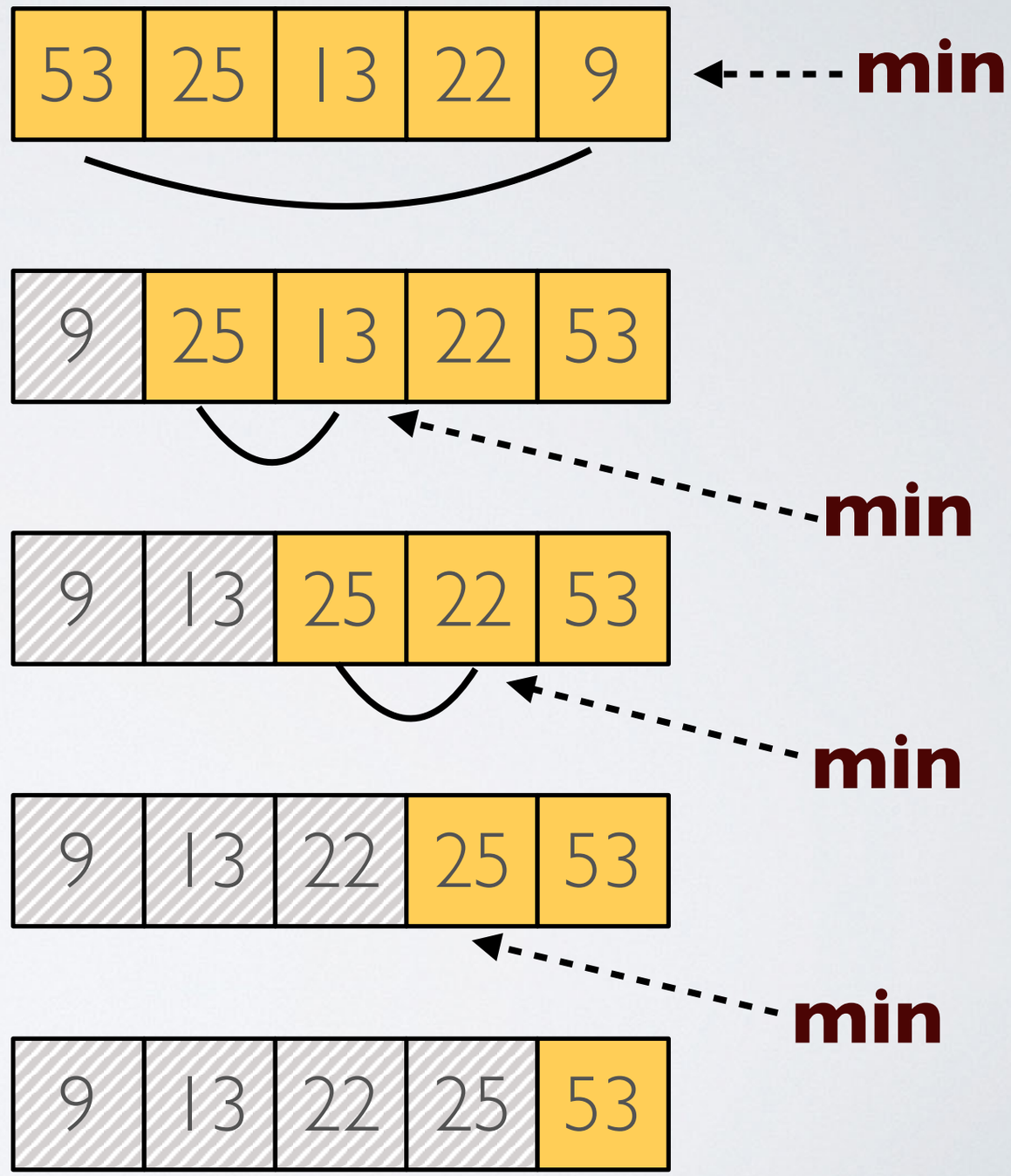
- Usually iterative and in-place
- Divides input array into two logical parts
  - elements already sorted
  - elements that still need to be sorted
- Selects smallest element & places it at index **0**
  - then selects second smallest & places it in index **1**
    - then the third smallest at index **2**, etc..

# Selection Sort

- Advantages
  - Very simple
  - Memory efficient if in-place (swaps elements in array)
- Disadvantages
  - Slow: $O(n^2)$

# Selection Sort

- Iterate through positions
- At each position
  - store smallest element from remaining set



# Selection Sort

```
function selection_sort(A):
   n = A.length
   for i = 0 to n-2:
      min = argmin(A[i:n-1])
      swap A[i] with A[min]
```

# Outline

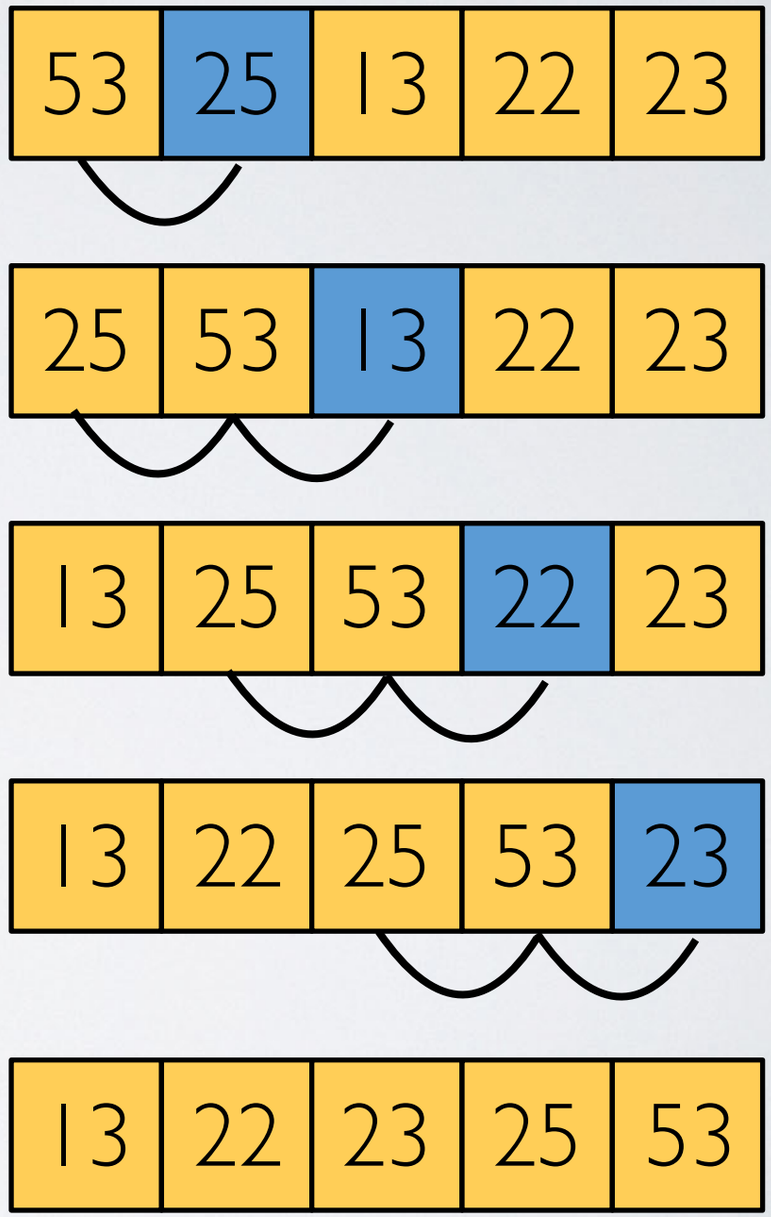
- Motivation
- Quadratic Sorting
  - Selection sort
  - **Insertion sort**
- Linearithmic Sorting
  - Merge Sort
  - Master Theorem
  - Quick Sort
- Comparative sorting lower bound
- Linear Sorting
  - Radix Sort

# Insertion Sort

- Usually iterative and in-place
- Compares each item w/ all items before it…
  - …and inserts it into correct position
- Advantages
  - Works really well if items partially sorted
  - Memory efficient if in-place (swaps elements in array)
- Disadvantages
  - Slow: $O(n^2)$

# Insertion Sort

- Compares each item w/ all items before it…
  - …and inserts it into correct position




**Note:** 23 > 22 so don't need to keep checking since rest is already sorted

# Insertion Sort

```
function insertion_sort(A):
   n = A.length
   for i = 1 to n-1:
      for j = i down to 1:
         if a[j] < a[j-1]:
            swap a[j] and a[j-1]
         else:
            break  # out of the inner for loop
                   # this prevents double checking the
                   # already sorted portion
```

# Outline

- Motivation
- Quadratic Sorting
  - Selection sort
  - Insertion sort
- Linearithmic Sorting
  - **Merge Sort**
  - Master Theorem
  - Quick Sort
- Comparative sorting lower bound
- Linear Sorting
  - Radix Sort

# Divide & Conquer

- Algorithmic design paradigm
  - divide: divide input $S$ into disjoint subsets $S_1, \ldots, S_k$
  - recur: solve sub-problems on $S_1, \ldots, S_k$
  - conquer: combine solutions for $S_1, \ldots, S_k$ into solution for $S$
- Base case is usually sub-problem of size $1$ or $0$

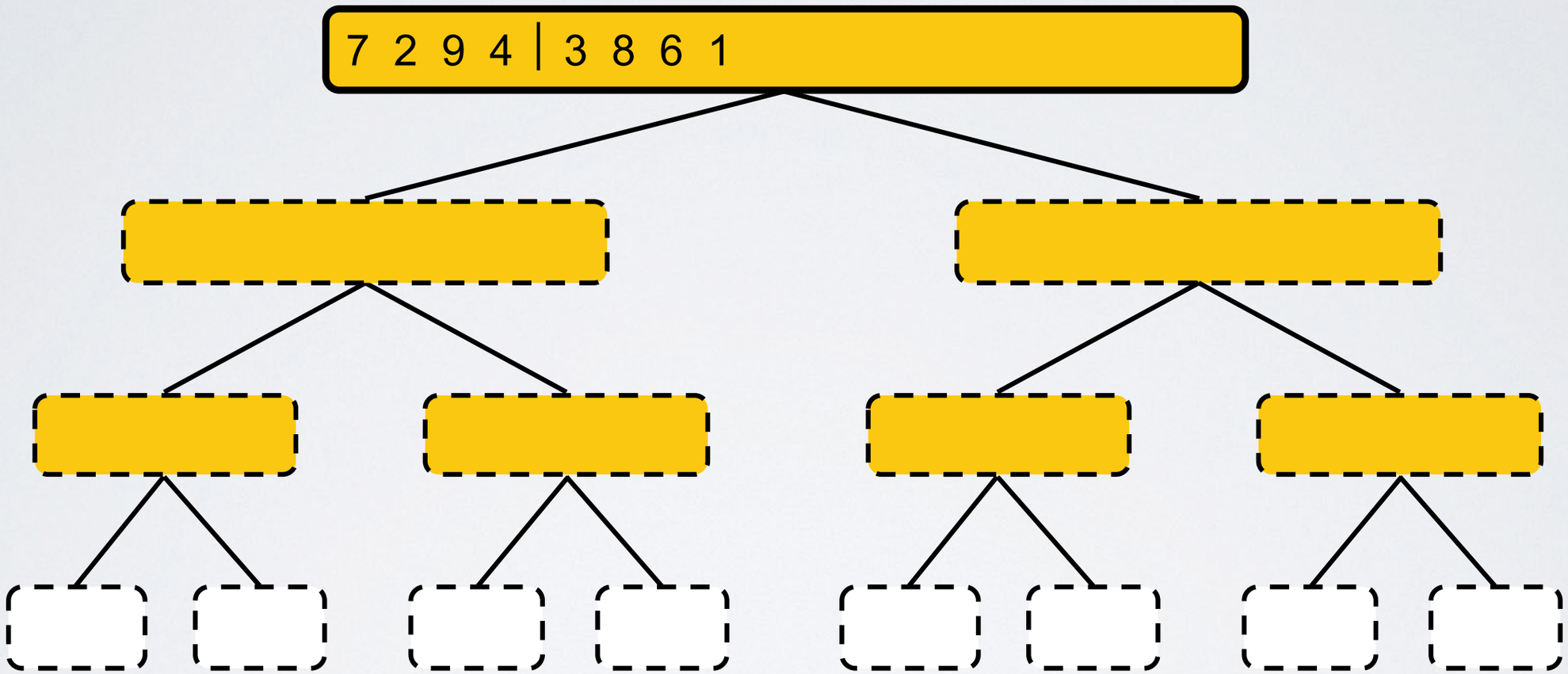
# Merge Sort

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

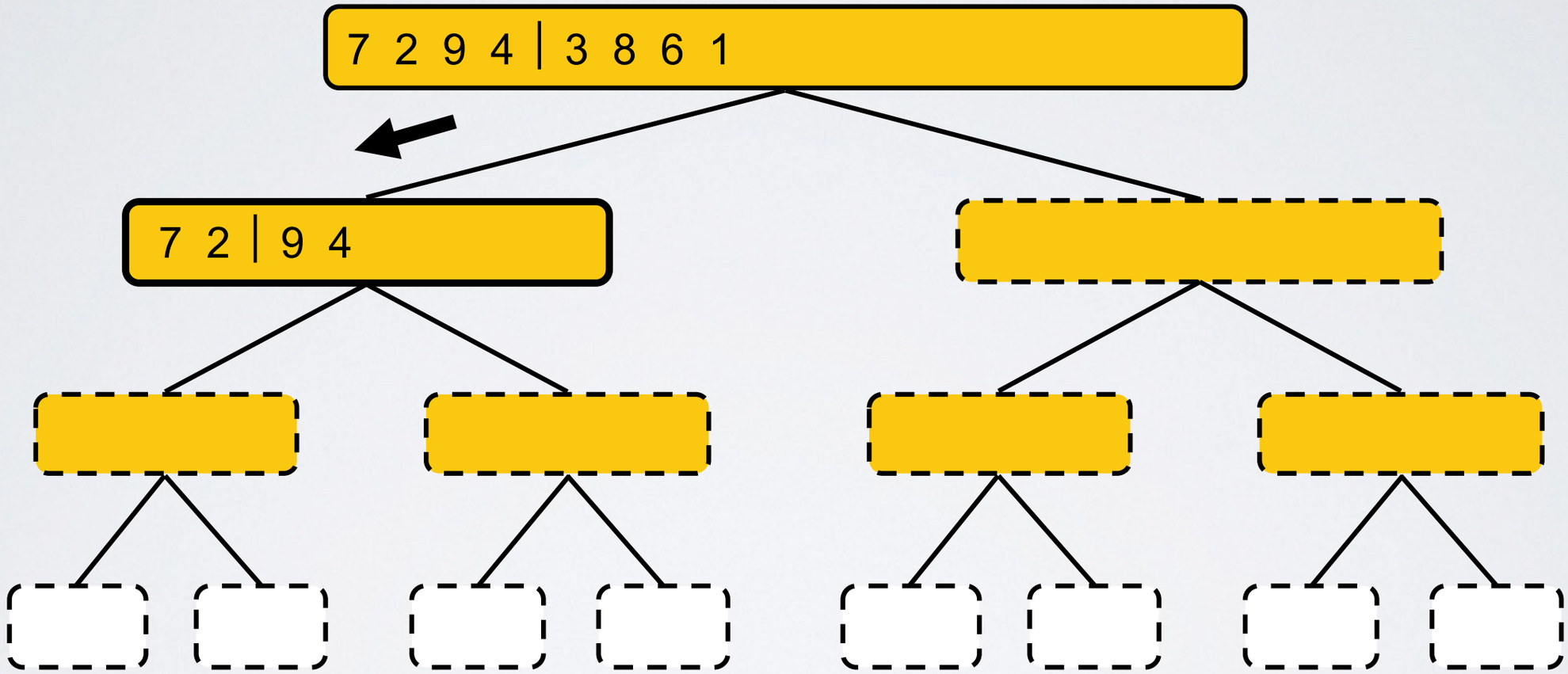
# Merge Sort

- Merge sort on n-element sequence S
  - divide: divide $\mathbf{S}$ into disjoint subsets $\mathbf{S}_1$ and $\mathbf{S}_2$
  - recur: recursively merge sort $\mathbf{S}_1$ and $\mathbf{S}_2$
  - conquer: merge $\mathbf{S}_1$ and $\mathbf{S}_2$ into sorted sequence
- Suppose we want to sort
  - `7,2,9,4,3,8,6,1`

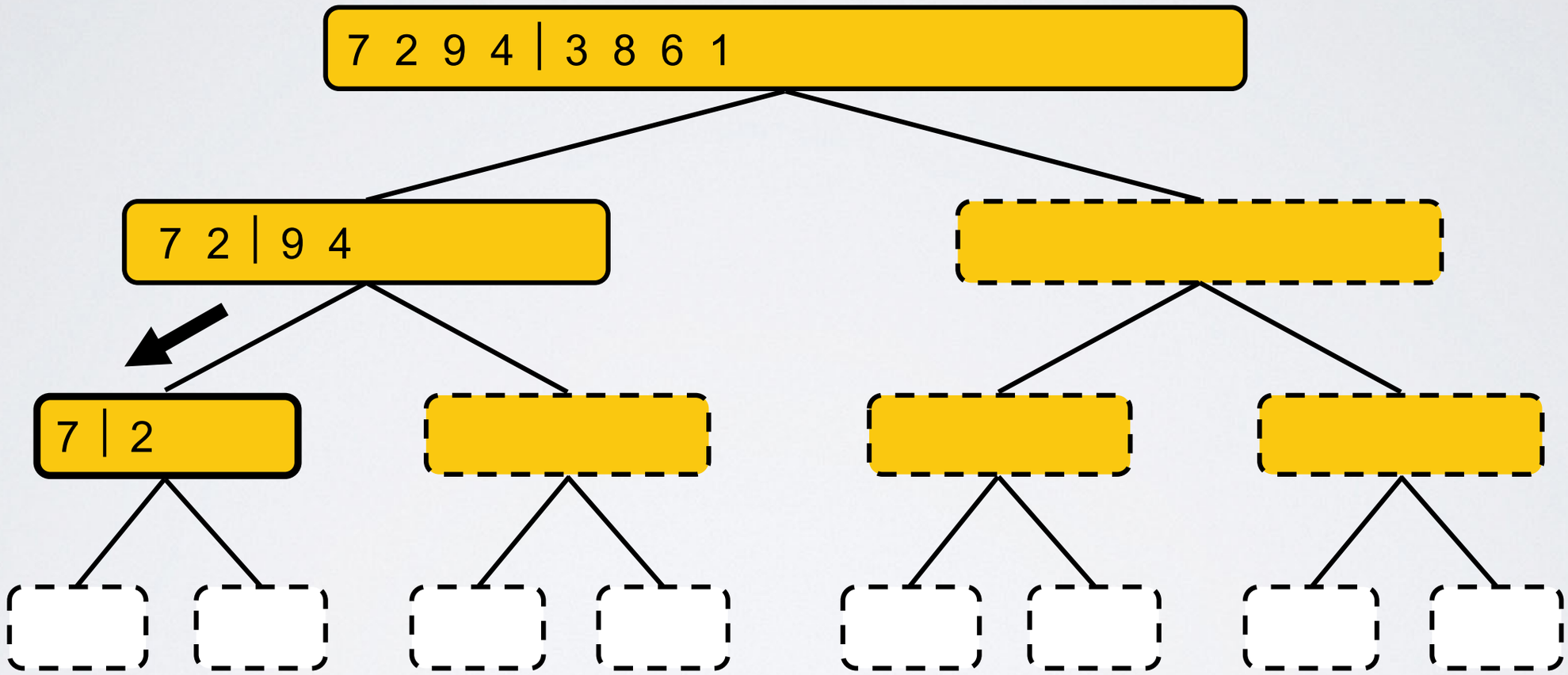
# Merge Sort Recursion Tree



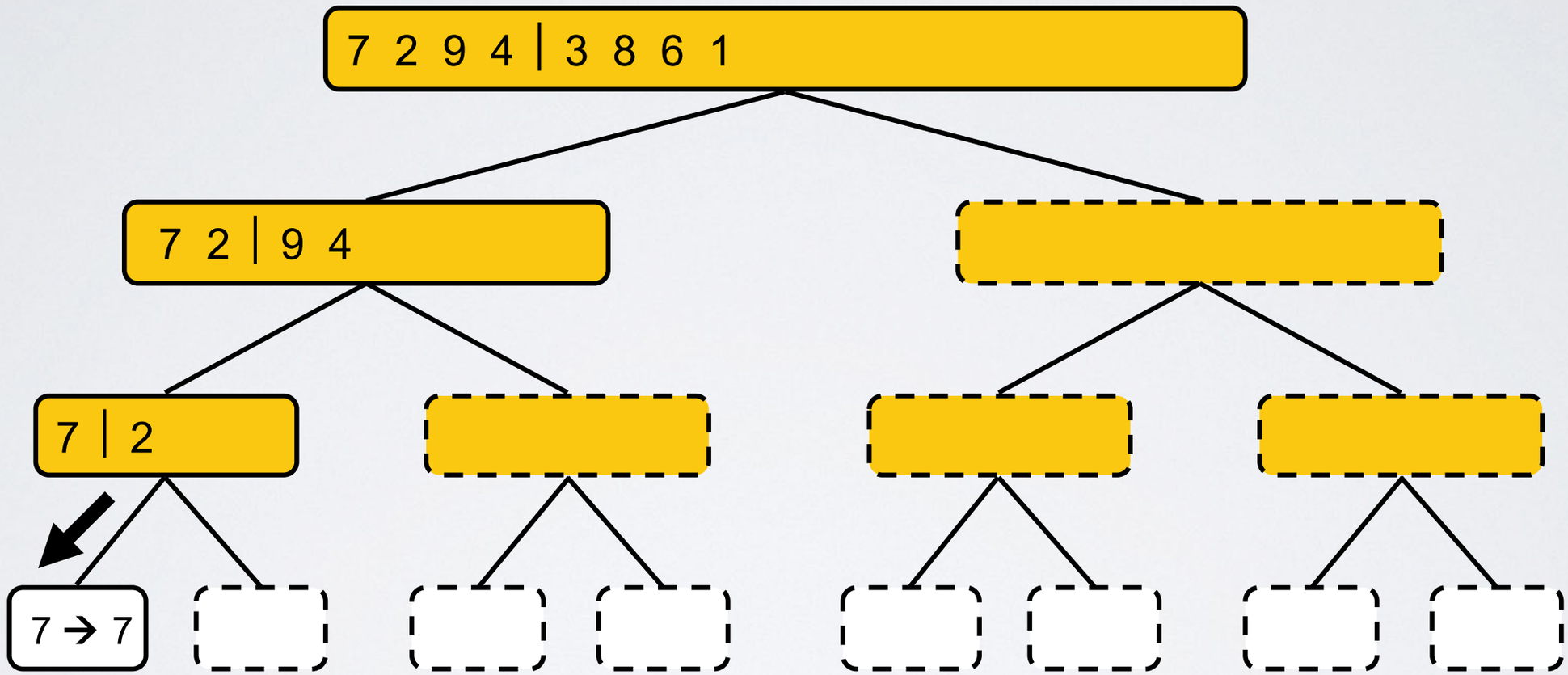
# Merge Sort Recursion Tree



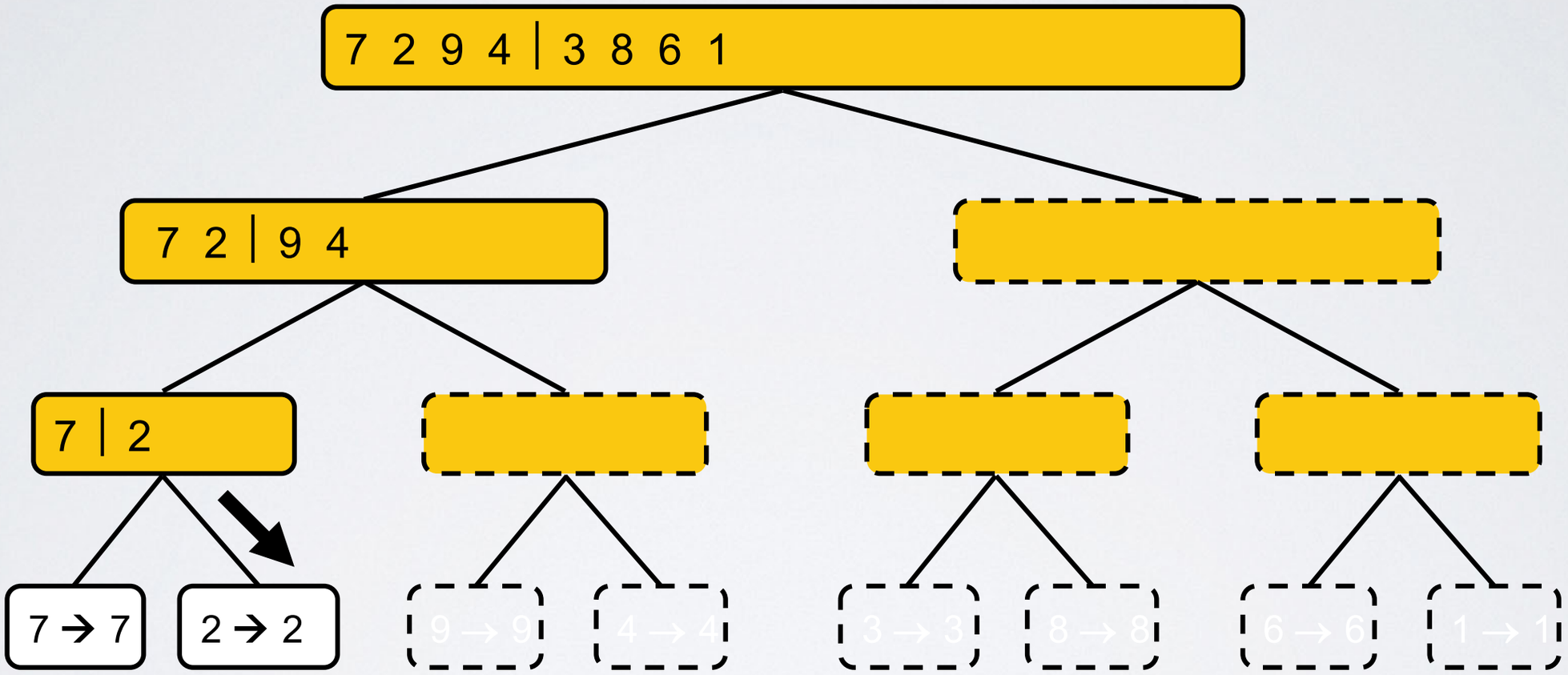
# Merge Sort Recursion Tree



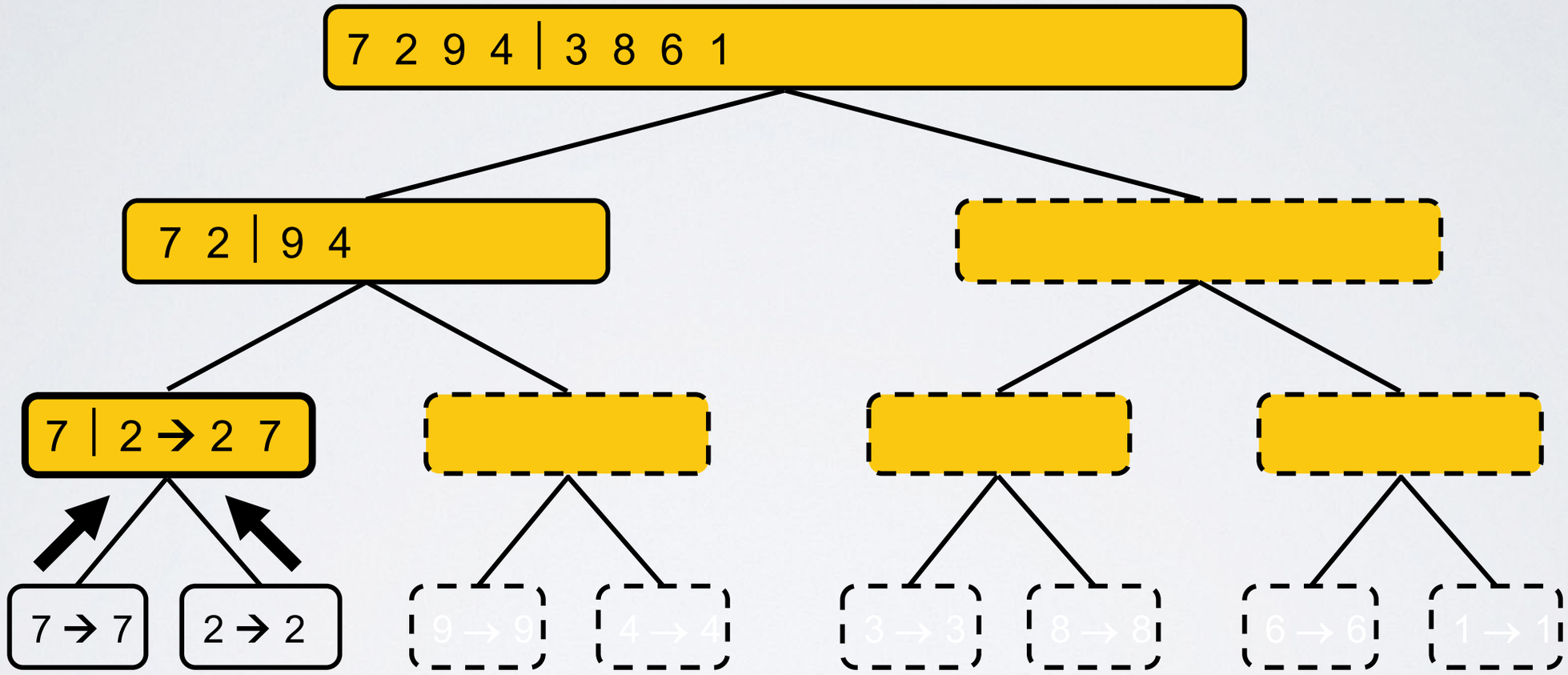
# Merge Sort Recursion Tree



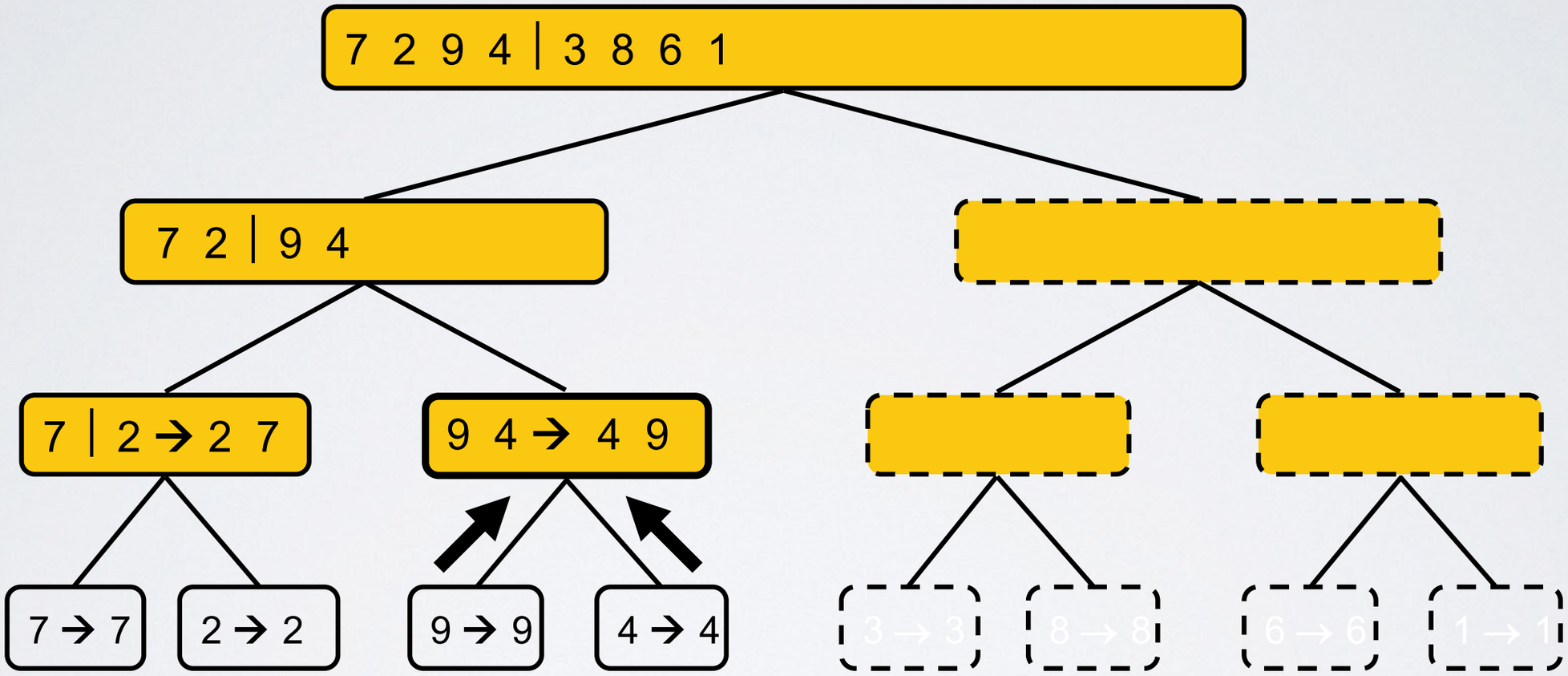
# Merge Sort Recursion Tree



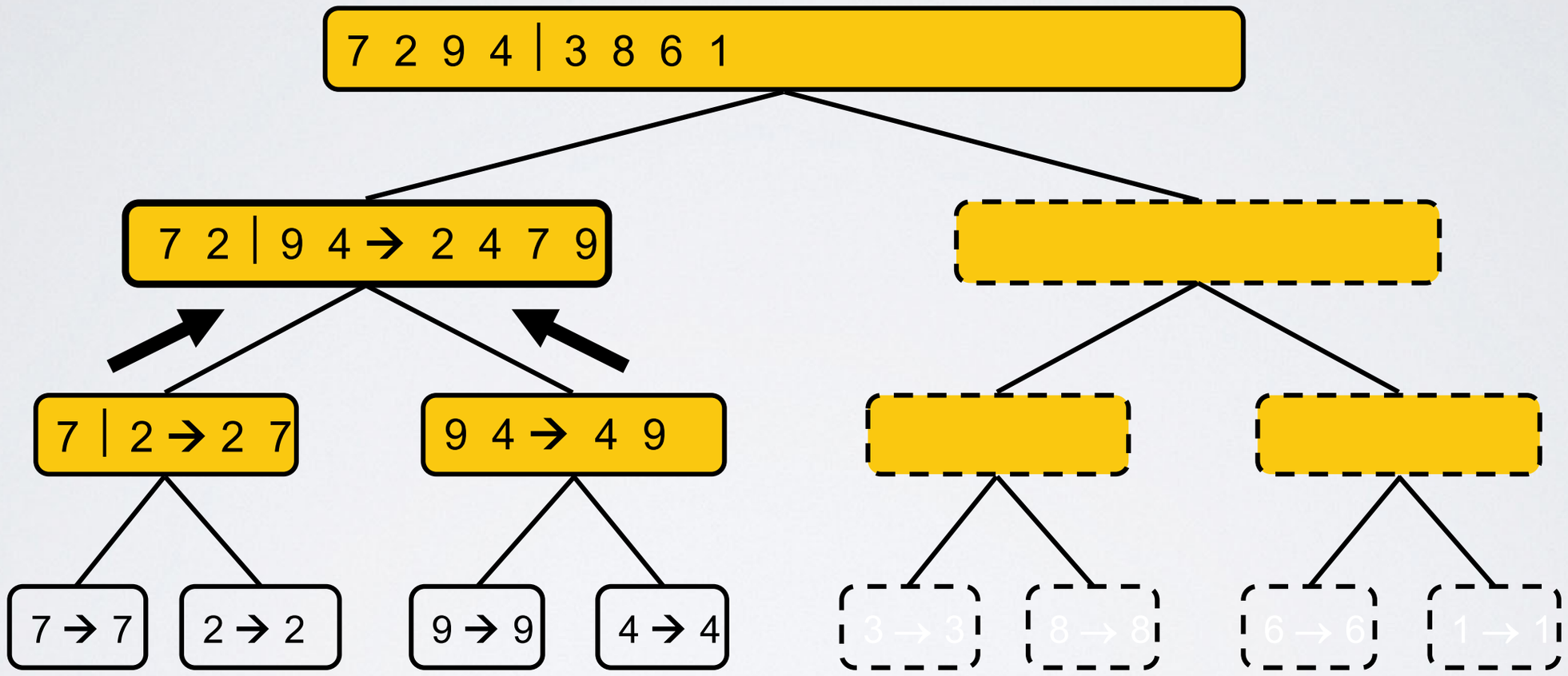
# Merge Sort Recursion Tree



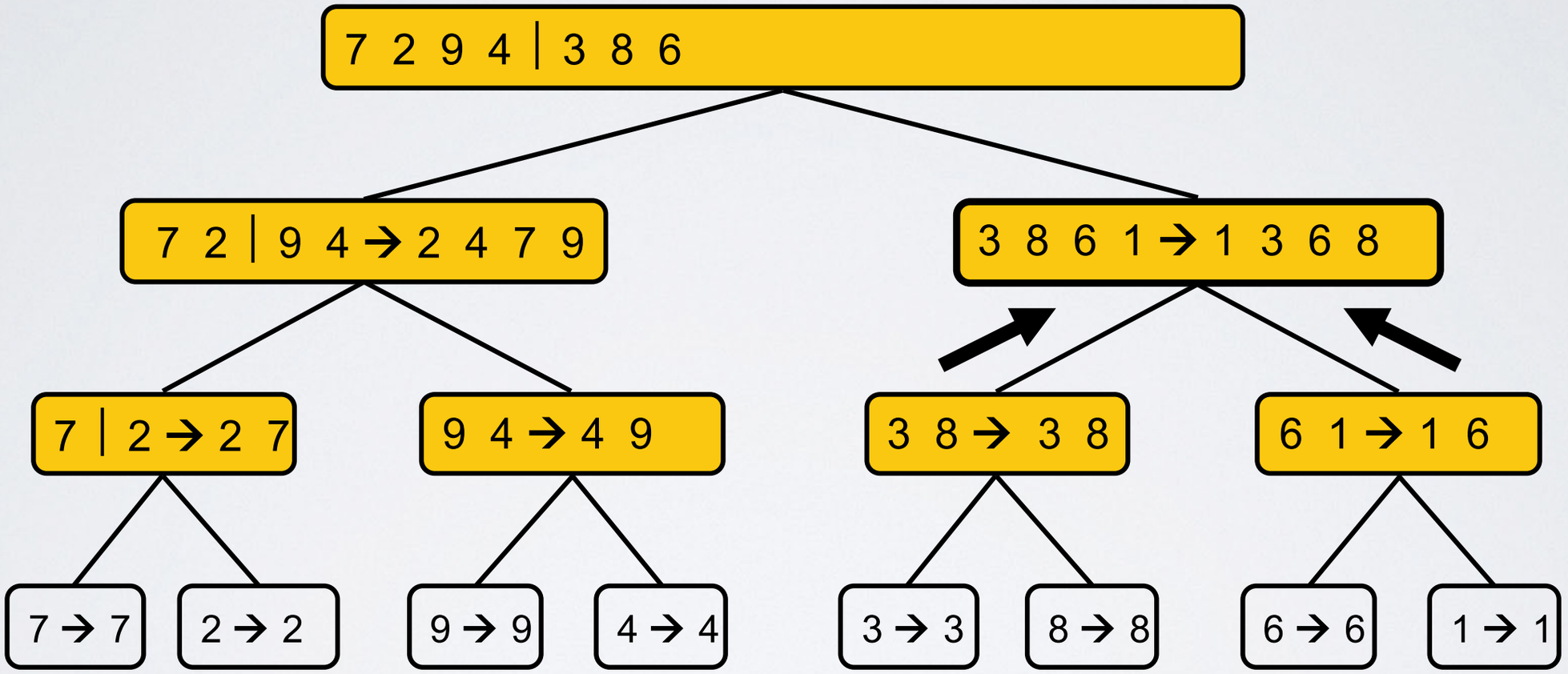
# Merge Sort Recursion Tree



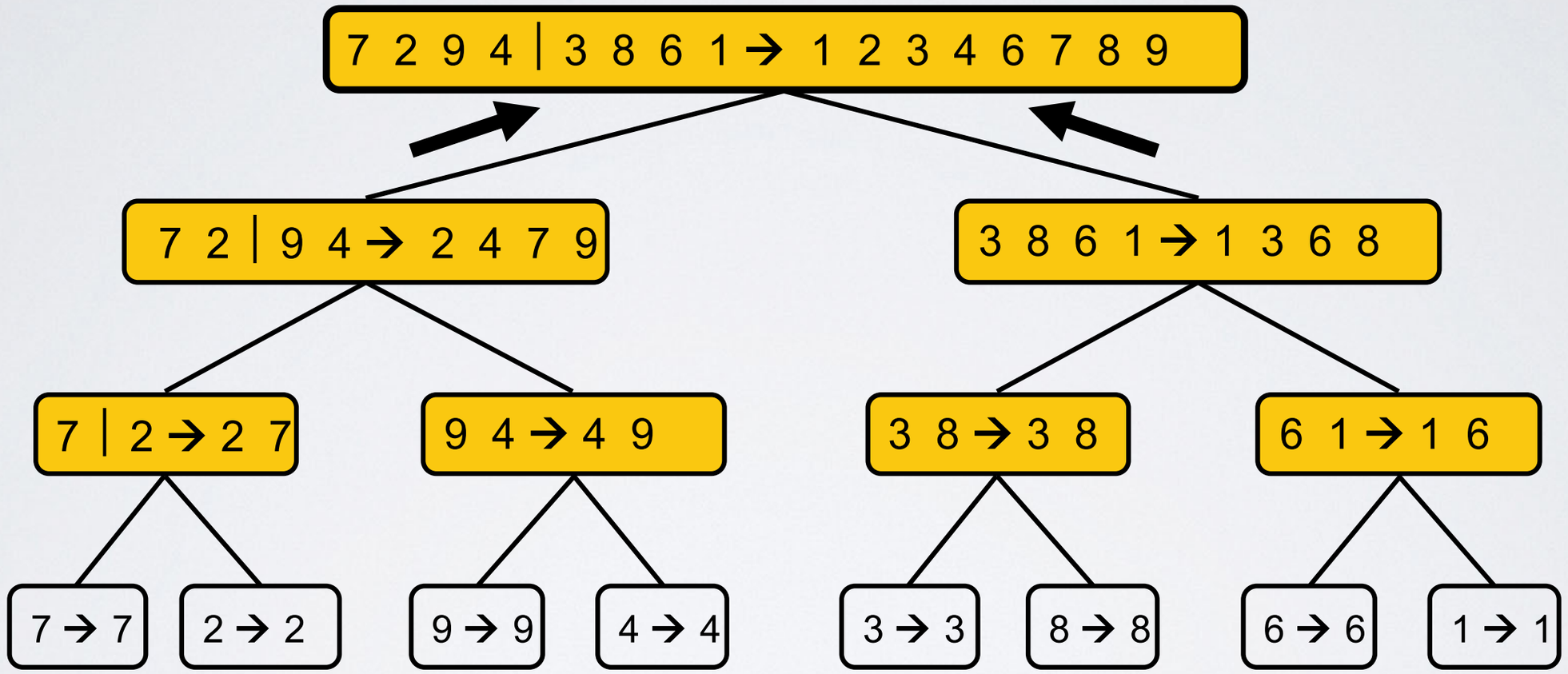
# Merge Sort Recursion Tree



# Merge Sort Recursion Tree



# Merge Sort Recursion Tree



# Merge Sort Pseudo-Code

```
function mergeSort(A):
   n = A.length
   if n <= 1:
      return A

   mid = n/2
   left = mergeSort(A[0...mid-1])
   right = mergeSort(A[mid...n-1])
   return merge(left, right)
```

# Merge Sort Pseudo-Code

```
function merge(A, B):
   result = []
   aIndex = 0
   bIndex = 0
   while aIndex < A.length and bIndex < B.length:
      if A[aIndex] <= B[bIndex]:
         result.append(A[aIndex])
         aIndex++
      else:
         result.append(B[bIndex])
         bIndex++
   if aIndex < A.length:
      result = result + A[aIndex:end]
   if bIndex < B.length:
      result = result + B[bIndex:end]
   return result
```

# Merge Sort

**Activity #1**

*2 min*

# Merge Sort

**Activity #1**

*2 min*

# Merge Sort

**Activity #1**

*1 min*

# Merge Sort

**Activity #1**

*0 min*

# Merge Sort Recurrence Relation

- Merge sort steps
  - Recursively merge sort left half
  - Recursively merge sort right half
  - Merge both halves
- **`T(n)`**: time to merge sort input of size n
  - **`T(n)`** = step **`1`** + step **`2`** + step **`3`**
  - Steps **`1`** & **`2`** are merge sort on half input so **`T(n/2)`**
  - Step **`3`** is **`O(n)`**

# Merge Sort Recurrence Relation

- General case

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

- Base case

$$T(1) = c$$

# Merge Sort Recurrence Relation

- Plug & chug

$$T(1) = c_1$$
$$T(2) = 2 \cdot T(1) + 2 = 2c_1 + 2$$
$$T(4) = 2 \cdot T(2) + 4 = 2(2c_1 + 2)4 = 4c_1 + 8$$
$$T(8) = 2 \cdot T(4) + 8 = 2(4c_1 + 8) + 8 = 8c_1 + 24$$
$$T(16) = 2 \cdot T(8) + 16 = 2(8c_1 + 24) + 16 = 16c_1 + 64$$

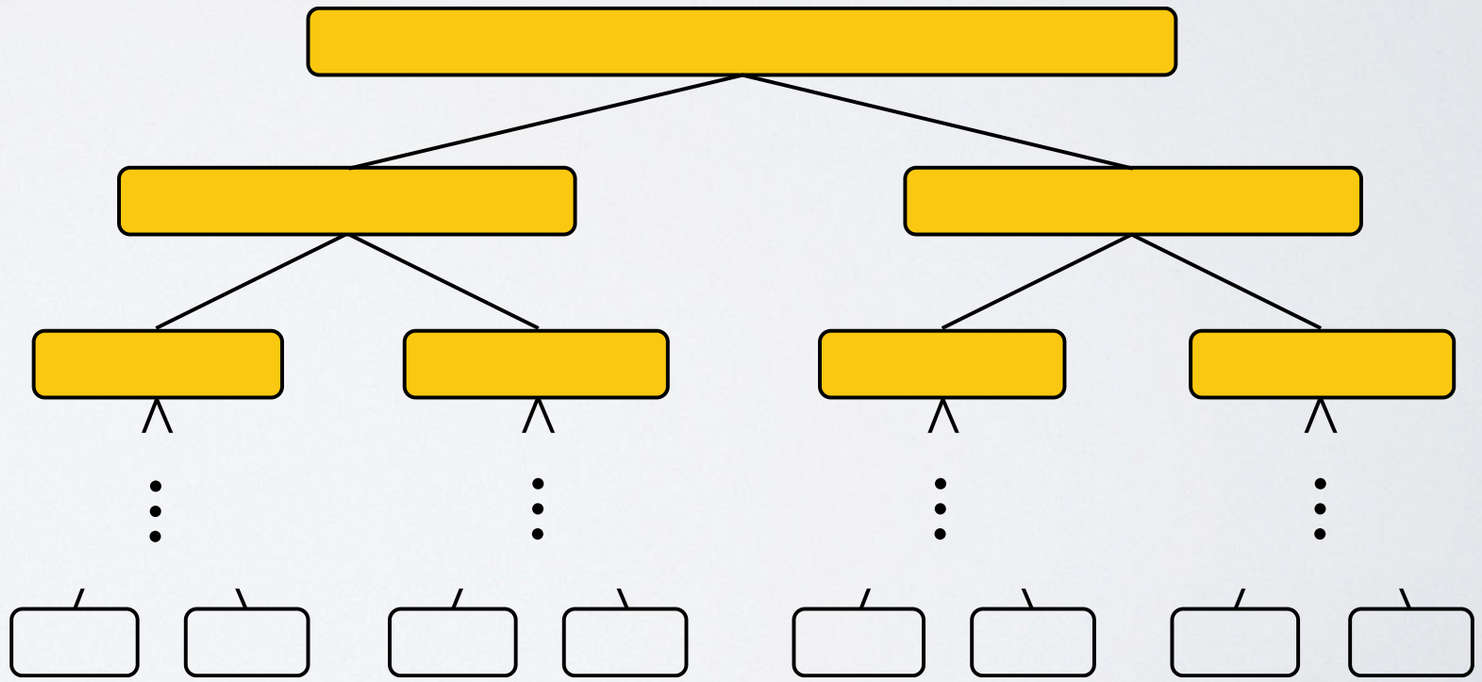
- Solution

$$T(n) = nc_1 + n \log n = O(n \log n)$$

# Analysis of Merge Sort

- Merge sort recursive tree is perfect binary tree so has height `O(log n)`
- At each depth `k`: need to merge $2^{k+1}$ sequences of size $n/2^{k+1}$
  - work at each depth is `O(n)`

| depth | sequenc | size |
|---|---|---|
| 0 | 2 | $n/2$ |
| 1 | 4 | $n/4$ |
| 2 | 8 | $n/4$ |
| ⋮ | ⋮ | ⋮ |
| k | $2^{k+1}$ | $n/2^{k+1}$ |



# Analysis of Merge Sort

- To determine that Merge sort was **`O(nlog n)`**
  - Use plug and chug to guess a solution
  - Prove that **`O(n log n)`** is correct (e.g., using induction)
- Can be a lot of work

# Outline

- Motivation
- Quadratic Sorting
  - Selection sort
  - Insertion sort
- Linearithmic Sorting
  - Merge Sort
  - **Master Theorem**
  - Quick Sort
- Comparative sorting lower bound
- Linear Sorting
  - Radix Sort

# The Master Theorem

- Solves large class of recurrence relations
  - we will learn how to use it but not its proof
  - See Dasgupta et al. p. 58-60 for proof

- Let T(n) be a monotonically-increasing function of form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^d)$$

  - **a**: number of sub-problems
  - **n/b**: size of each sub-problem
  - **$n^d$**: work to prepare sub-problems & combine their solutions

# The Master Theorem

- If $a \geq 1$, $b > 1$, $d \geq 0$, then
  - if $a < b^d$ then $T(n) = \Theta(n^d)$
  - if $a = b^d$ then $T(n) = \Theta(n^d \log n)$
  - if $a > b^d$ then $T(n) = \Theta(n^{\log_b a})$

- Applying Master Theorem to merge sort
  - Recurrence relation of merger sort: $T(n) = 2T(n/2) + O(n^1)$
  - $a=2$, $b=2$ and $d=1$ so $a = b^d$
  - and $T(n) = \Theta(n^d \log n)$
    $= \Theta(n^1 \log n)$
    $= \Theta(n \log n)$

# Master Theorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^d)$$

- `T(n) = Θ(n^d)` if `a<b^d`
- `T(n) = Θ(n^d log n)` if `a=b^d`
- `T(n) = Θ(n^log_b a)` if `a>b^d`

*2 min* • **Activity #2+3**

# Master Theorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^d)$$

- `T(n) = Θ(n^d)` if `a<b^d`
- `T(n) = Θ(n^d log n)` if `a=b^d`
- `T(n) = Θ(n^log_b a)` if `a>b^d`

*2 min* • **Activity #2+3**

# Master Theorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^d)$$

- `T(n) = Θ(n^d)` if `a<b^d`
- `T(n) = Θ(n^d log n)` if `a=b^d`
- `T(n) = Θ(n^(log_b a))` if `a>b^d`

**Activity #2+3**

*1 min*

# Master Theorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^d)$$

- `T(n) = Θ(n^d)` if `a<b^d`
- `T(n) = Θ(n^d log n)` if `a=b^d`
- `T(n) = Θ(n^(log_b a))` if `a>b^d`

**Activity #2+3**

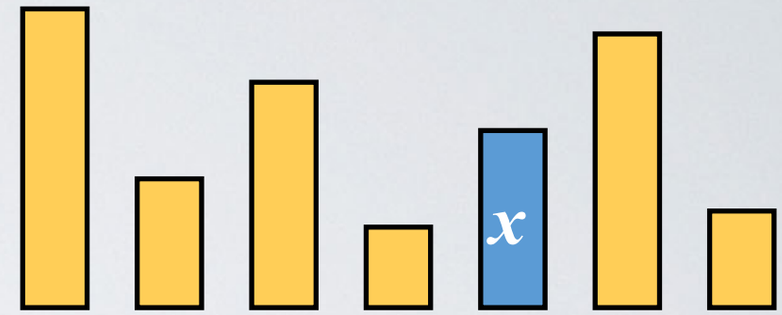
*0 min*

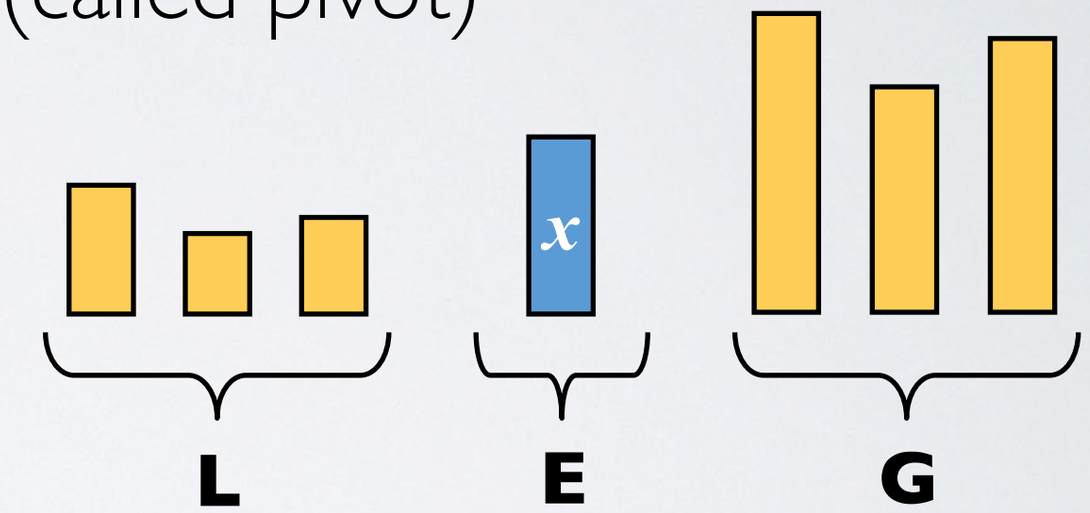
# Outline

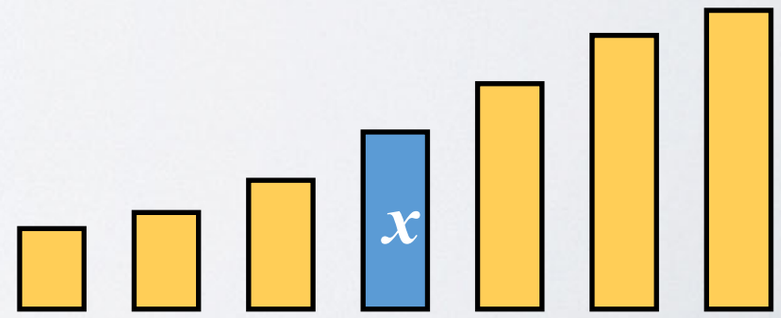
- Motivation
- Quadratic Sorting
  - Selection sort
  - Insertion sort
- Linearithmic Sorting
  - Merge Sort
  - Master Theorem
  - **Quick Sort**
- Comparative sorting lower bound
- Linear Sorting
  - Radix Sort

# 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

**Activity #4**

*2 min*

# Quicksort

**Activity #4**

*2 min*

# Quicksort

**Activity #4**

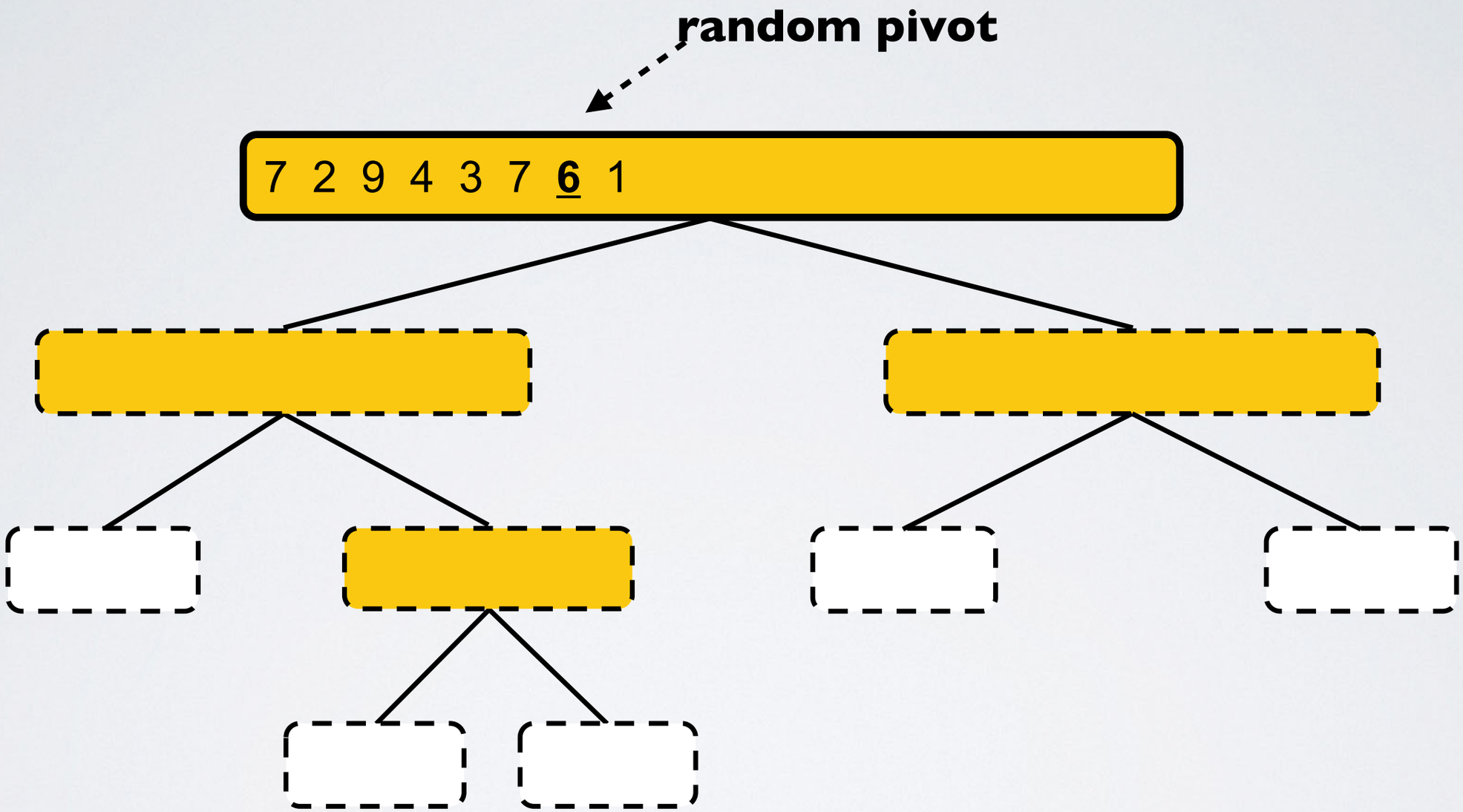
*1 min*

# Quicksort

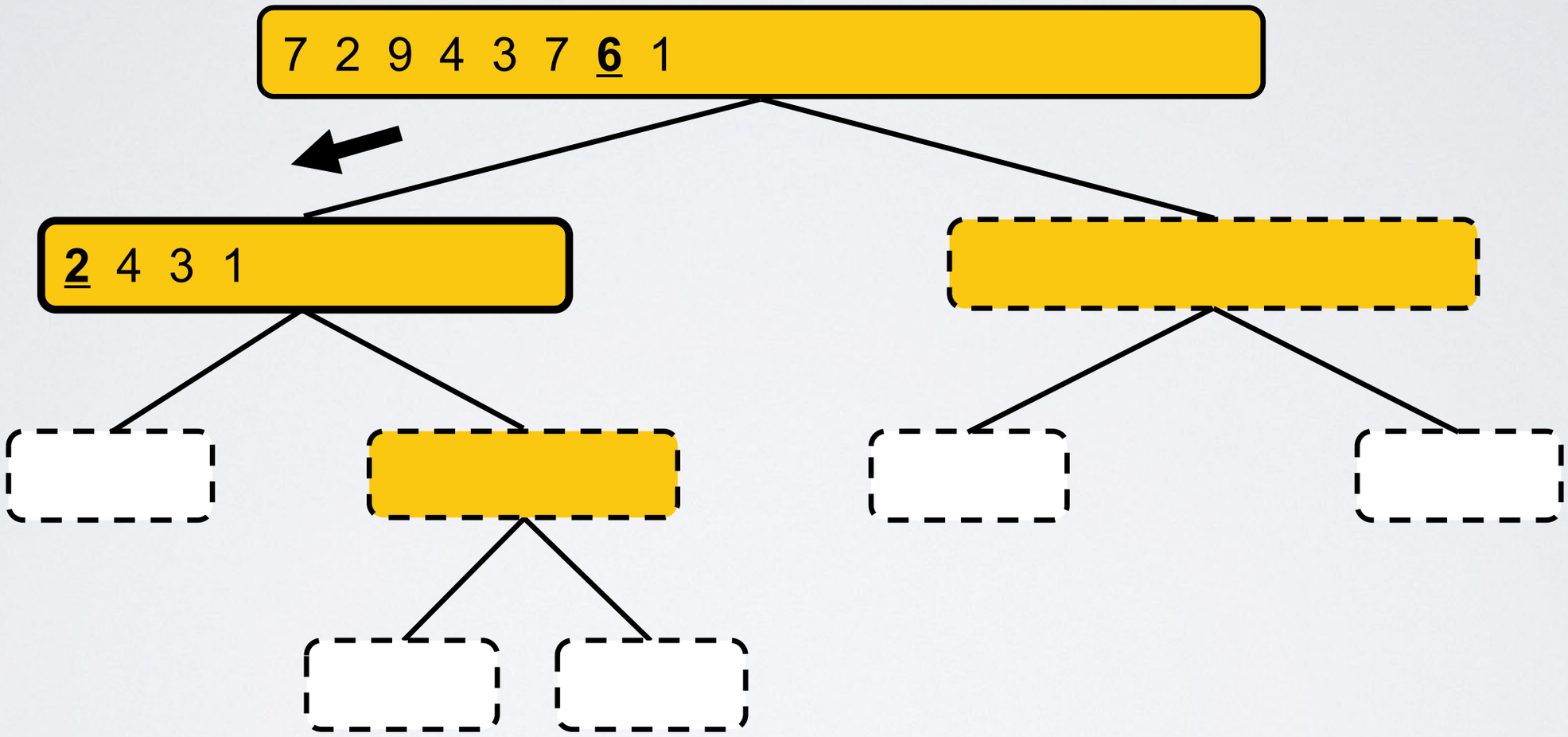
**Activity #4**

*0 min*

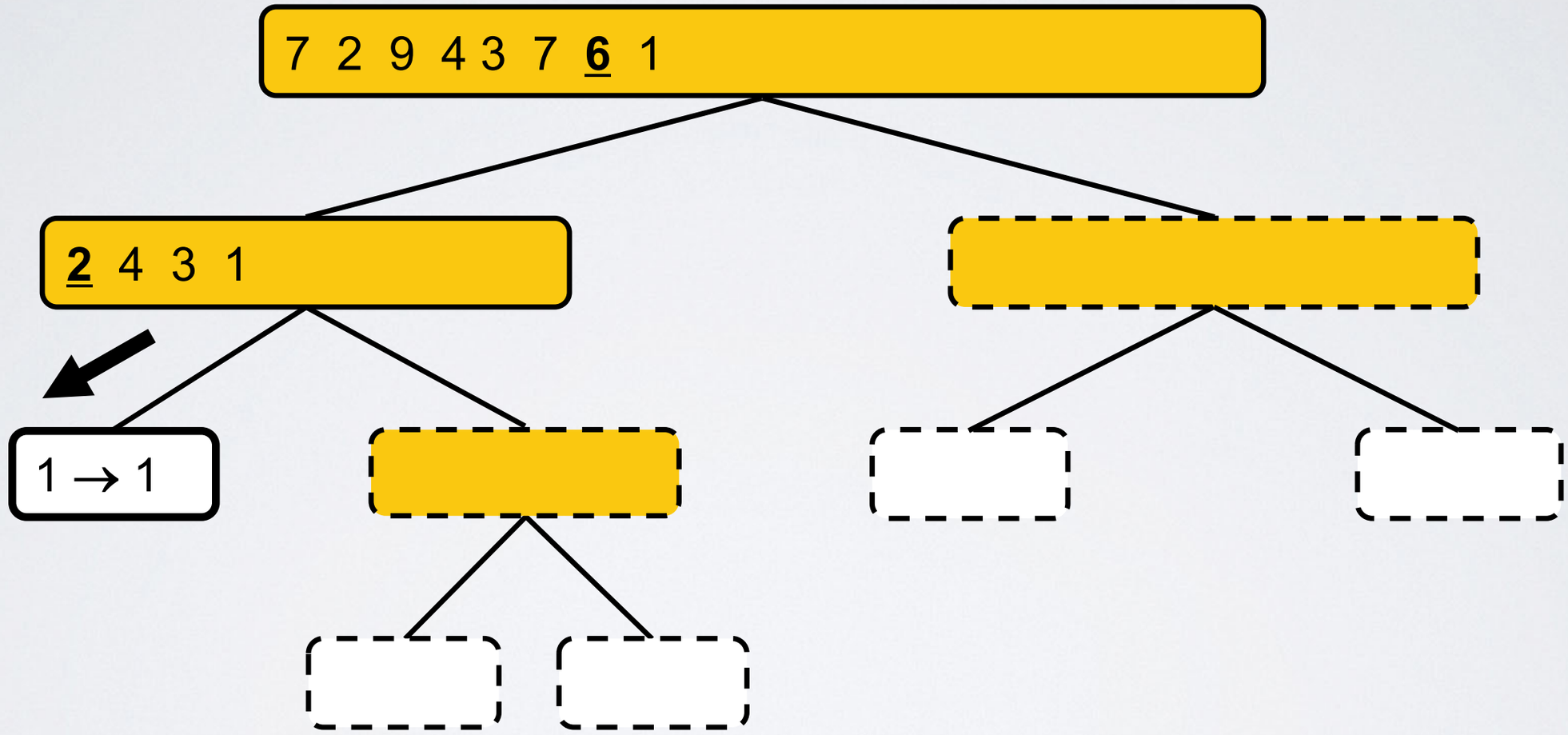
# Quicksort Example



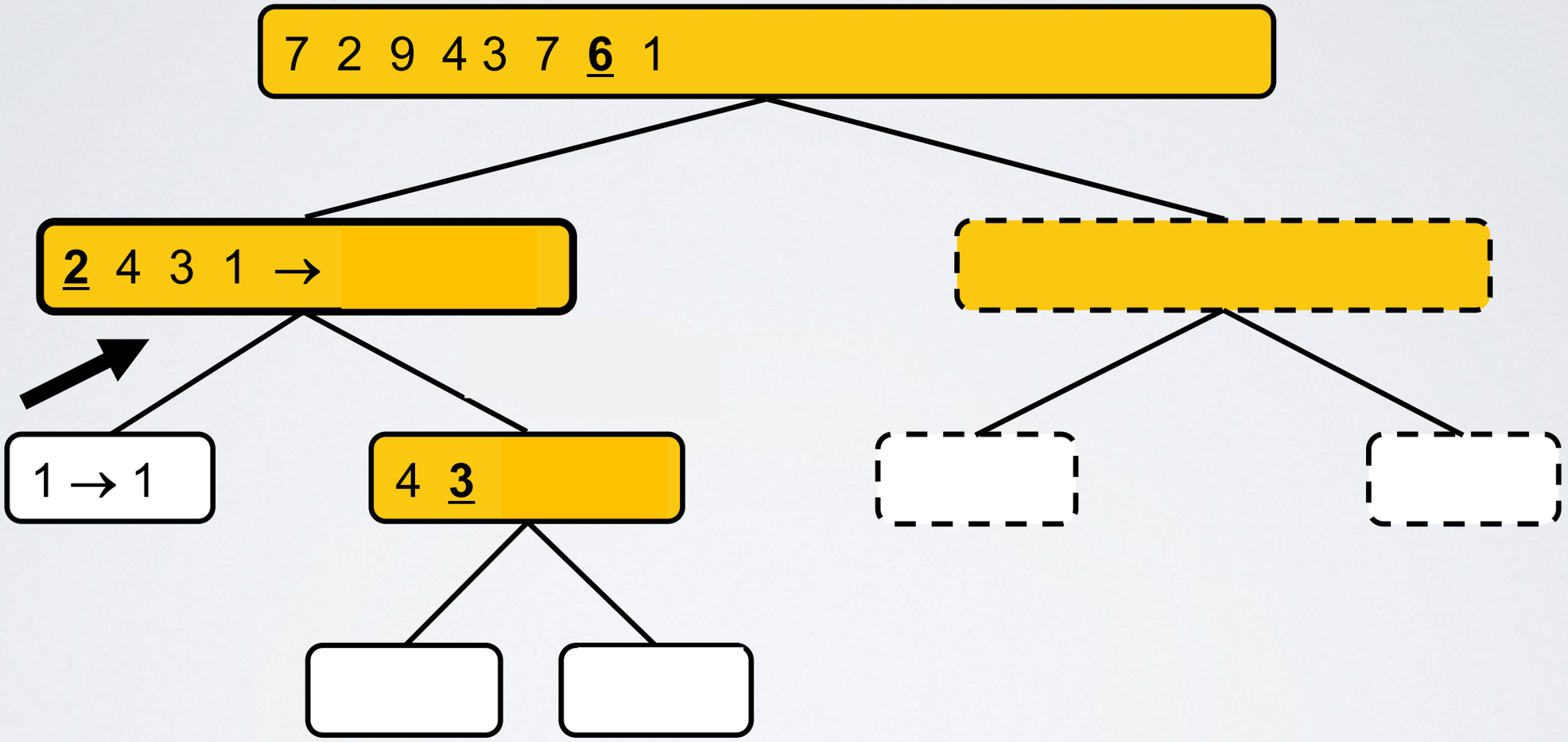
# Quicksort Example



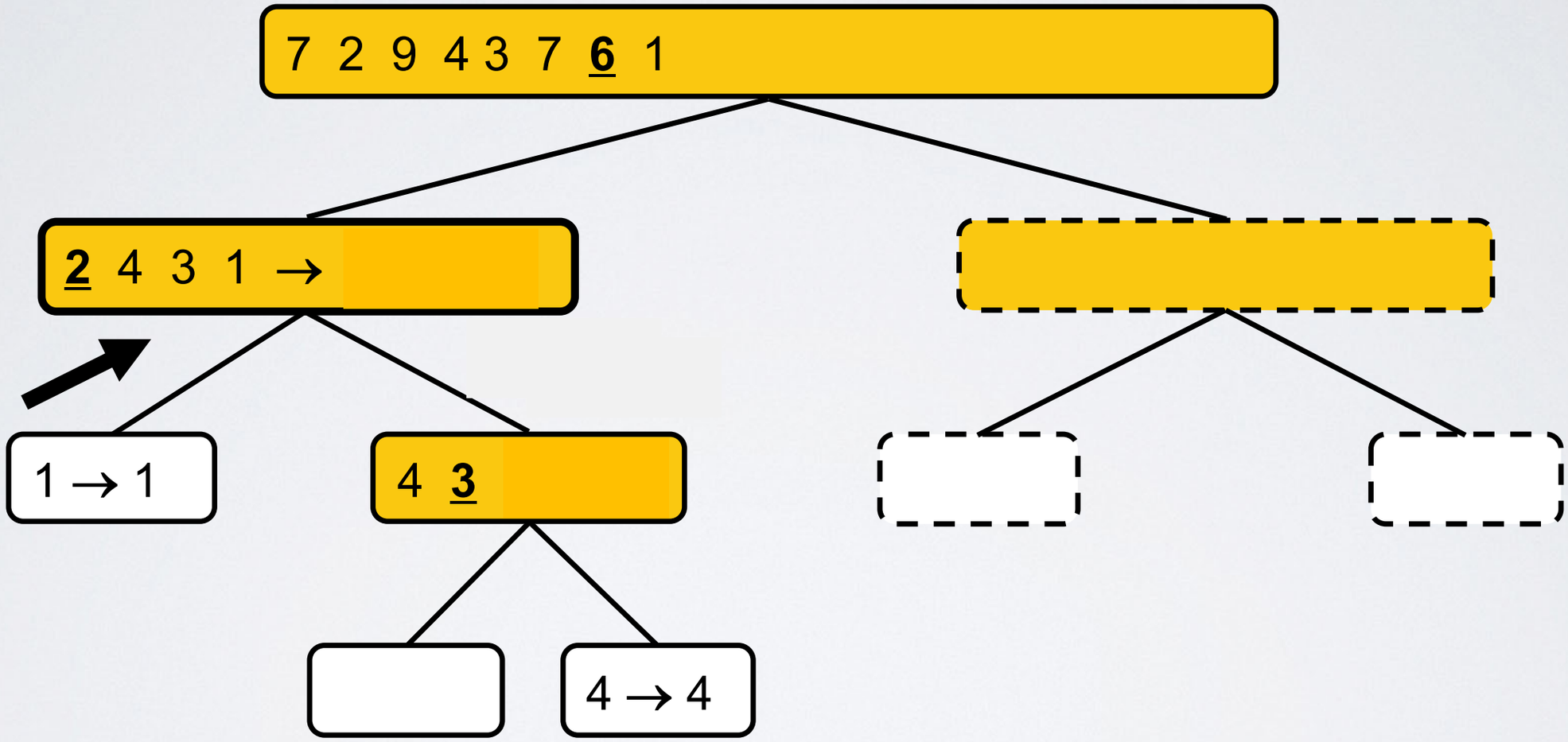
# Quicksort Example



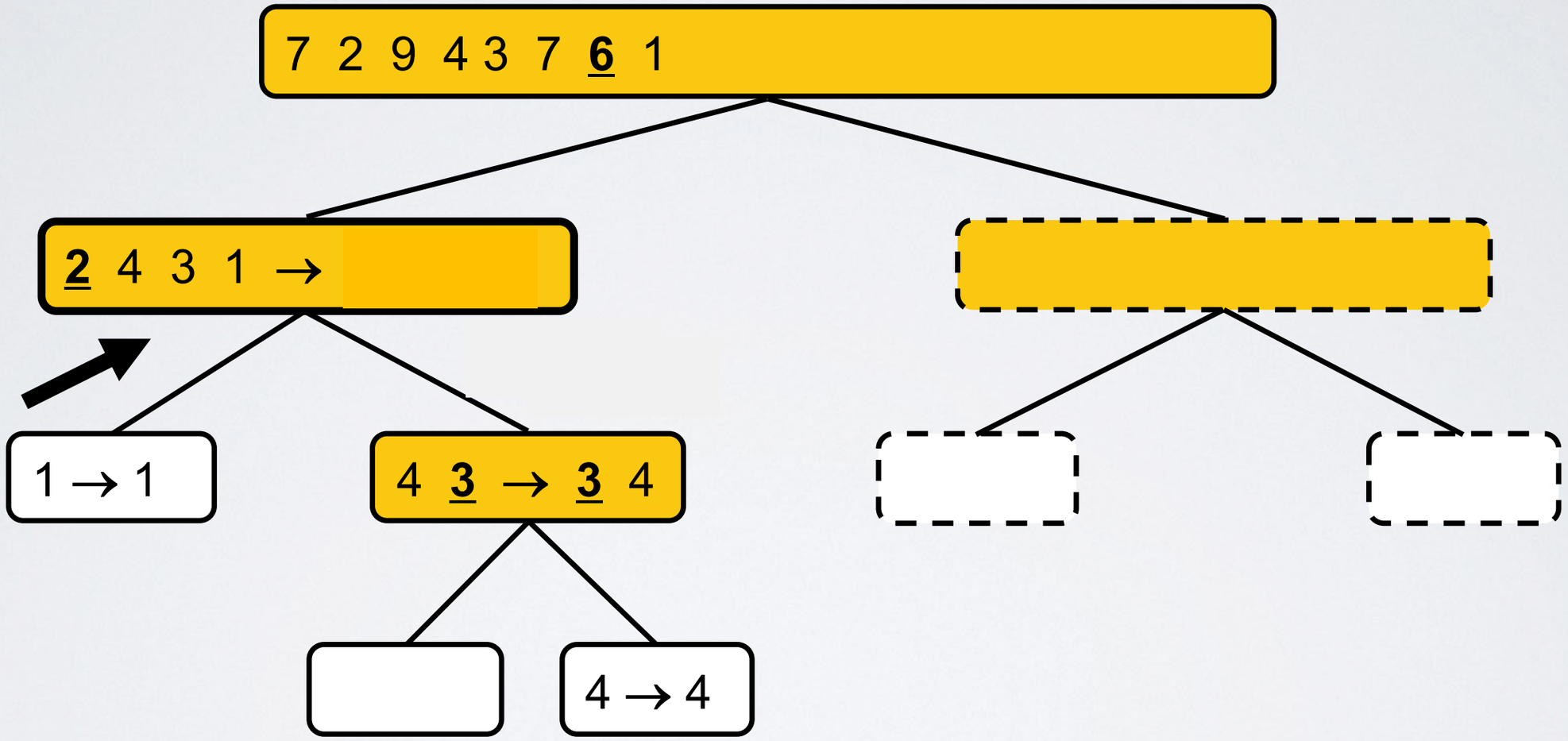
# Quicksort Example



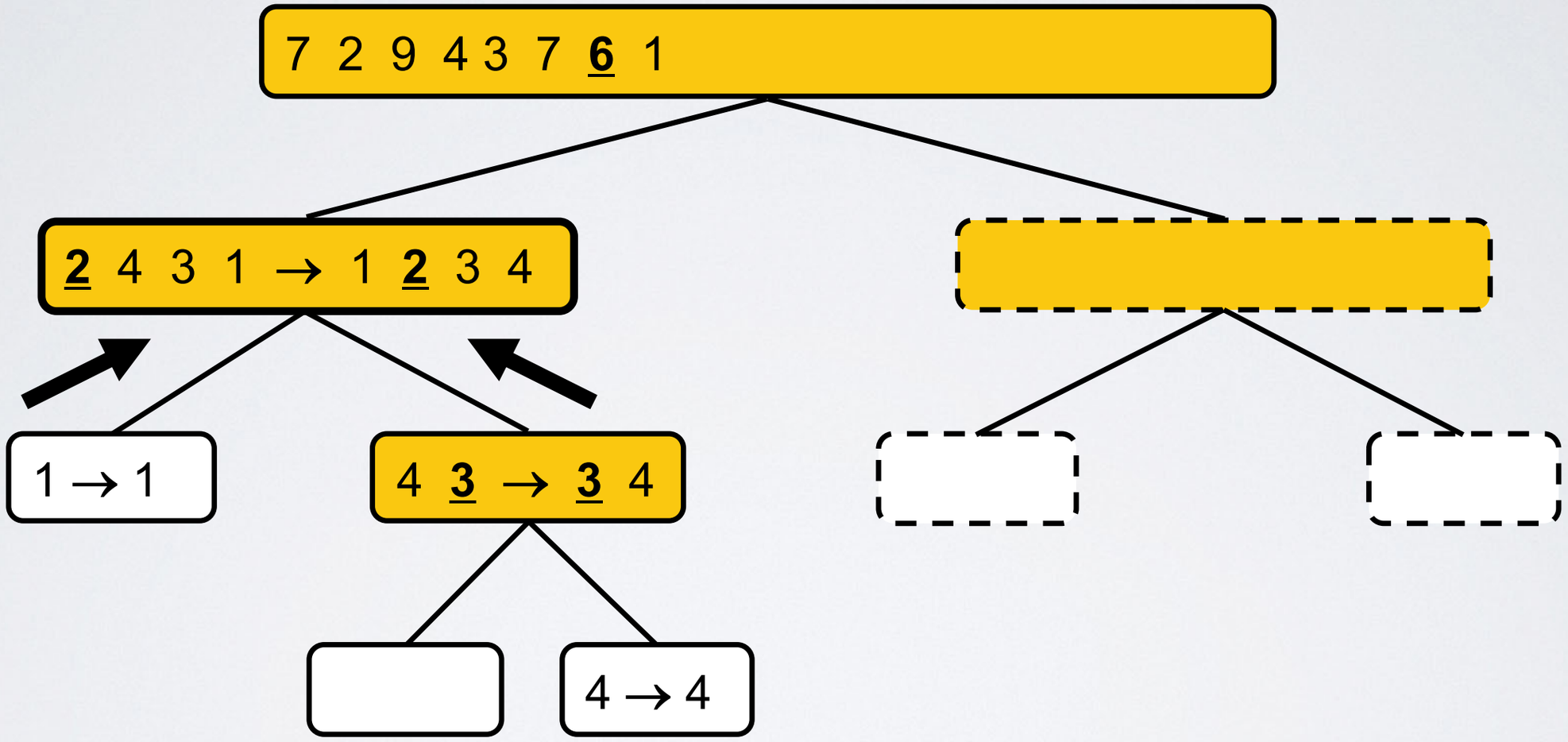
# Quicksort Example



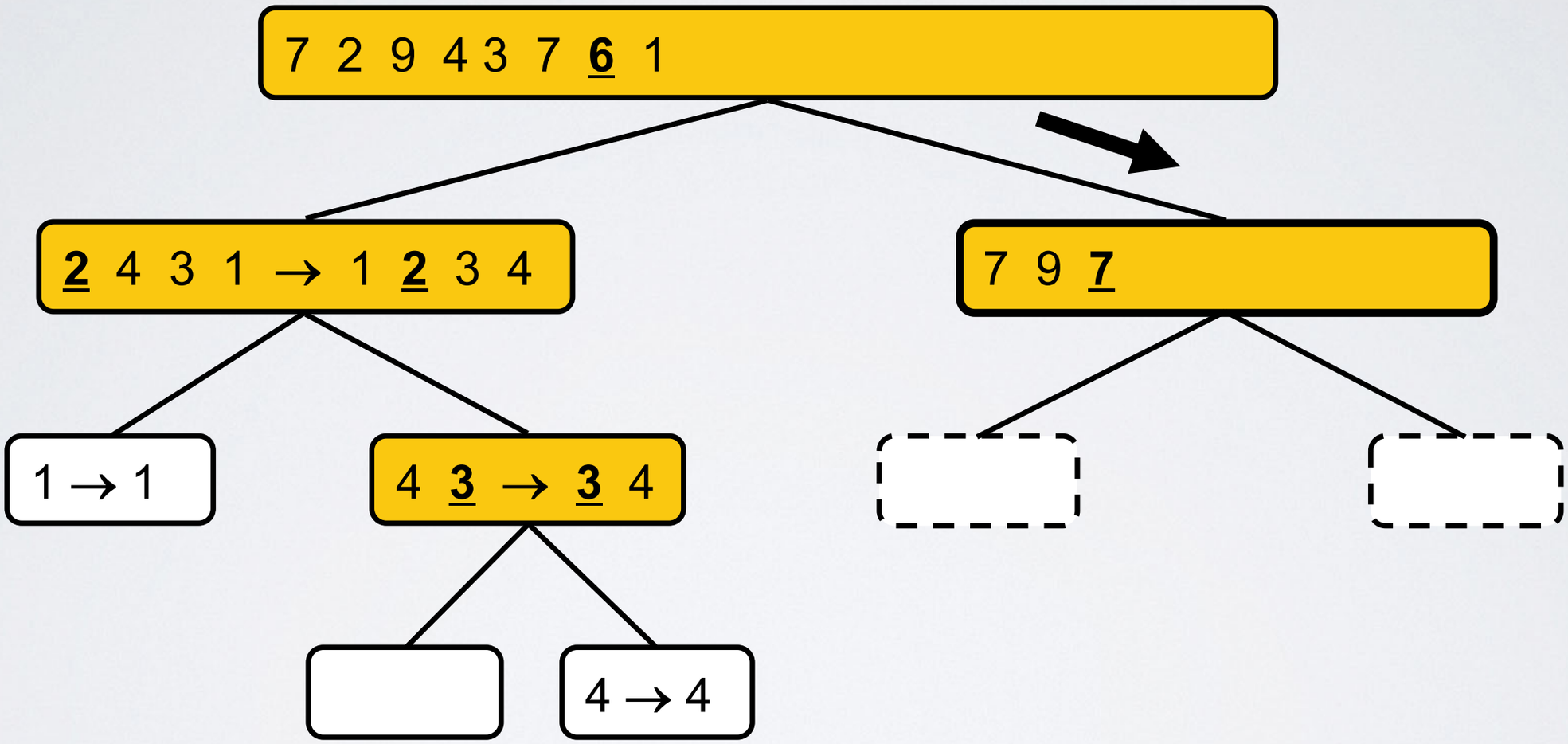
# Quicksort Example



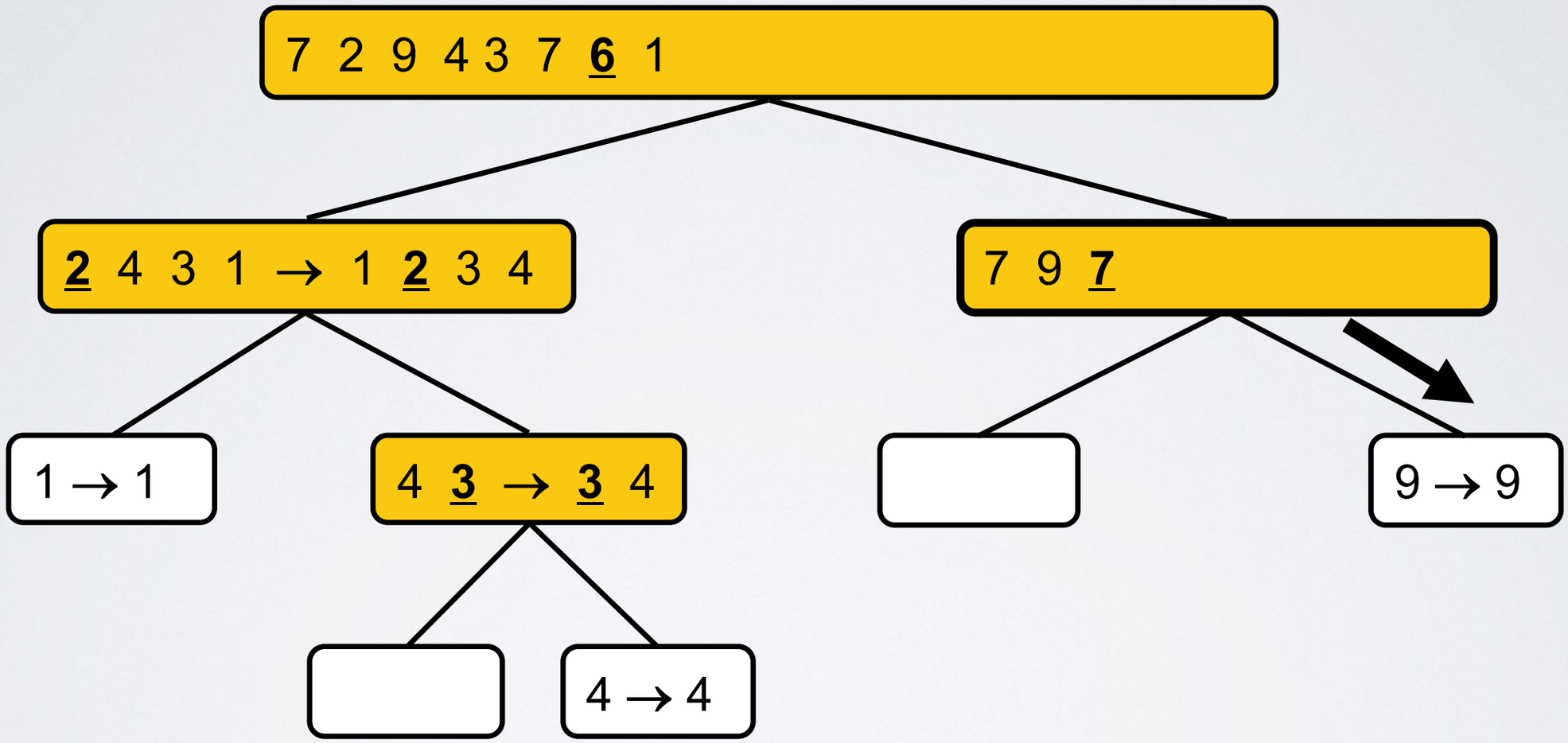
# Quicksort Example



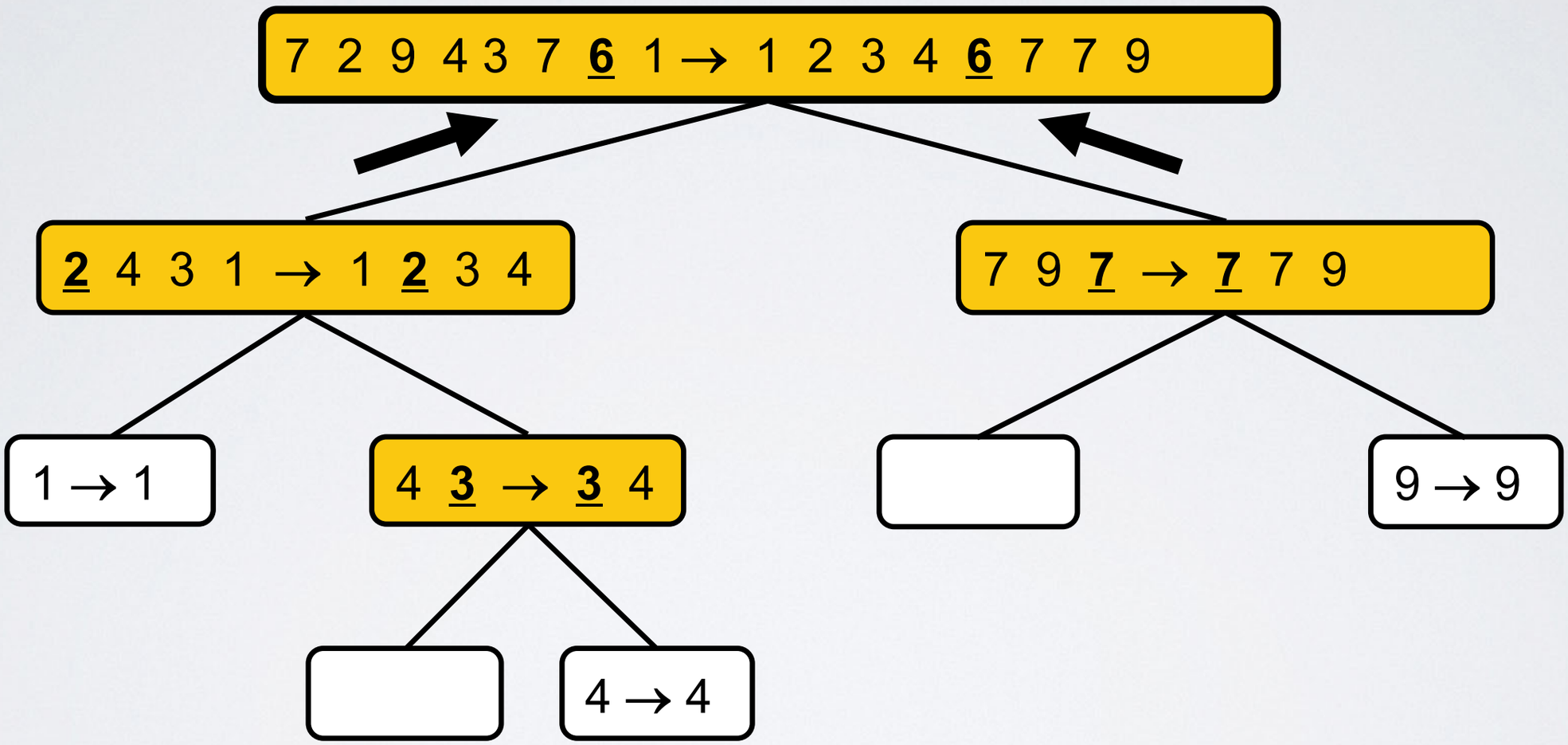
# Quicksort Example



# Quicksort Example



# Quicksort Example



# Quicksort Pseudo-Code

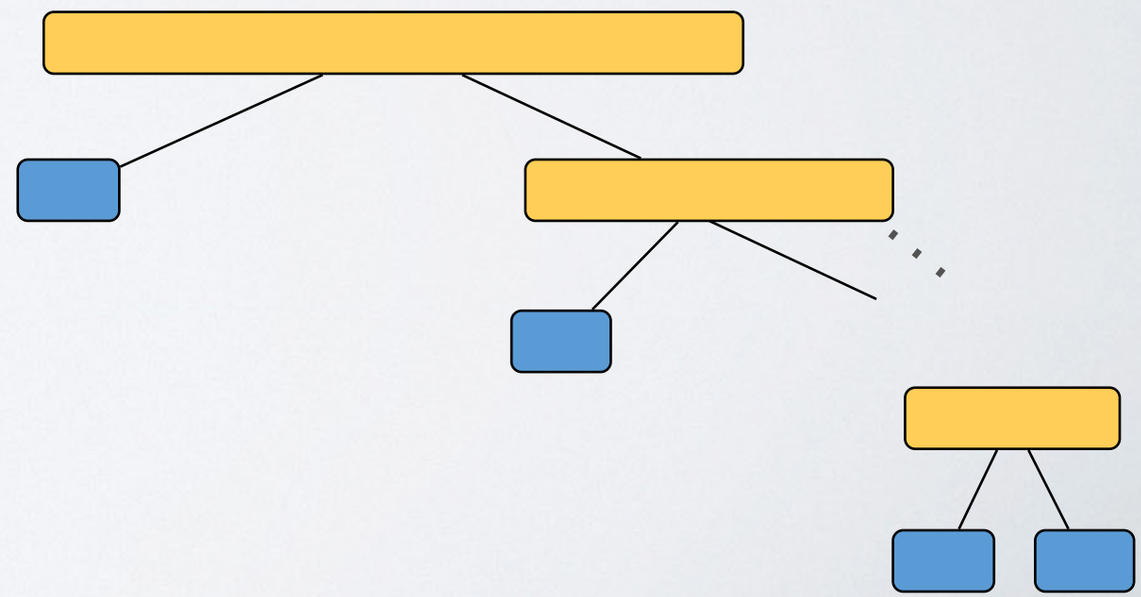
```
function quick_sort(A):
   if A.length ≤ 1
      return A

   pivot = random element from A
   L = [],  E = [],  G = []
   for each x in A:
      if x < pivot:
         L.append(x)
      else if x > pivot:
         G.append(x)
      else E.append(x)
   return quick_sort(L) + E + quick_sort(G)
```

# Worst-Case Running Time

- Worst-case for Quicksort
  - when pivot is (unique) min or max element
  - Either `L` or `G` has size `n-1` and other has size `0`
  - Runtime is proportional to
    - `n + (n-1) + (n-2) + … + 2 + 1`
- Which is $O(n^2)$

| depth | time |
|---|---|
| 0 | n |
| 1 | n-1 |
| 2 | n-2 |
| ⋮ | ⋮ |
| n-1 | 1 |



# Expected Runtime of Quicksort

- Assume there are no duplicates
  - if there are then we have even less recursive calls
- At each level of recursion, Quicksort can make **`n`** different & unique recursive calls depending on the chosen split/pivot
  - **`|L|=0`** and **`|G|=n-1`**
  - **`|L|=1`** and **`|G|=n-2`**
  - …
  - **`|L|=n`** and **`|G|=0`**
- Since there are **`n`** possible splits…
- …and since the split is chosen uniformly at random…
- …each split is chosen with probability **`1/n`**

# Expected Runtime of Quicksort

- Each split is chosen with probability `1/n`
- So expected running time is

**prob of first split** **cost of first split** **prob of last split** **cost of last split**

$$T(n) = n + \frac{1}{n} \cdot \Big( T(0) + T(n-1) \Big) + \cdots + \frac{1}{n} \cdot \Big( T(n-1) + T(n-1-(n-1)) \Big)$$

$$= n + \frac{1}{n} \cdot \sum_{i=0}^{n-1} \Big( T(i) + T(n-1-i) \Big)$$

- Solution is $T(n) = 2n \ln n = 1.39 \cdot n \log_2 n = O(n \log n)$

# Quicksort Pseudo-Code

```
function quick_sort(A):
   if A.length ≤ 1
      return A

   pivot = random element from A
   L = [],  E = [],  G = []
   for each x in A:
      if x < pivot:
         L.append(x)
      else if x > pivot:
         G.append(x)
      else E.append(x)
   return quick_sort(L) + E + quick_sort(G)
```

**Not in place!**

# In-Place Quicksort

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

# In-Place Quicksort

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

# 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

# Outline

- Motivation
- Quadratic Sorting
  - Selection sort
  - Insertion sort
- Linearithmic Sorting
  - Merge Sort
  - Master Theorem
  - Quick Sort
- **Comparative sorting lower bound**
- Linear Sorting
  - Radix Sort

# How Fast Can We Sort?

- Merge sort and Quicksort are **`O(n log n)`**
- Can we do better?
  - No!
  - Well kind of…

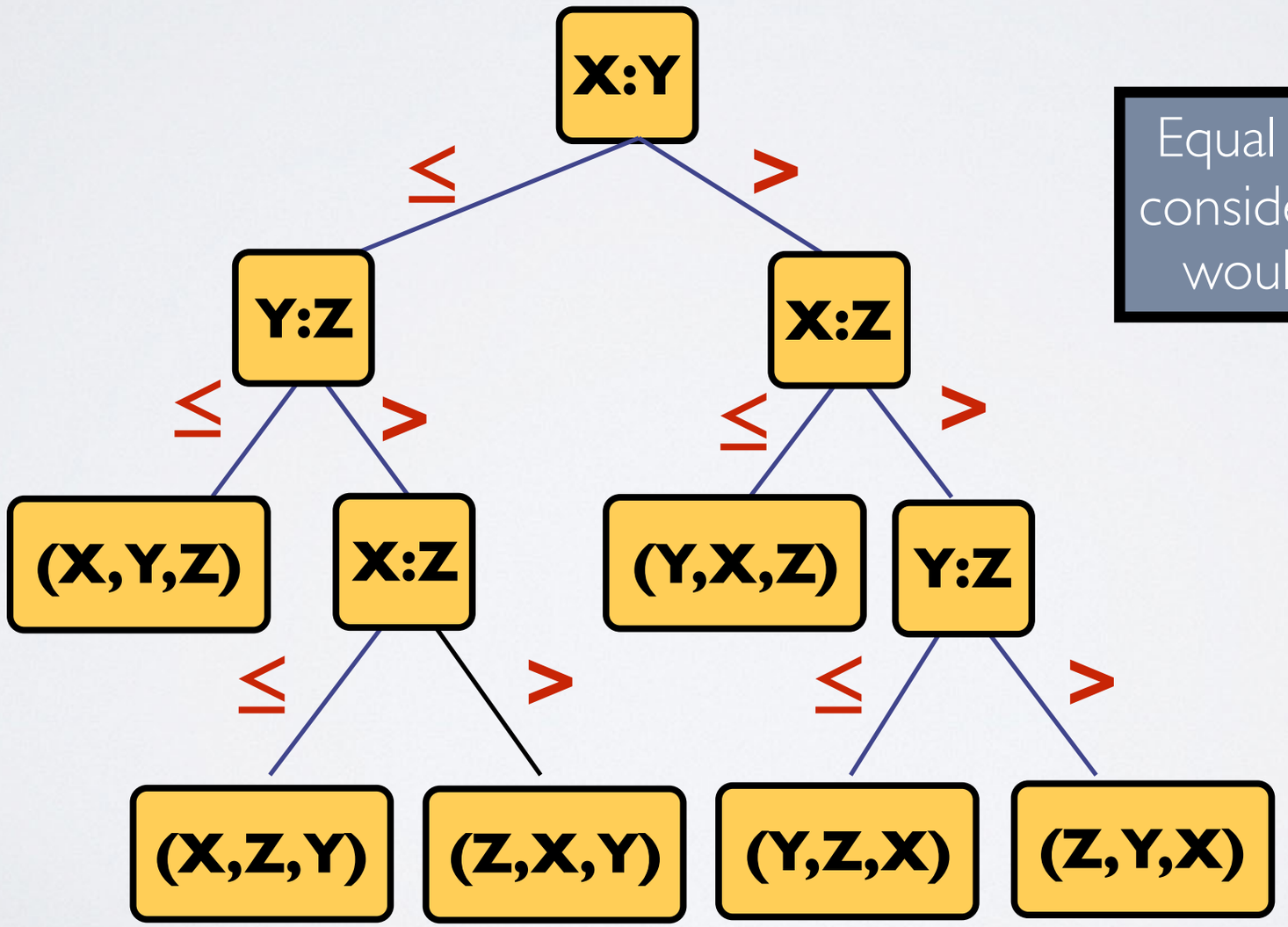
Any comparison-based sorting algorithm has to make at least **`Ω(n log n)`** comparisons in the worst-case to sort n keys

# Lower Bound on Comparative Sorting

- Viewed abstractly, a sorting algorithm
  - takes a sequence of keys $k_1, \ldots, k_n$
  - outputs a permutation of the keys that has them in order
- We can view the optimal (i.e., best possible) algorithm as a perfect binary **decision** tree
  - internal nodes do comparisons of keys
  - leaves are the correct permutation
- To sort a sequence, we traverse tree
- Worst-case number of comparisons is height of tree

# Lower Bound on Comparative Sorting

- Suppose our input is X, Y, Z…
- …and the proper order is Z, X, Y




Equal terms will be considered ≤ but ≥ would also work

# Lower Bound on Comparative Sorting

- How many leaves does tree have?
  - **`n!`** because there are **`n!`** permutations of a sequence of **`n`** elements
  - A perfect binary tree with **`L`** leaves has height **`log L`**
- So tree with **`n!`** leaves has height **`log(n!)`**
- Based on Stirling's formula: $n! > \left(\frac{n}{e}\right)^n$

$$\log(n!) > \log\left(\left(\frac{n}{e}\right)^n\right)$$

$$\log(n!) > n\log n - n\log e$$

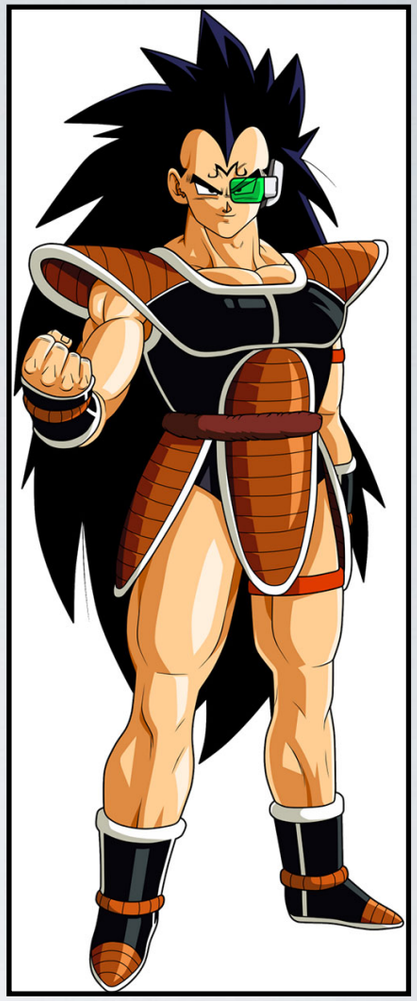
- So height of tree (and # of comparisons) is **`Ω(n log n)`**

# 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 Radix sort

# Radix Sort

- How would you sort `258391` and `258492`?
  - digit by digit
  - the 3 high order digits are same…
  - …so you keep going until you see `3<4` so `258391` must less than `258492`



# Radix Sort

- How would you sort an array of numbers between 0 and 9?
  - example: `[5,1,6,2,3,1]` → `[1,1,2,3,5,6]`
- Create array of `10` buckets
- for each number `x`, add it to bucket at index `x`
- Return concatenation of all buckets (in order)
  - print out `[1,1]+[2]+[3]+[5]+[6]`
- Runtime is `O(n)`

# Radix Sort


- Radix sort combines both approaches
  - iterate from least significant to most significant digit
  - sort number by digit
- Takes advantage of
  - the "**digit-iness**" of integers
  - for every digit there are `O(1)` number of options

# Radix Sort

- Sort `[273,279,8271,7891,8736,8735]`
- Start with lowest-order digit (the **1**'s place)
  - add number to bucket corresponding to that digit

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 8271 | | 273 | | 8735 | 8736 | | | 279 |
| | 7891 | | | | | | | | |

- Concatenate all buckets
  - `[8271,7891,273,8735,8736,279]`
- Now sorted by lowest-order digit

# Radix Sort

- Sort `[8271,7891,273,8735,8736,279]`
- Start with second lowest-order digit (the `10`'s place)
  - add number to bucket corresponding to that digit

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | 8735 | | | | 8271 | | 7891 |
| | | | 8736 | | | | 273 | | |
| | | | | | | | 279 | | |

- Concatenate all buckets
  - `[8735,8736,8271,273,279,7891]`
- Now sorted by second and lowest-order digit

# Radix Sort

- Sort `[8735,8736,8271,273,279,7891]`
- Start with third lowest-order digit (the `100`'s place)
  - add number to bucket corresponding to that digit

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | 8271 | | | | | 8735 | 7891 | |
| | | 273 | | | | | 8736 | | |
| | | 279 | | | | | | | |

- Concatenate all buckets
  - `[8271,273,279,8735,8736,7891]`
- Now sorted by third, second and lowest-order digit

# Radix Sort

- Sort `[8271,273,279,8735,8736,7891]`
- Start with third lowest-order digit (the `1000`'s place)
  - add number to bucket corresponding to that digit

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 273 | | | | | | | 7891 | 8271 | |
| 279 | | | | | | | | 8735 | |
| | | | | | | | | 8736 | |

- Concatenate all buckets
  - `[273,279,7891,8271,8735,8736]`
- Now sorted by third, second and lowest-order digit

# Radix Sort

```
function radix_sort(A):
   buckets = array of 10 lists
   for place = least to most significant
      for number in A
         d = digit in number at place
         buckets[d].append(number)
   A = concatenate all buckets in order
   empty all buckets
return A
```

- Very efficient!
  - **O(nd)**
  - d is number of digits in the largest number

# More on Radix Sort

- Can be applied to
  - positive integers in base 10 (we just saw this)
  - Octals (base 8)
  - Hexadecimals (base 16)
  - Strings (one bucket for every valid character)
- Number of buckets can be different at each round
- Can represent almost anything as a bit string
  and radix sort with two buckets
  - number of digits will dominate runtime
  - for long sequences will be very slow

# Radix Sort

**Activity #5**

*2 min*

# Radix Sort

**Activity #5**

*2 min*

# Radix Sort

**Activity #5**

*1 min*

# Radix Sort

**Activity #5**

*0 min*

# Summary of Sorting Algorithms

| **Algorithm** | **Time** | **Notes** |
|---|---|---|
| Selection sort | $O(n^2)$ | in-place<br>slow (good for small inputs) |
| Insertion sort | $O(n^2)$ | in-place<br>slow (good for small inputs) |
| Merge sort | $O(n \log n)$ | fast (good for large inputs) |
| Quick sort | $O(n \log n)$<br>expected | randomized<br>fastest (good for large inputs) |
| Radix sort | $O(nd)$ | d is number of digits in largest number<br>basically linear when d is small |

# Readings

- Dasgupta et al.
  - **Section 2.1:** good intro to divide & conquer
  - **Section 2.2:** review of recurrence rels. & master theorem
  - **Section 2.3:** analysis of merge sort & lower bound on comparative sorting

# References

- Slide #62
  - The character depicted is Raditz (sometimes called Radix) from the Anime *Dragon Ball Z*. He is the biological brother of Goku and one of the four remaining Universe 7 Saiyans.
- Slide #64
  - The RZA is the main producer and leader of the Wu-Tang Clan. He also released albums as his alter ego Bobby Digital