RŪŠIAVIMO ALGORITMAI

RŪŠIAVIMO ALGORITMŲ ANIMACIJA:

https://www.bluffton.edu/~nesterd/java/SortingDemo.html

Outline

- Motivation
- Quadratic $O(n^2)$ Sorting
  - Selection Sort
  - Insertion Sort
- "Linearithmic" $O(n \log n)$ Sorting
  - Merge Sort
  - Quick Sort
- The Master Theorem
- Linear Sorting
  - Radix Sort

Motivation

- 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 Algorithms

- There are many ways to sort arrays:
  - Iterative vs. recursive
  - In-place vs. not-in-place
    - An **in-place** algorithm transforms an input data structure with a **small (constant) amount of extra storage space**
    - In the context of sorting, this means that the input array is **overwritten** by the output as the algorithm executes instead of introducing a new array
  - Comparison-based vs. non-comparative
    - Most sorting algorithms we'll analyze are comparison-based--that is, the sort is produced by comparing elements with each other--but some (like Radix Sort, which we'll see later) are not

“In-placeness”

Consider this function to reverse 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!

Now consider this version:

```
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
    // return statement not needed!
```

In-place!

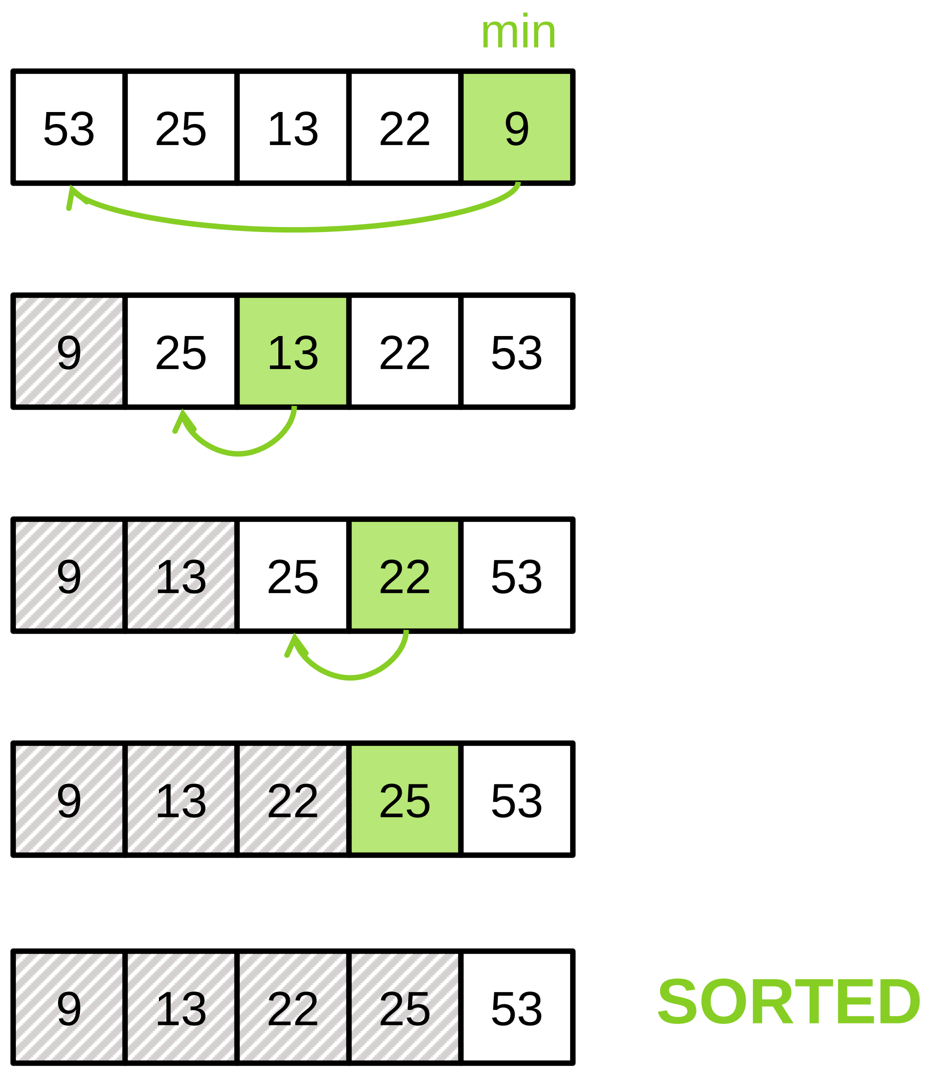
Advantages of in place solutions?

- They are sometimes more difficult to write, but take up much less memory
- Tradeoff between space efficiency and simplicity of the algorithm
- More difficult to write in-place solutions for recursive algorithms

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* the smallest element of the array and places it at index 0, then *selects* the second smallest and places it in index 1, then the third smallest in index 2, etc..
- Advantages:
  - Very simple
  - **Memory efficient**: in-place means swapping elements within same array
- Disadvantages:
  - Slow: runs in **quadratic $O(n^2)$ time**.

Selection Sort (2)



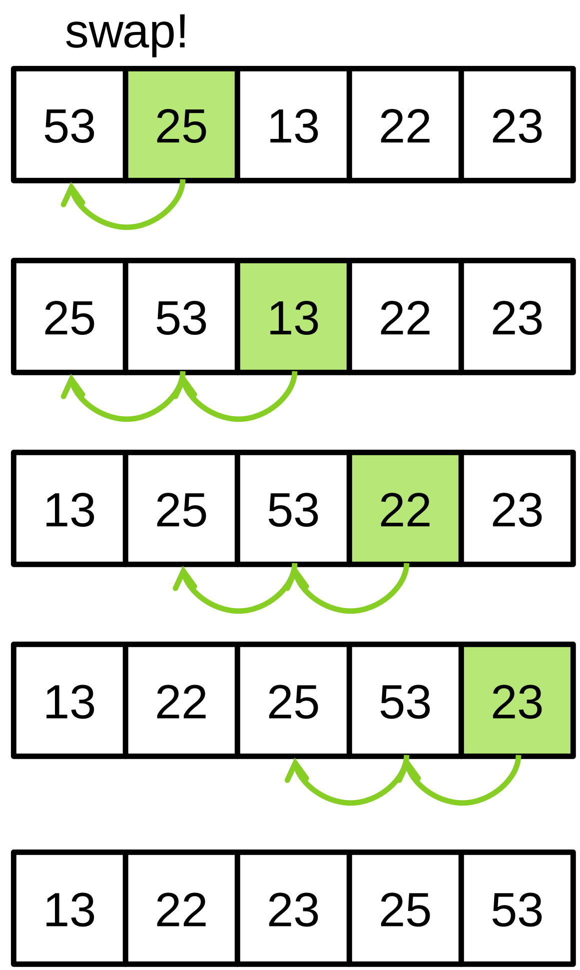
Selection Sort (3)

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

Insertion Sort

- Usually **iterative** and **in-place**
- Arranges items one at a time by comparing each element with every element before it and inserting it into the correct position
- Advantages:
  - Works particularly well if the list is already **partially sorted** (we'll see why)
  - Memory efficient: in-place means swapping elements within same array
- Disadvantages:
  - Slow: runs in **quadratic $O(n^2)$ time**.

Insertion Sort (2)



Note that since 23 > 22, we don't need to check any further, since that part of the array is already sorted. This demonstrates that insertion sort will cruise through an already sorted array in linear time!

Insertion Sort (3)

```
function insertion_sort(A):
   // Input: Unsorted List
   // Output: Sorted List

   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  // this prevents double checking the
                    // already sorted portion
```

Divide-and-Conquer

- **Divide-and-conquer** is a general algorithm design paradigm:
  - **Divide**: divide the input data S into disjoint subsets $S_1, S_2, \ldots, S_k$
  - **Recur**: solve the subproblems associated with $S_1, S_2, \ldots, S_k$
  - **Conquer**: combine the solutions for $S_1, S_2, \ldots, S_k$ into a solution for S
- The base case for the recursion is generally subproblems of size 0 or 1

Merge Sort

- Merge sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like the quadratic sorts, merge sort is **comparative**
- Unlike the quadratic sorts, merge sort is **recursive** and runs in **linearithmic,** $\mathbf{O(n \log n)}$**,** time!
  - Let's see why…

Merge Sort (2)

- Merge sort on an input sequence S with n elements consists of three steps:
  - **Divide**: partition S into two sequences S_1 and S_2 of $n/2$ elements each
  - **Recur**: recursively merge sort S_1 and S_2
  - **Conquer**: merge S_1 and S_2 into a sorted sequence

Merge Sort: Example

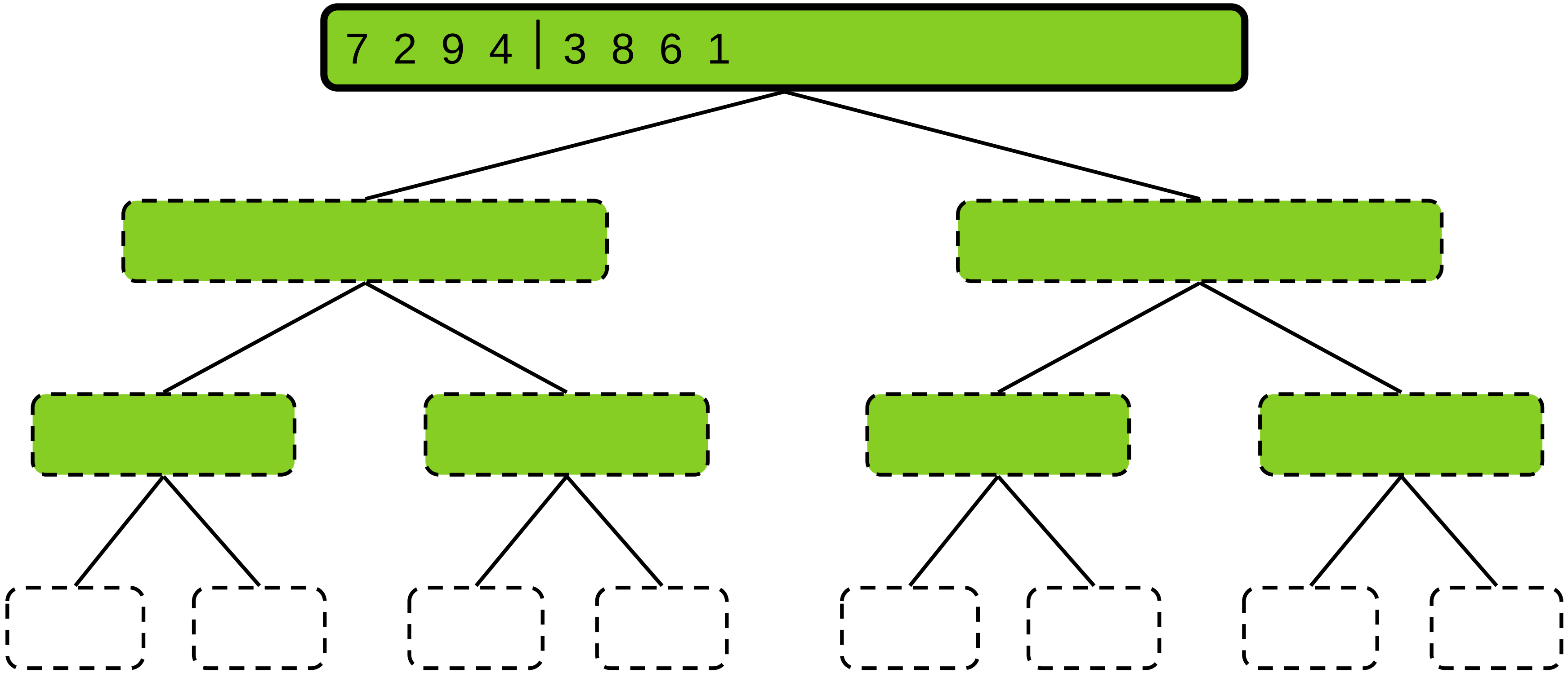
- The execution of merge sort can be depicted by a binary tree
  - each node represents a subproblem in merge sort, showing the unsorted subproblem before calling merge sort recursively
  - the root is the initial call
  - the leaves are the base cases of the recursion, with subsequences of size 0 or 1

Merge Sort: Example

- Suppose we want to sort [7, 2, 9, 4, 3, 8, 6, 1]

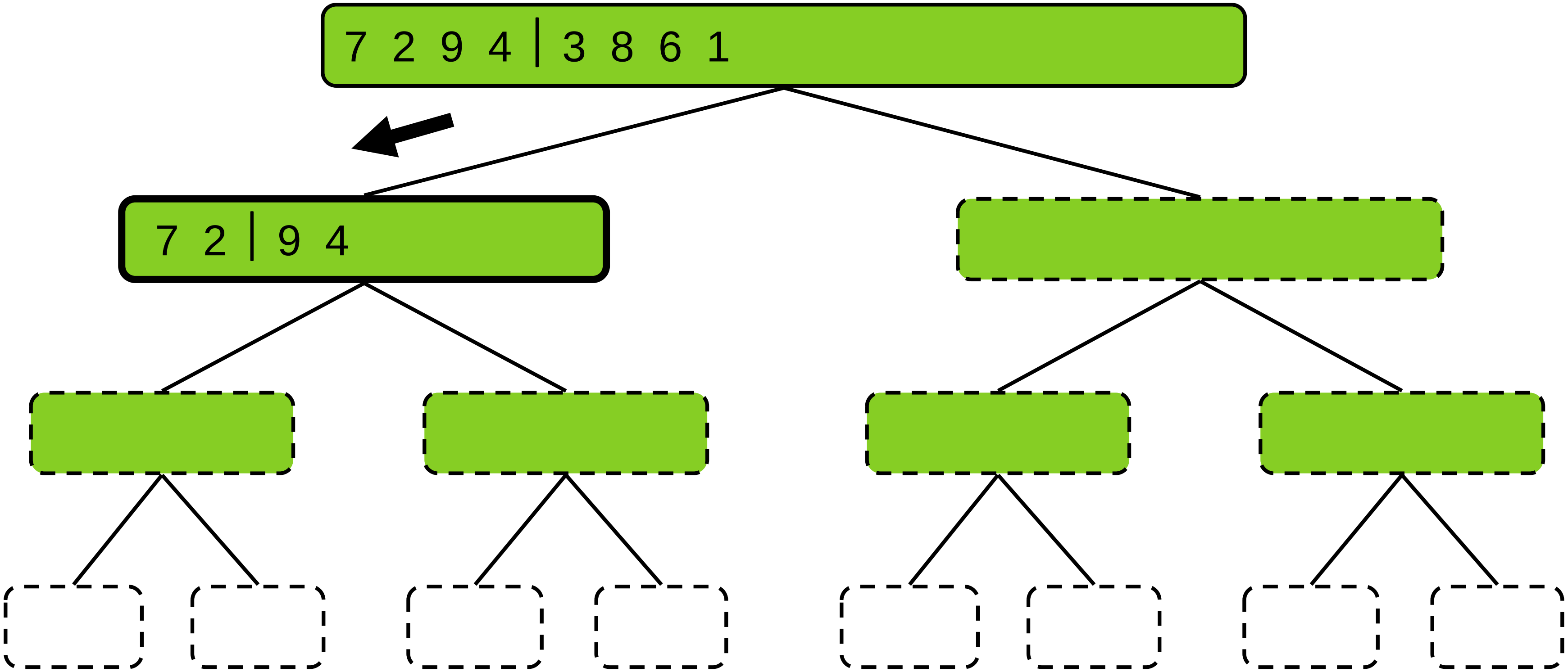
Merge Sort: Example (2)

- Partition



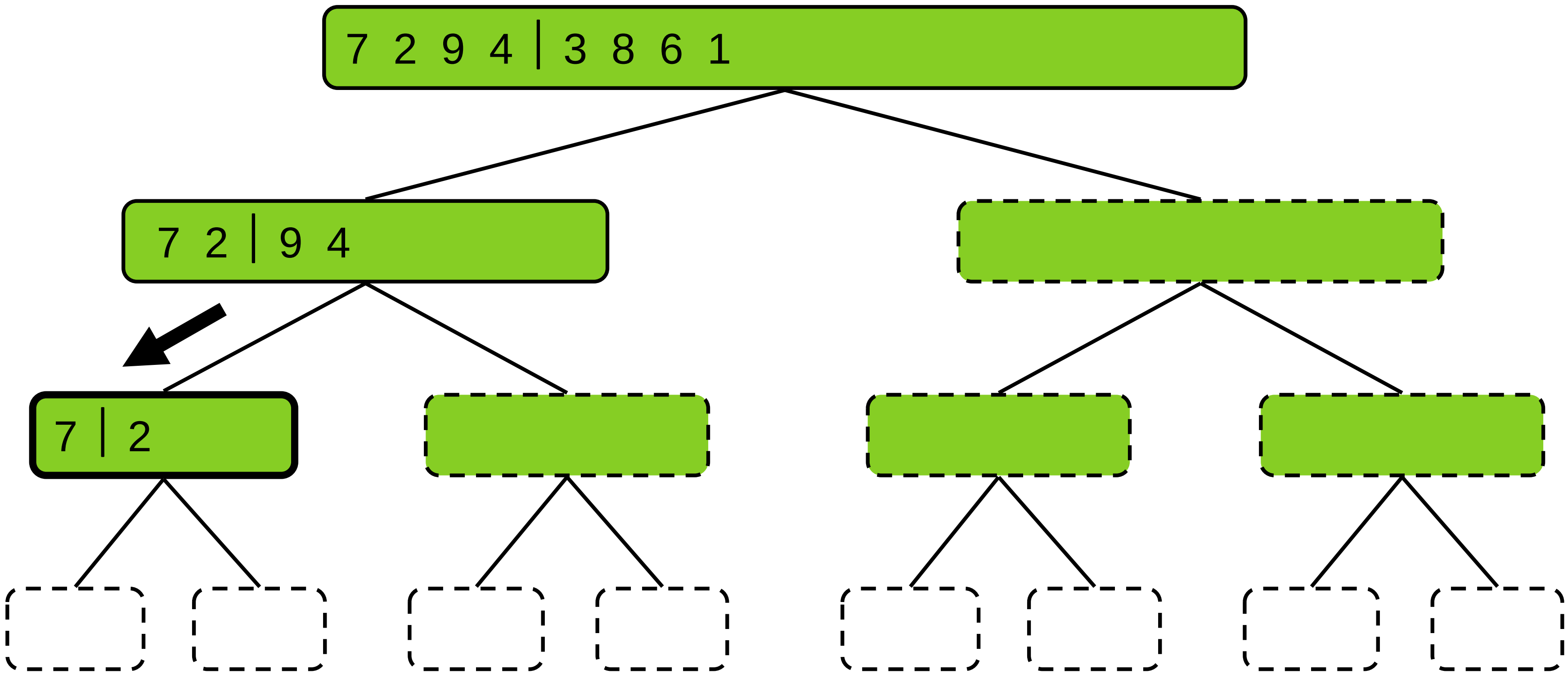
Merge Sort: Example (3)

- Recursive call, partition



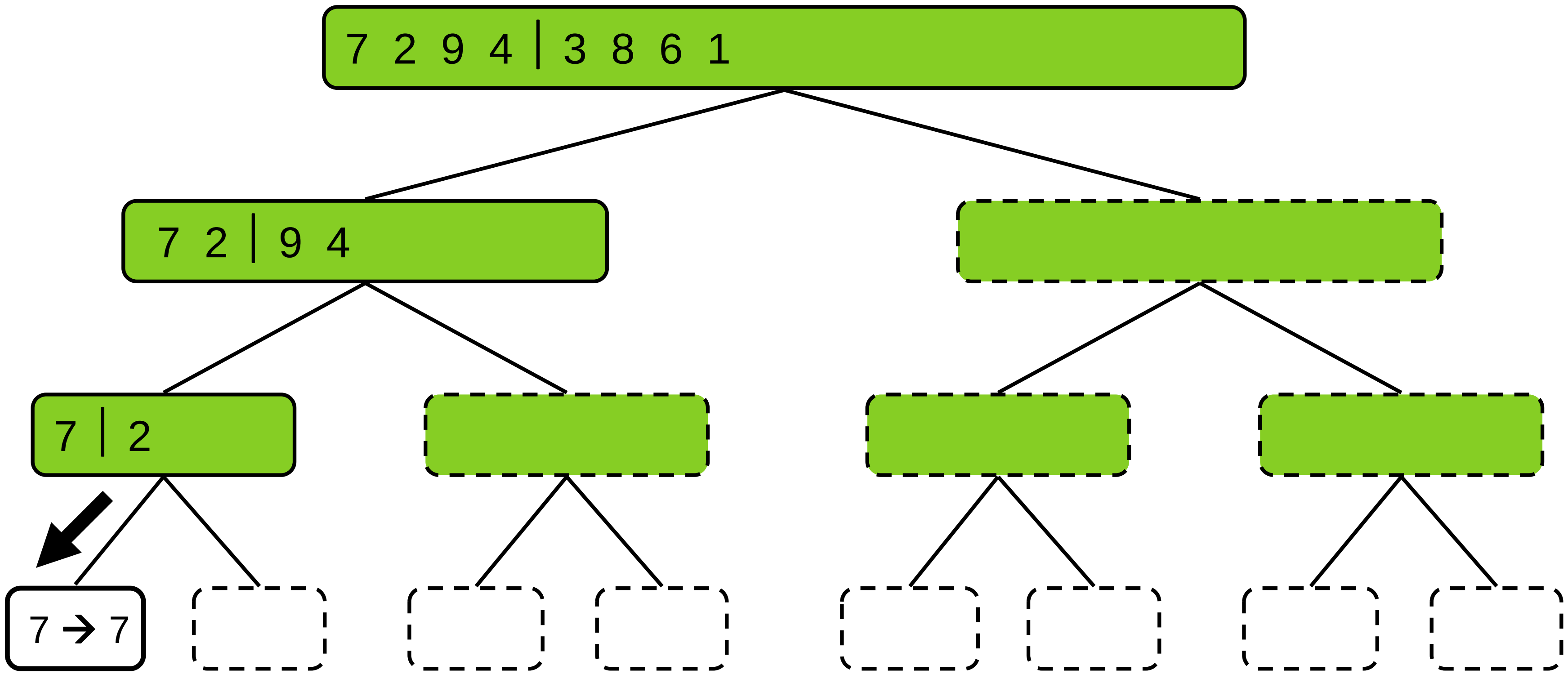
Merge Sort: Example (4)

- Recursive call, partition



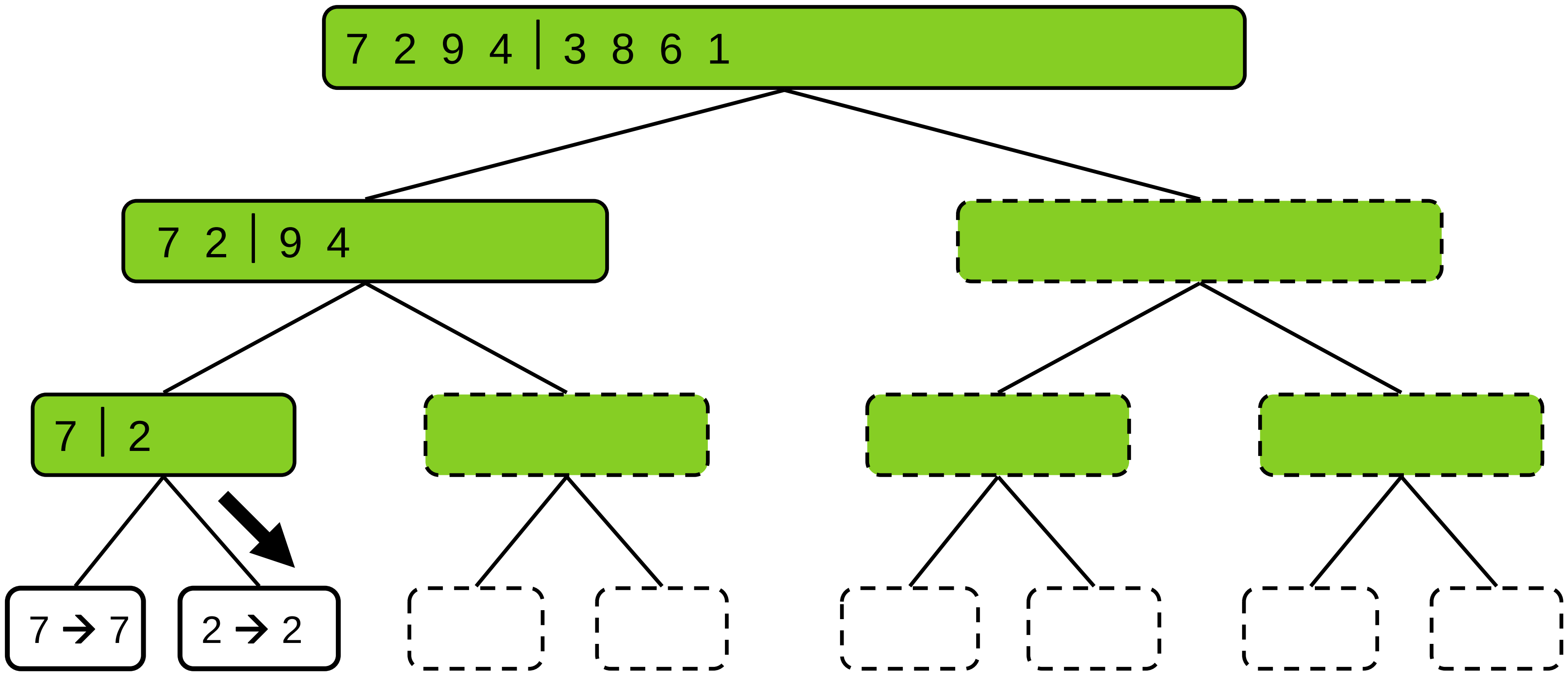
Merge Sort: Example (5)

- Recursive call, base case



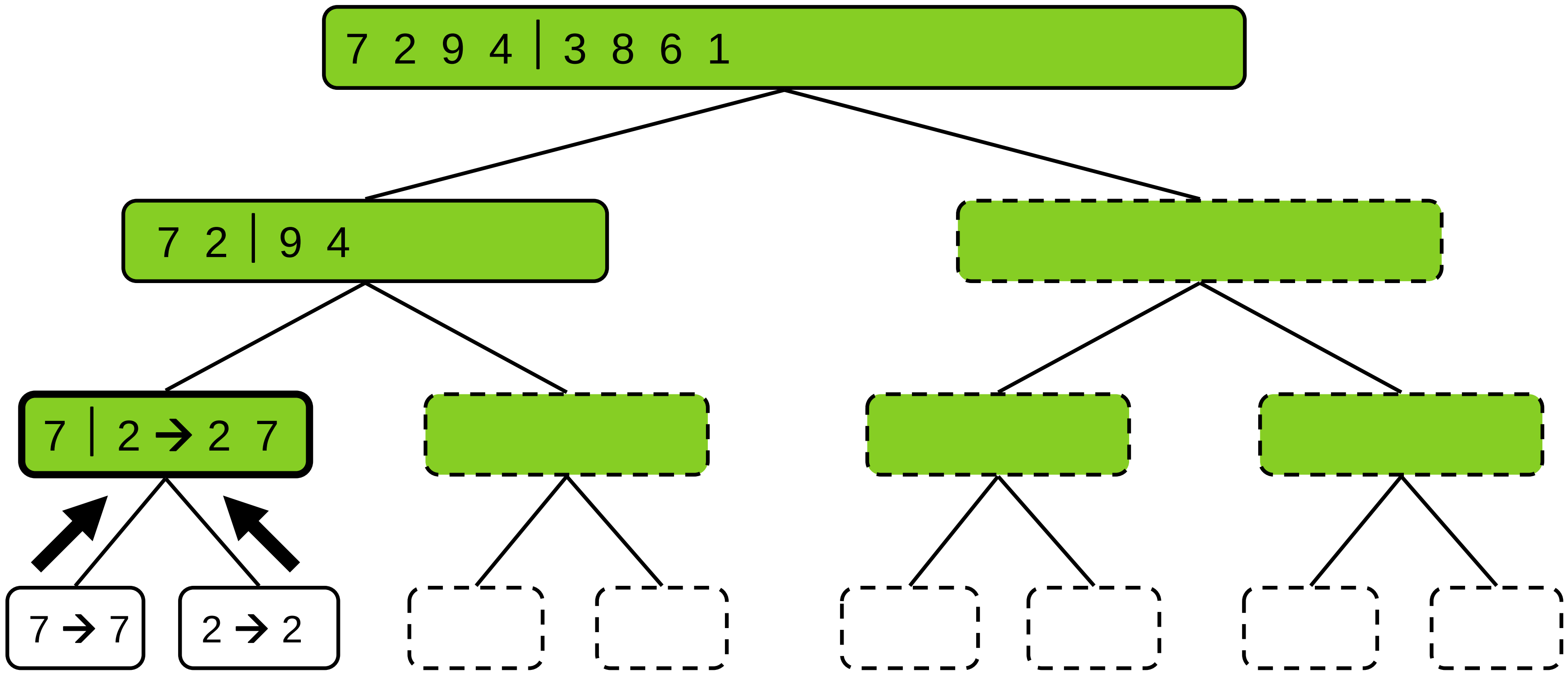
Merge Sort: Example (6)

- Recursive call, base case



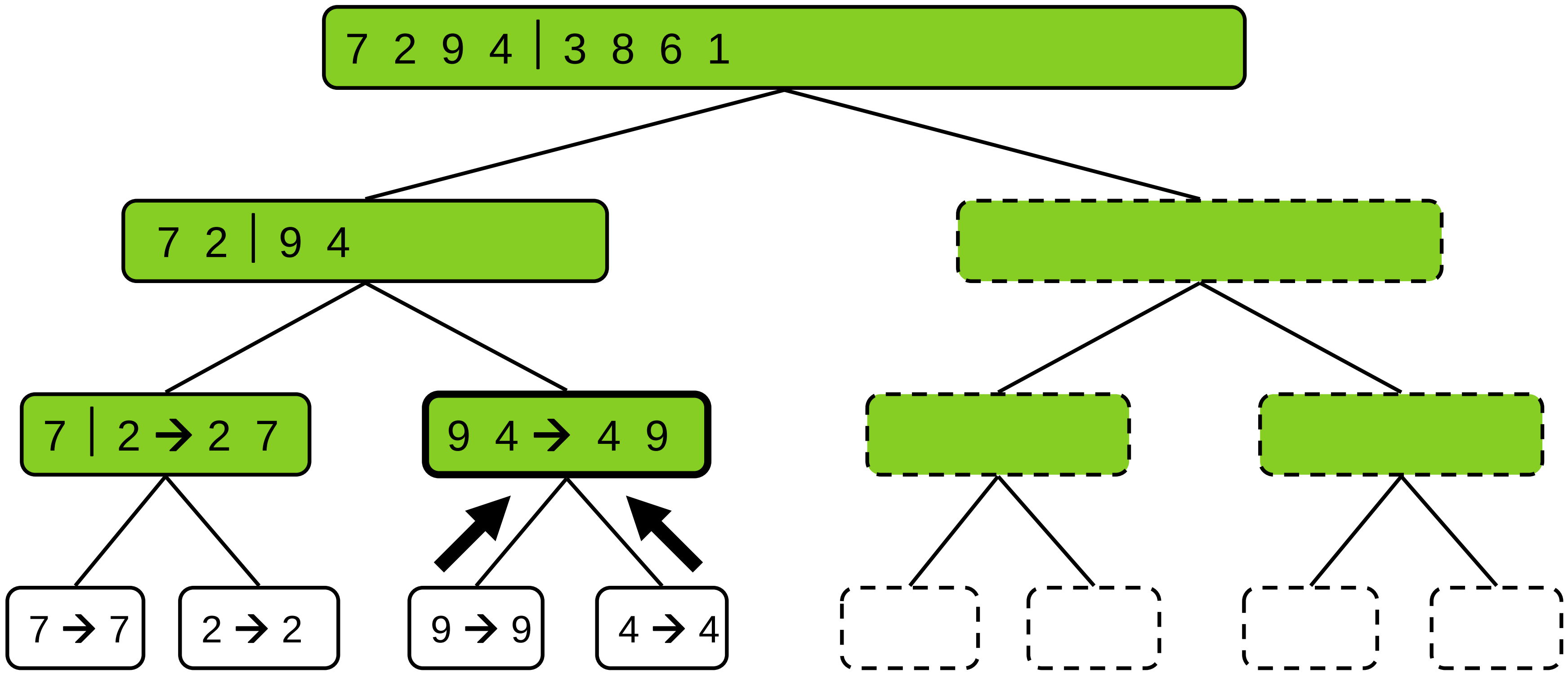
Merge Sort: Example (7)

- Merge



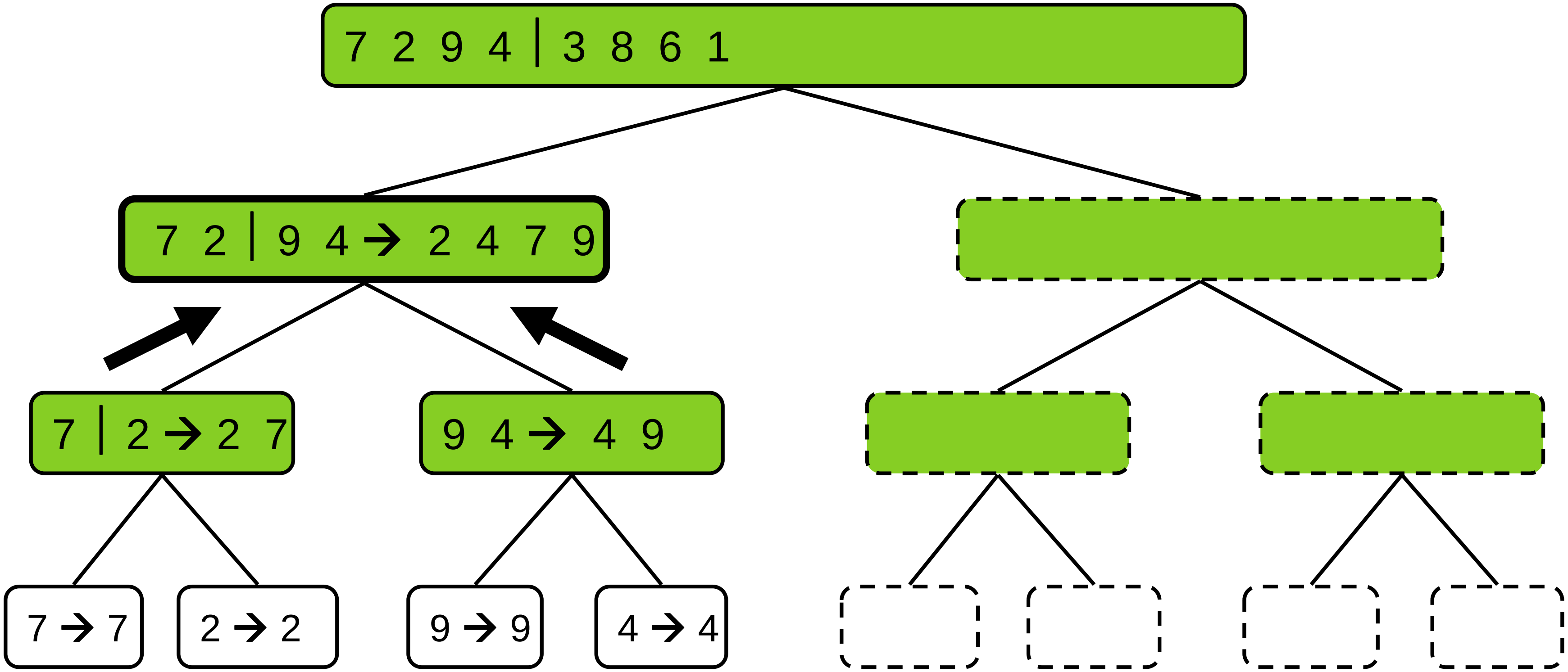
Merge Sort: Example (8)

- Recursive call, …, base case, merge



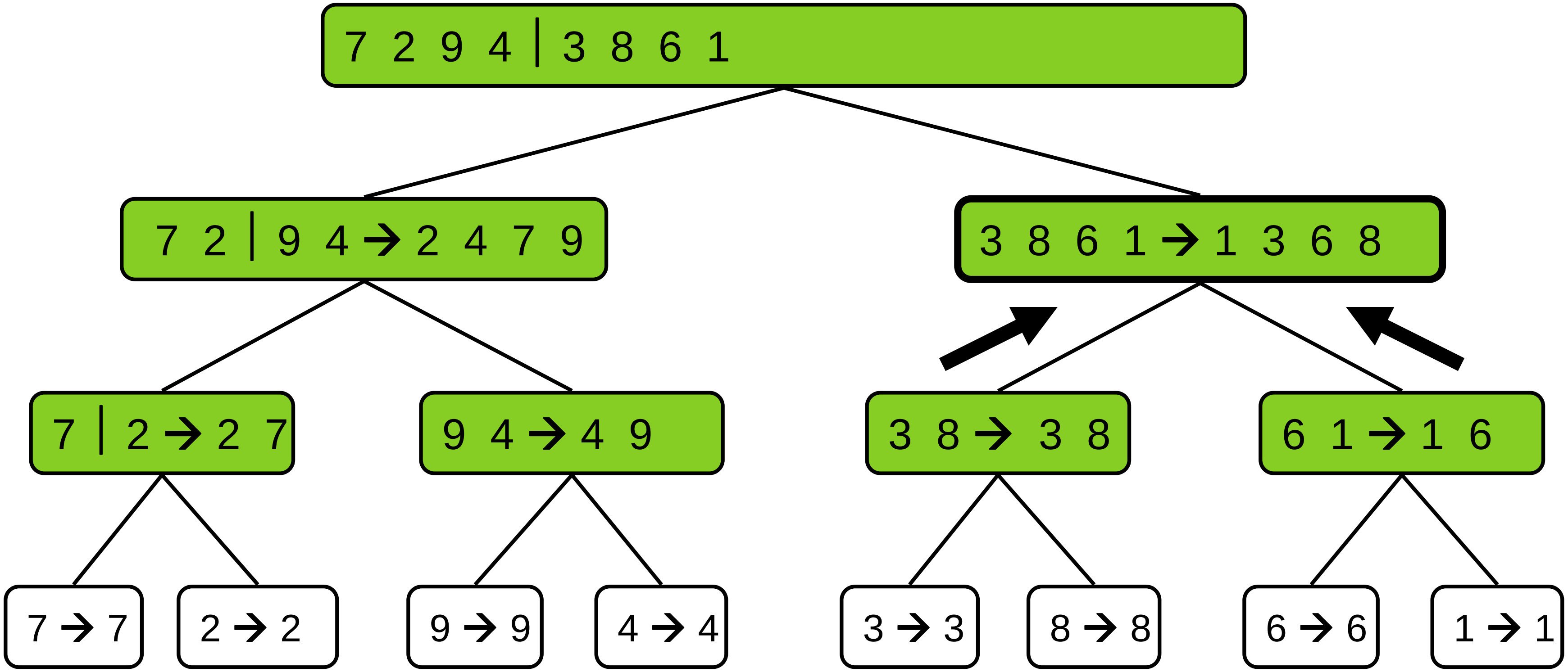
Merge Sort: Example (9)

- Merge



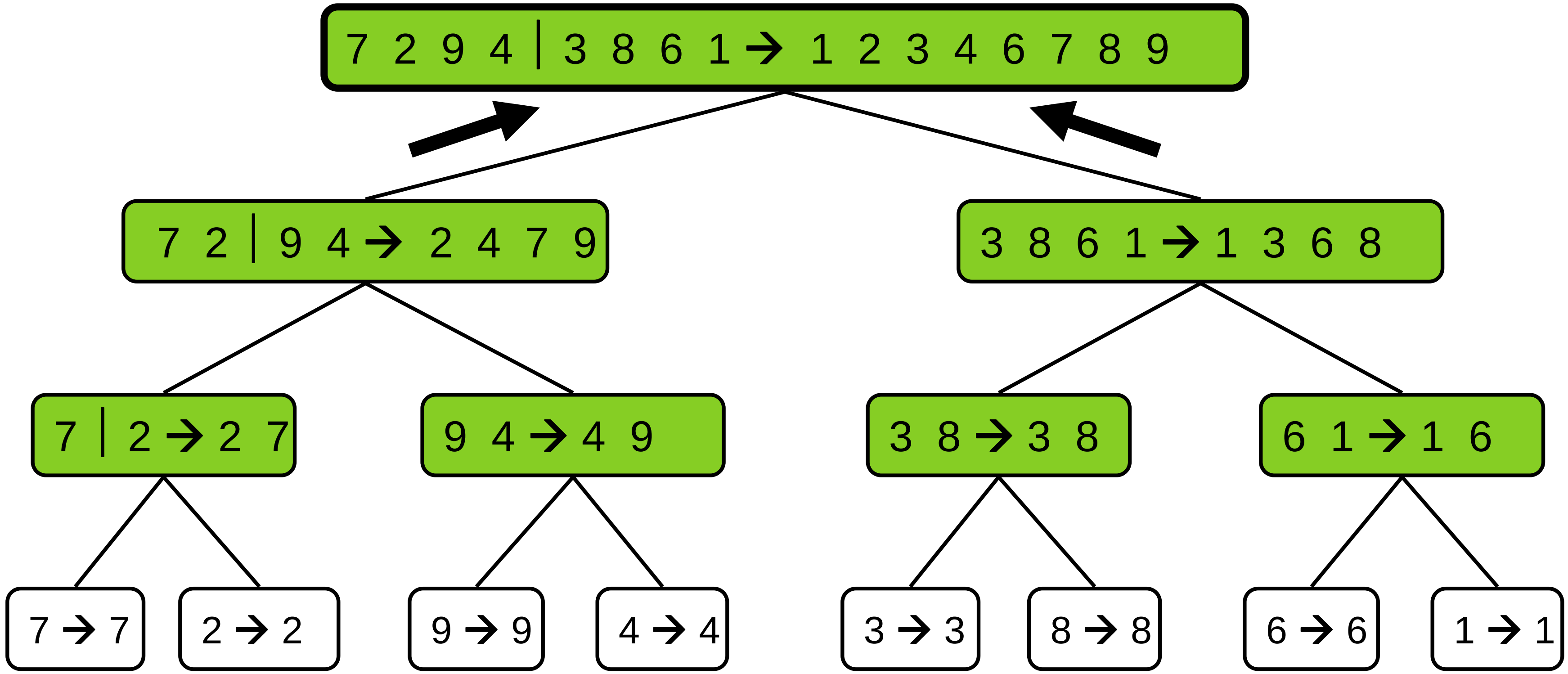
Merge Sort: Example (10)

- Recursive call, …, merge, merge



Merge Sort: Example (11)

- Final merge and return



Merge Sort Pseudocode

```
function mergeSort(A):
    // Input: an unsorted array a
    // Output: array a in sorted order
    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 Pseudocode (2)

```
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 Recurrence Relation

- Steps to merge sort:
  1. Recursively merge sort the left half of the list
  2. Recursively merge sort the right half of the list
  3. Merge both halves together
- Let T(n) be the running time of merge sort on an input of size n
- T(n) = step 1 + step 2 + step3
- Notice that steps 1 and 2 are simply merge sorts on half the input and step 3 runs in O(n) time
- $T(n) = T(n/2) + T(n/2) + O(n)$

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

Merge Sort Recurrence Solution

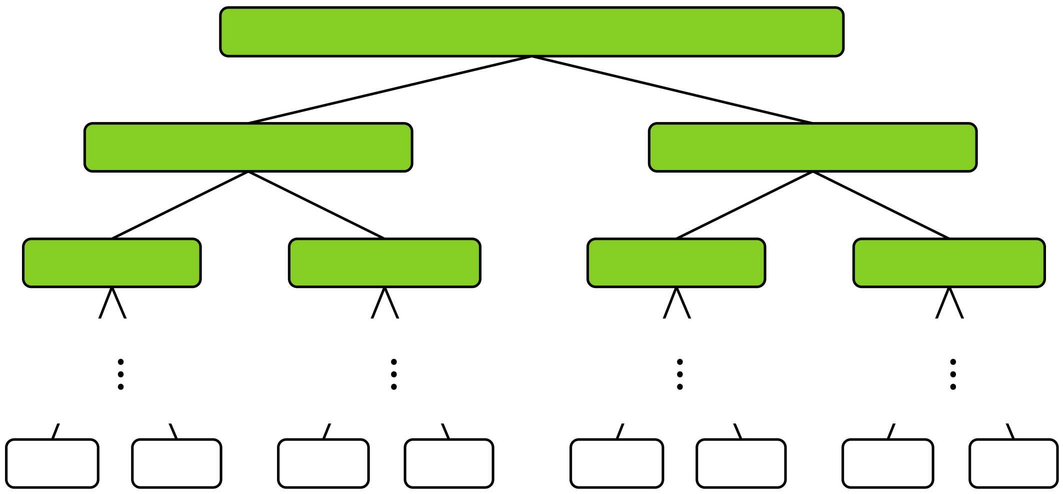
- Recurrence relation:
  - Base case: $T(1) = c_1$
  - General case: $T(n) = 2T(n/2) + O(n)$
- Plug 'n' chug for a solution:

$$
\begin{array}{llll}
T(1) & = c_1 & & = c_1 \\
T(2) & = 2\,T(1) + 2 & & = 2c_1 + 2 \\
T(4) & = 2\,T(2) + 4 & = 2(2c_1 + 2) + 4 & = 4c_1 + 8 \\
T(8) & = 2\,T(4) + 8 & = 2(4c_1 + 8) + 8 & = 8c_1 + 24 \\
T(16) & = 2\,T(8) + 16 & = 2(8c_1 + 24) + 16 & = 16c_1 + 64 \\
T(n) & & & = nc_1 + n\log n \\
 & & & = \mathbf{O(n\log n)}
\end{array}
$$

Analysis of Merge Sort

- To understand why merge sort is $O(n \log n)$, notice that the **height h of the merge sort tree is $O(\log n)$** since it forms a perfect binary tree
- Overall amount of work done **at each depth k is $O(n)$** to partition and merge 2^k sequences of size $n/2^k$

depth	sequences	size
0	1	n
1	2	$n/2$
2	4	$n/4$
$\vdots$	$\vdots$	$\vdots$
k	2^k	$n/2^k$



Solving Recurrence Relations

- To determine that merge sort was O(n log n), we determined the recurrence relation and used plug ‘n’ chug to conjecture a solution
  - $T(n) = T(n/2) + O(n) \rightarrow O(n \log n)$

Solving Recurrence Relations (2)

- Plug ‘n’ chug on recurrence relations is sort of a pain, but it turns out there’s an easier way of solving recurrence relations…

Master Theorem

- We will cover how to use Master Theorem, but the proof for why it works is in Dasgupta on pages 58-60

The Master Theorem

Where…

- a is the number of subproblems
- n/b is the size of each subproblem (if n/b is a fraction, b will be a whole number)
- n^d is the work done to prepare the subproblems and assemble the sub-results

Let $a \geq 1$, $b > 1$, $d \geq 0$, and $T(n)$ be a monotonically increasing function of the form:

$$T(n) = aT(n/b) + \Theta(n^d)$$

Then:

$T(n)$ is $\Theta(n^d)$ if $a < b^d$

$T(n)$ is $\Theta(n^d \log n)$ if $a = b^d$

$T(n)$ is $\Theta(n^{\log_b a})$ if $a > b^d$

Applying the Master Theorem

$$T(n) = aT(n/b) + \Theta(n^d)$$

$$T(n) \text{ is } \Theta(n^d) \qquad \text{if } a < b^d$$

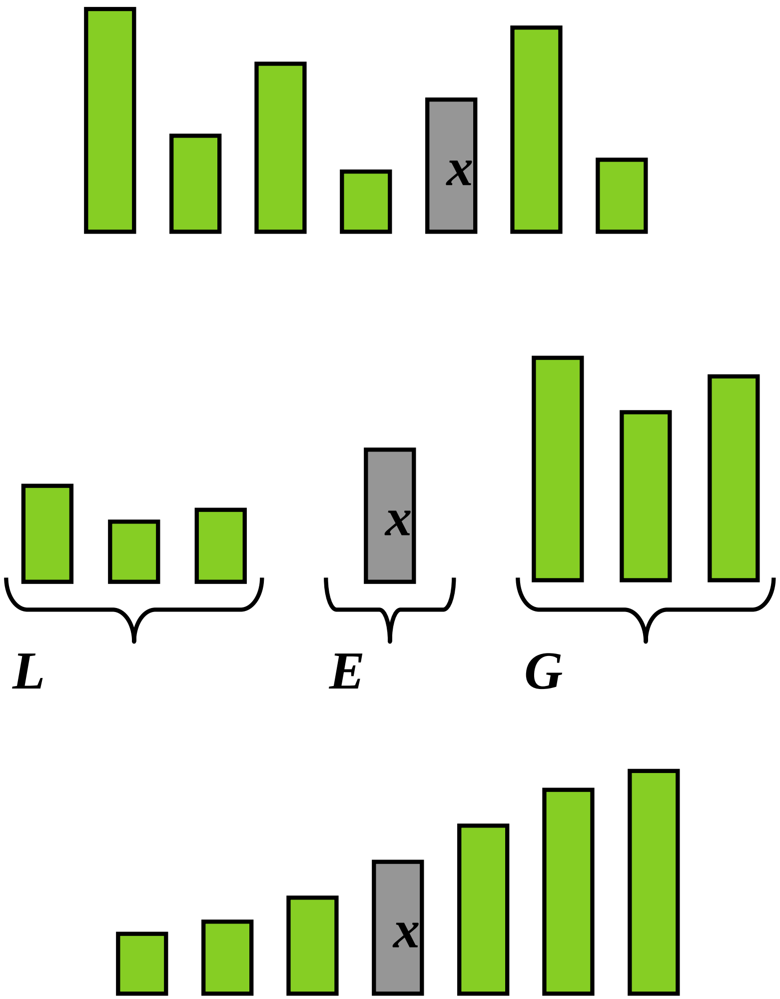
$$T(n) \text{ is } \Theta(n^d \log n) \qquad \text{if } a = b^d$$

$$T(n) \text{ is } \Theta(n^{\log_b a}) \qquad \text{if } a > b^d$$

- Merge sort's recurrence relation is $T(n) = 2T(n/2) + O(n^1)$
  - So, $a = 2$, $b = 2$, $d = 1$ and therefore $a = b^d$
  - T(n) is $\Theta(n^d \log n) = \Theta(n^1 \log n) = \Theta(n \log n)$

Quick Sort

- Quick sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - **Divide**: pick a random element x (called the pivot) and partition sequence S into
    - L elements less than x
    - E elements equal to x
    - G elements greater than x
  - **Recur**: quicksort L and G
  - **Conquer**: join L, E and G

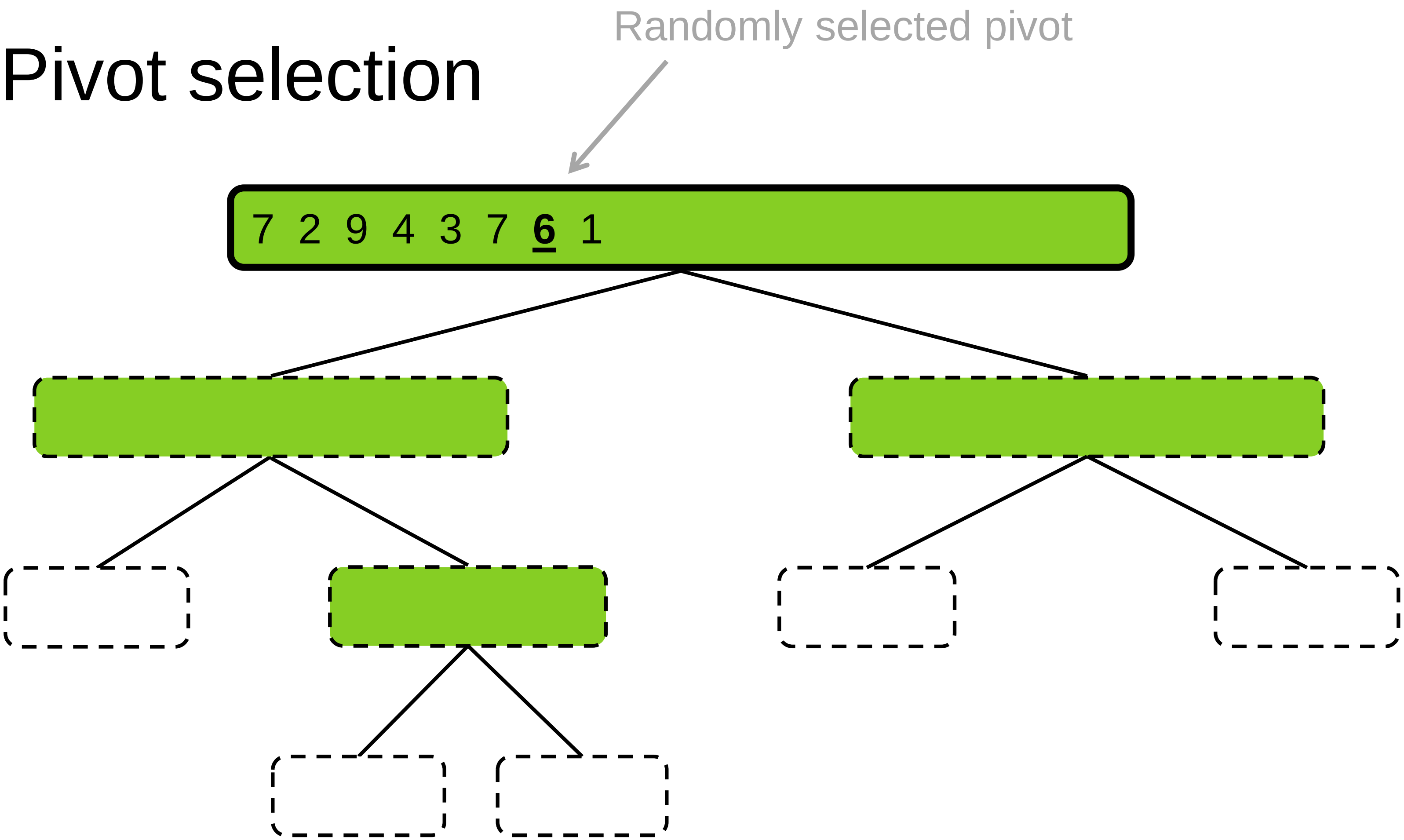


Quick Sort: Example

- Suppose we want to sort [7, 2, 9, 4, 3, 8, 6, 1]

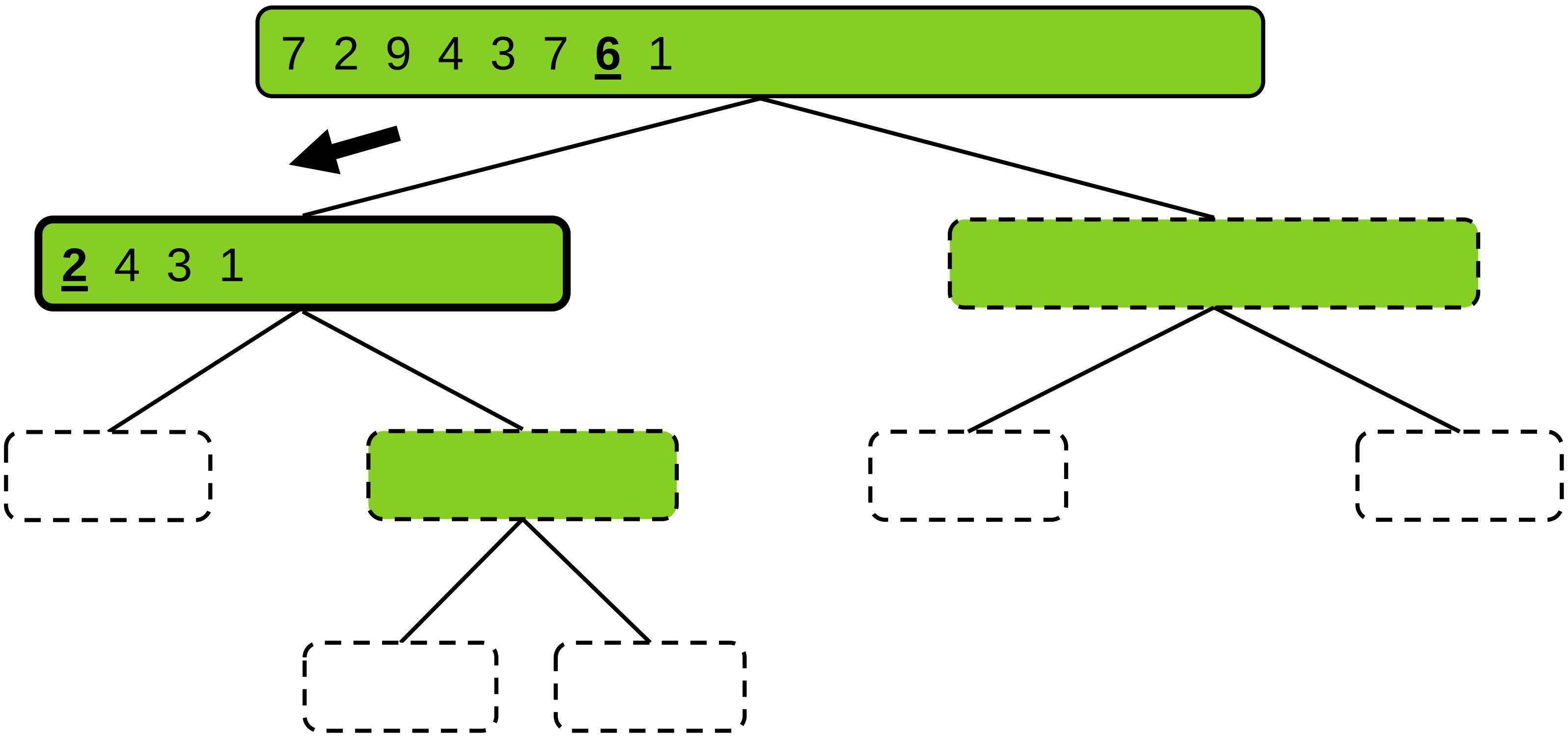
Quick Sort: Example

- Pivot selection



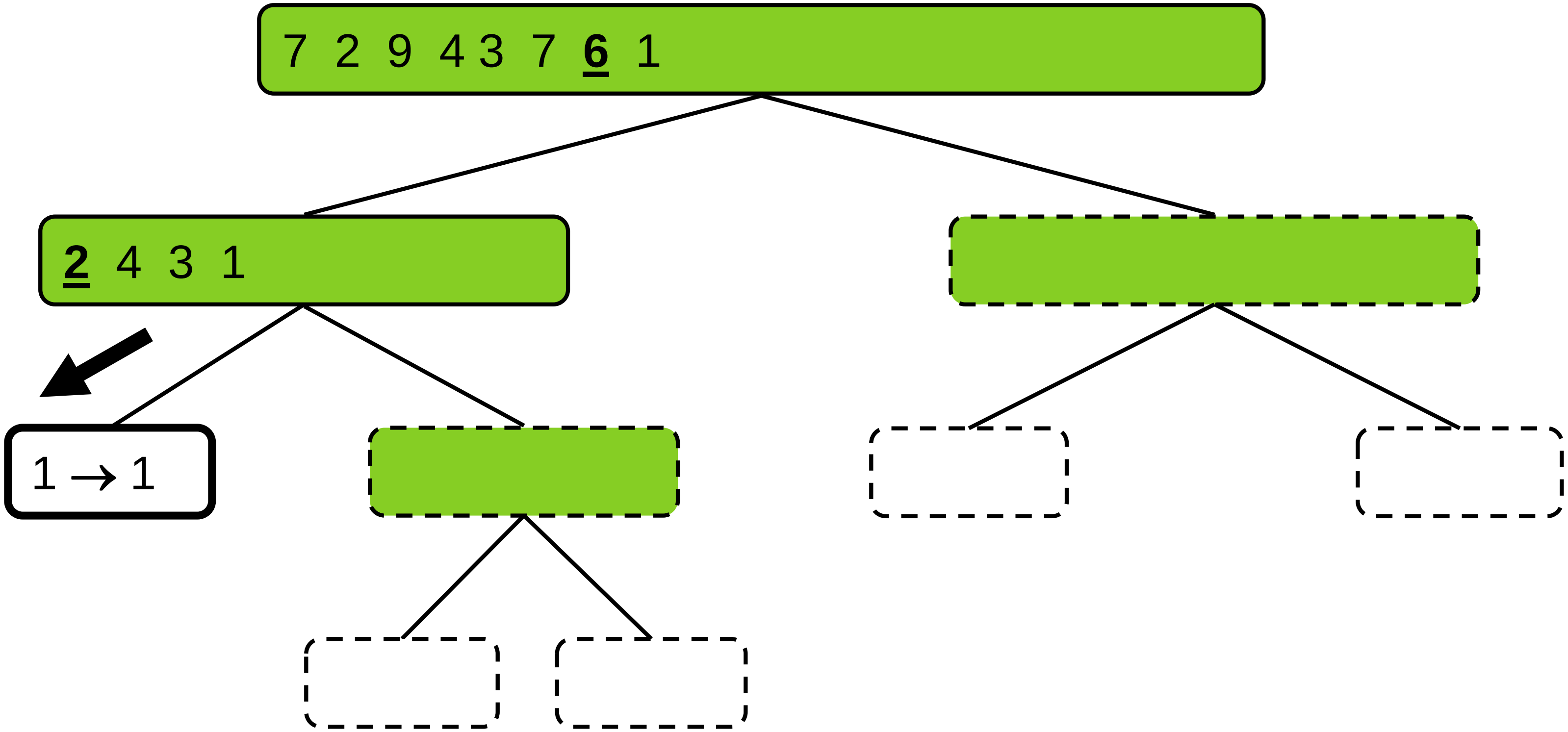
Quick Sort: Example (2)

- Partition, recursive call, pivot selection



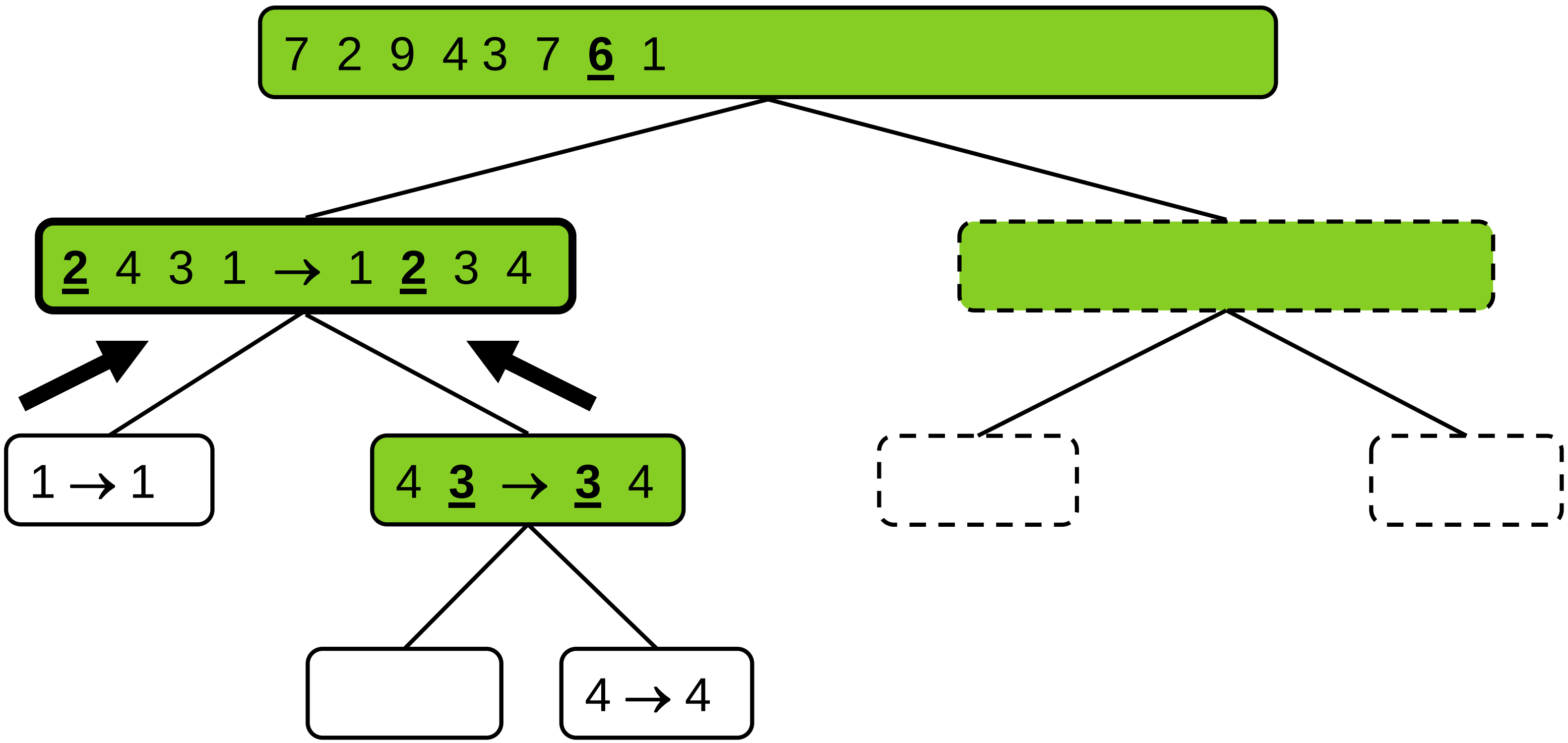
Quick Sort: Example (3)

- Partition, recursive call, base case



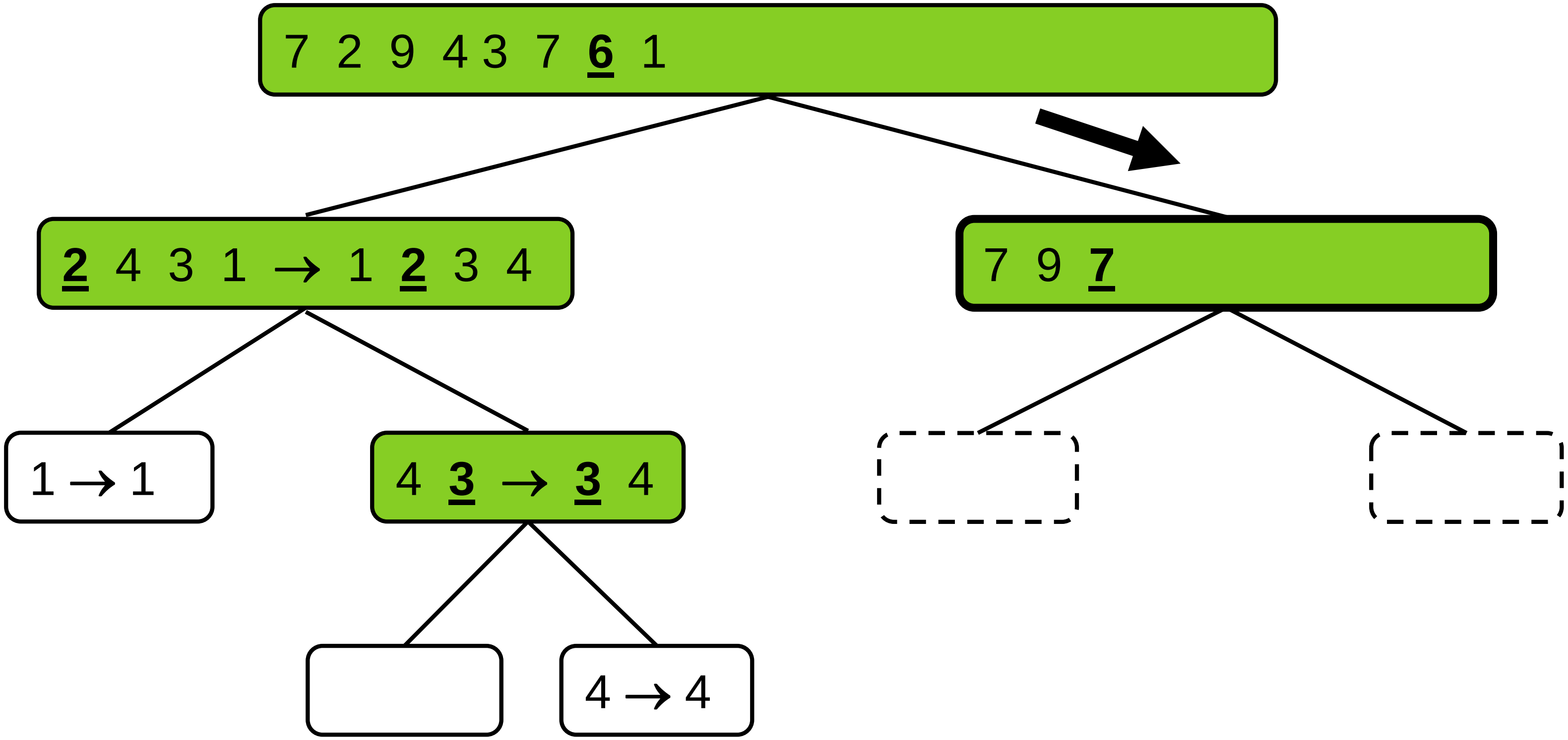
Quick Sort: Example (4)

- Recursive call, …, base case, join



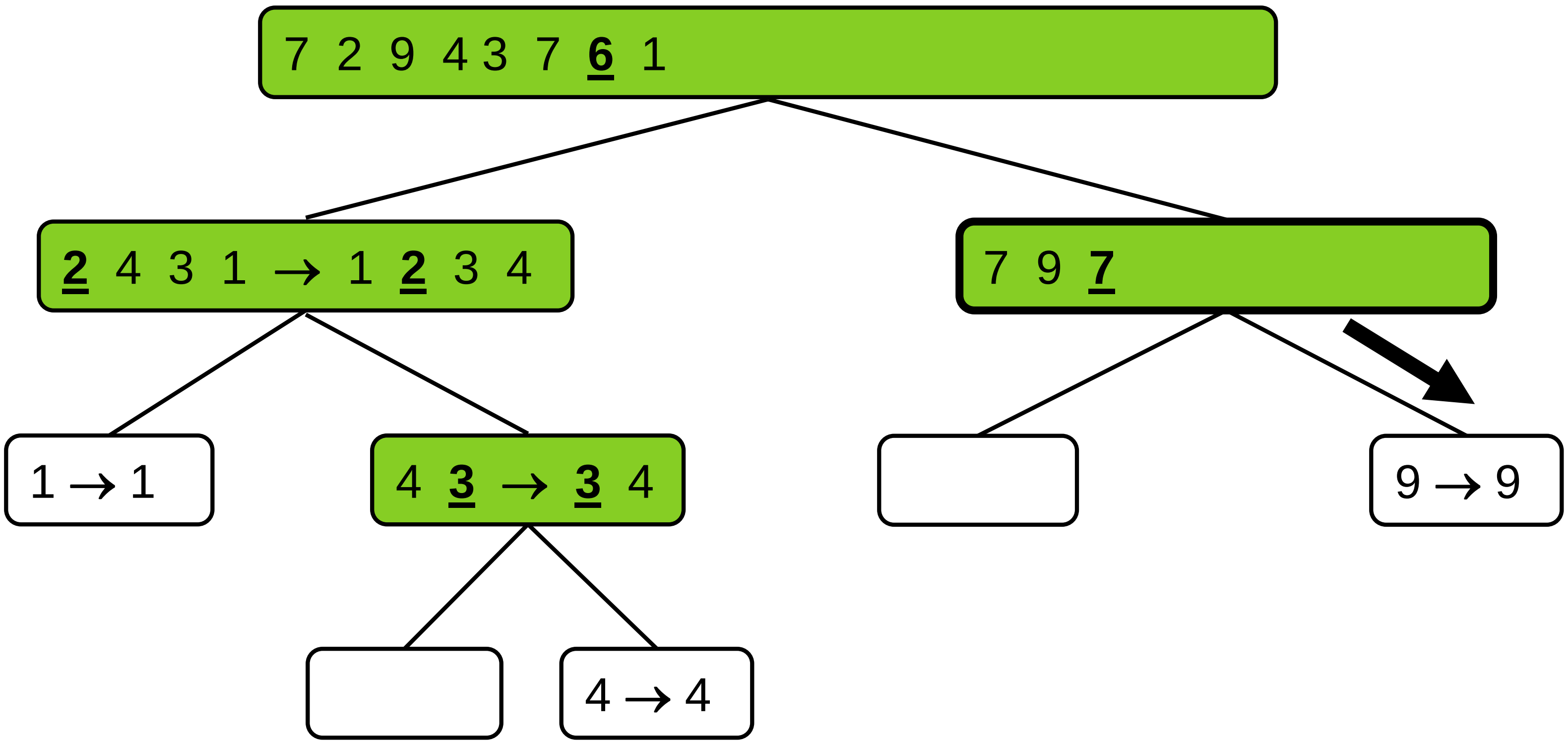
Quick Sort: Example (5)

- Recursive call, pivot selection



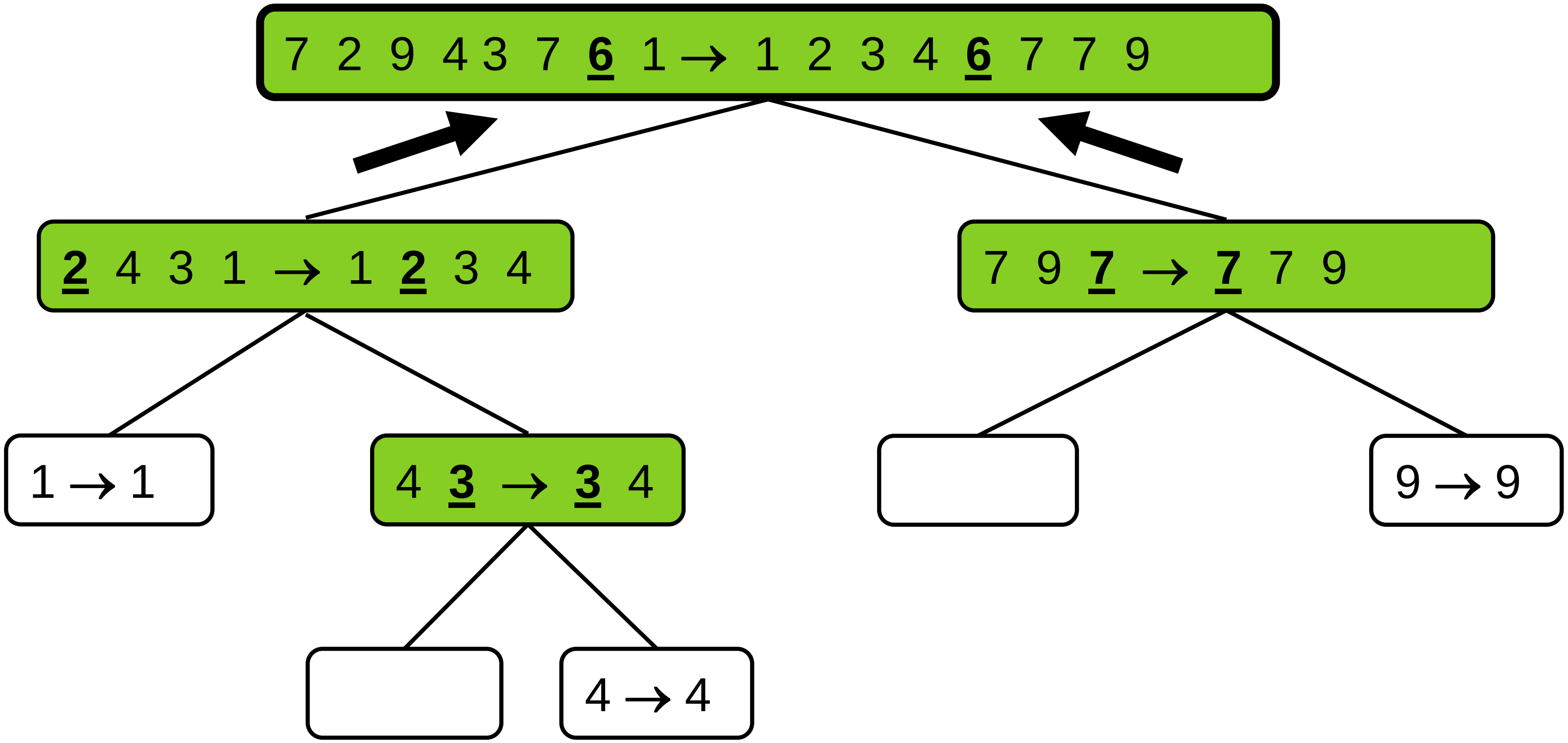
Quick Sort: Example (6)

- Partition, …, recursive call, base case



Quick Sort: Example (7)

- Join, join



Quick Sort Pseudocode

```
function quick_sort(A):
   // Input: unsorted list
   // Output: sorted list
   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)
```

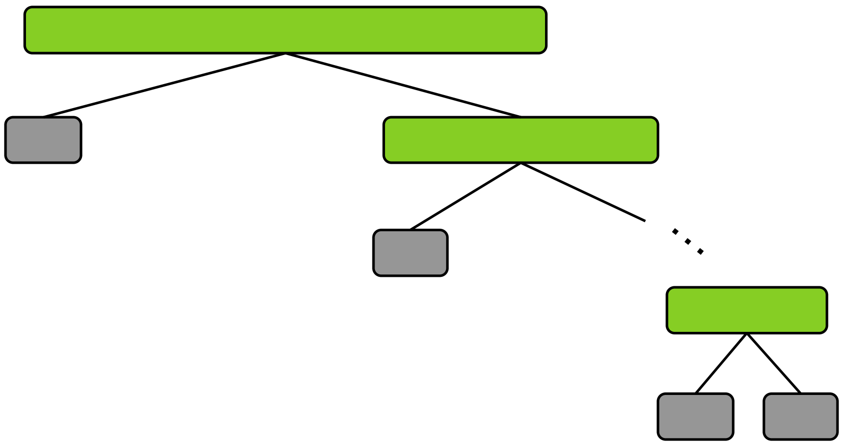
Worst-case Running Time

- The **worst case** for quick-sort occurs when the pivot is a **unique minimum** or **maximum** element
- One of $\mathbf{L}$ and $\mathbf{G}$ has size $\mathbf{n}$ - 1 and the other has size 0
- The running time is proportional to the sum

$$\mathbf{n} + (\mathbf{n} - 1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $\mathbf{O(n^2)}$

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



Expected-case Running Time

- To find an upper bound on the expected running time, assume there are no duplicates (if there are duplicates, the recursive calls will be even smaller)
- There are n possible unique ways quick sort will make its recursive calls:
  - |L| = 0, |G| = n-1
  - |L| = 1, |G| = n-2

    ⋮
  - |L| = n-1, |G| = 0
- Since there are n possible ways quick sort will recur, each has probability 1/n.
- We average over all possible splits and note that an additional linear amount of work is done:

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

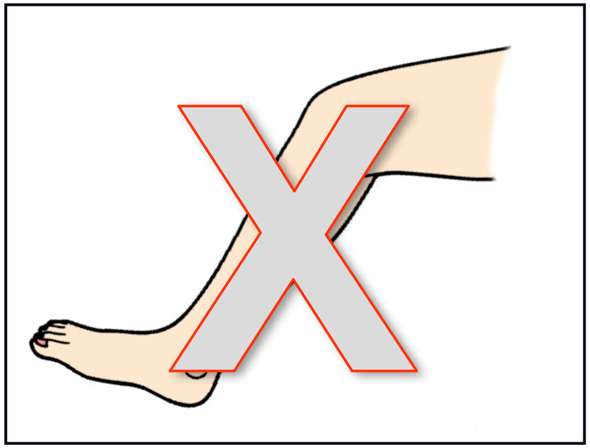
- The solution to this recurrence relation is (just trust us):

$$T(n) = 2n\ln n = 1.39n\log_2 n = O(n\log n)$$

In-Place Quick Sort?

```
function quick_sort(A):
    // Input: unsorted list
    // Output: sorted list
    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)
```



Hmmm… This whole "LEG" business doesn't seem very in-place!

Let's see if we can partition the array in-place!

In-Place Quick Sort!

```
function partition(A, low, high):
    // Input: list A with bounds low and high for the area
               we're partitioning
    // Output: The index of the pivot, once the array has been
                partitioned
    pivotIndex = random index between low and high
    pivotValue = A[pivotIndex]
    swap A[pivotIndex] and A[high]    // move the 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 Quick Sort!

```
function quick_sort(A, low, high):
    // Input: unsorted list
    // Output: sorted list between low and high
    if low < high:
        pivotIndex = partition(A, low, high)
        quicksort(A, low, pivotIndex – 1)
        quicksort(A, pivotIndex + 1, high)
```

How fast can we sort?

- Both merge and quick sort are $O(n \log n)$
- Can we do better? No! (sort of)

- **Claim**: Any comparison sort must in the worst case make at least $\Omega(n \log n)$ comparisons to sort n keys.
- Let's try to prove this!

Comparative Sorting is Ω(n log n)

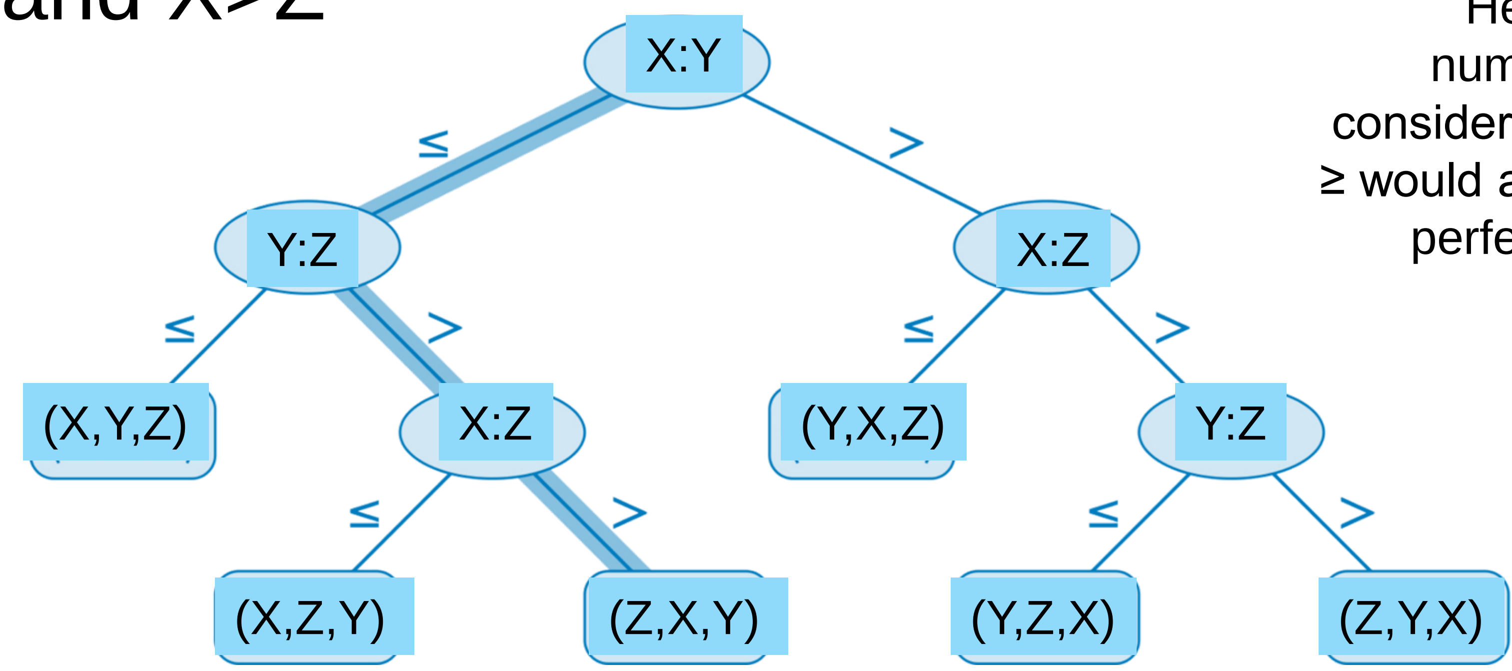
- A sorting algorithm, viewed abstractly, takes a sequence of keys k_1, k_2, ... k_n and outputs the permutation that will sort them.

- For a given n, we can represent the optimal algorithm as a perfect binary decision tree:
  - internal nodes are comparisons of two keys
  - leaves are the correct permutations.

- Sorting a particular sequence is equivalent to walking the tree from the root to a leaf. The number of comparisons in the worst case is the height of the tree.

Comparison Tree

- The path to any leaf shows the minimum number of comparisons necessary
- If (Z,X,Y) is the proper sort, then X≤Y, Y>Z, and X>Z



Here equal numbers are considered ≤, but ≥ would also work perfectly well!

Comparative Sorting Proof (2)

- n! permutations of a given sequence, so the tree has n! leaves.
- We know that a perfect binary tree with height h has 2^h leaves. Thus, a tree with n! leaves has height log (n!)
- Stirling's Formula:

$$n! \geq n^n e^{-n}$$

$$\log(n!) \geq \log(n^n e^{-n})$$

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

- Therefore the height (and number of comparisons in the worst case) is $\Omega(n \log n)$.

Non-comparative Sorting

- Sorting functions may need to accept a variety of inputs
  - Integers
  - Floats
  - Strings
  - Arrays
  - Other objects… People? Cows?
- As long as there is some way to compare elements in the input domain, we can apply one of the comparison-based sorting algorithms
- But sometimes, if we know something about our input domain in advance, we be a little cleverer in our sorting…
- Radix sort: great for positive integers!

Non-comparative Sorting (2)

- First let's think about an easier problem:
  - Given an array of integers between 0 and 9, sort the array
  - e.g. [5, 1, 6, 2, 3, 1] -> [1, 1, 2, 3, 5, 6]
- Since we know that every element is guaranteed to be an integer between 0 and 9, let's use that to our advantage!
  - Make an array of 10 buckets, one for each possible number
  - For each number x in the array, add it to the bucket at index x
  - Return the concatenation of all the buckets, in order

    [1, 1] + [2] + [3] + [5] + [6]
- What's the runtime of this? O(n)!

Radix Sort

- What about sorting integers of unknown size?
- Check out these two numbers

  258391 258492

- How would you compare these by hand?
  - Digit by digit, probably!
  - The 3 highest order bits are the same (258), so we keep going until we find that 4 is greater than 3, so 258492 must be greater than 258391
- Radix sort takes advantage of the digity-ness of integers, and also the fact that for any given digit, there are a constant number of options (0-9)

Radix Sort Example

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

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

- After sorting, concatenate all buckets in order to get a new list: **[8271, 7891, 273, 8735, 8736, 279]**
- Everything's now sorted – by 1's place

Radix Sort Example

- The list is now **[8271, 7891, 273, 8735, 8736, 279]**
- Now we repeat, sorting by 10's place

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

- New list: **[8735, 8736, 8271, 273, 279, 7891]**
- Everything's now sorted – by 10's and 1's places

Radix Sort Example

- The list is now **[8735, 8736, 8271, 273, 279, 7891]**
- Now we repeat, sorting by 100's place

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

- New list: **[8271, 273, 279, 8735, 8736, 7891]**
- Everything's now sorted – by 100's, 10's and 1's places

Radix Sort Example

- The list is now **[8271, 273, 279, 8735, 8736, 7891]**
- Now we repeat, sorting by 1000’s place
  - We treat numbers with fewer than 4 digits as if they had additional leading 0’s

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

- New list: **[273, 279, 7891, 8271, 8735, 8736]**
- We’ve finished sorting by all digits! The list is sorted!

Radix Sort

```
function radix_sort(A):
    //Input: unsorted list of positive integers
    //Output: sorted list
    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: $O(n*d)$, where d is the number of digits in the largest number.

Radix Sort

- Radix sort can also be applied to different kinds of inputs, other than positive integers in base-10
  - Octal (base-8) or hexadecimal (base-16) numbers
  - Strings: make 1 bucket for each valid letter or character
- The number of buckets don't need to be the same for each "round" of sorting
  - Sorting a deck of cards: use numbers as the first buckets (lowest order bit) and suits as second buckets (highest order bit)
- You can also represent just about anything as a binary sequence of 0's and 1's and radix sort using two buckets
  - But then the number of digits will dominate the runtime, and for very long sequences, that sucks

Summary of Sorting Algorithms

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

Readings

- Dasgupta
  - Section 2.1: A good complementary introduction to divide and conquer algorithms
  - Section 2.2: A review of recurrence relations and the master theorem
  - Section 2.3: Analysis of merge sort and a lower bound on comparative sorting

Outline

- Motivation
- Quadratic $O(n^2)$ Sorting
  - Selection Sort
  - Insertion Sort
- “Linearithmic” $O(n \log n)$ Sorting
  - Merge Sort
  - Quick Sort
- The Master Theorem
- Linear Sorting
  - Radix Sort