Sorting

CS16: Introduction to Data Structures & Algorithms

Summer 2021

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

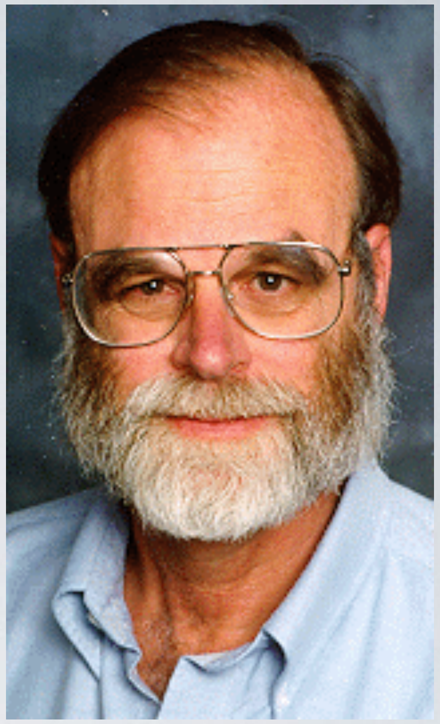
Sorting is Serious!

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
- TaichiSort from University of Washington and UT Austin
  - 61,000 records per *joule*

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)
- Basic part of data science workflows
  - Towns: sort by size, area, population, mean income, …
  - Batters: sort by average, home runs, OBP, wRC+, …

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 and efficiency

Let’s sort

Selection Sort

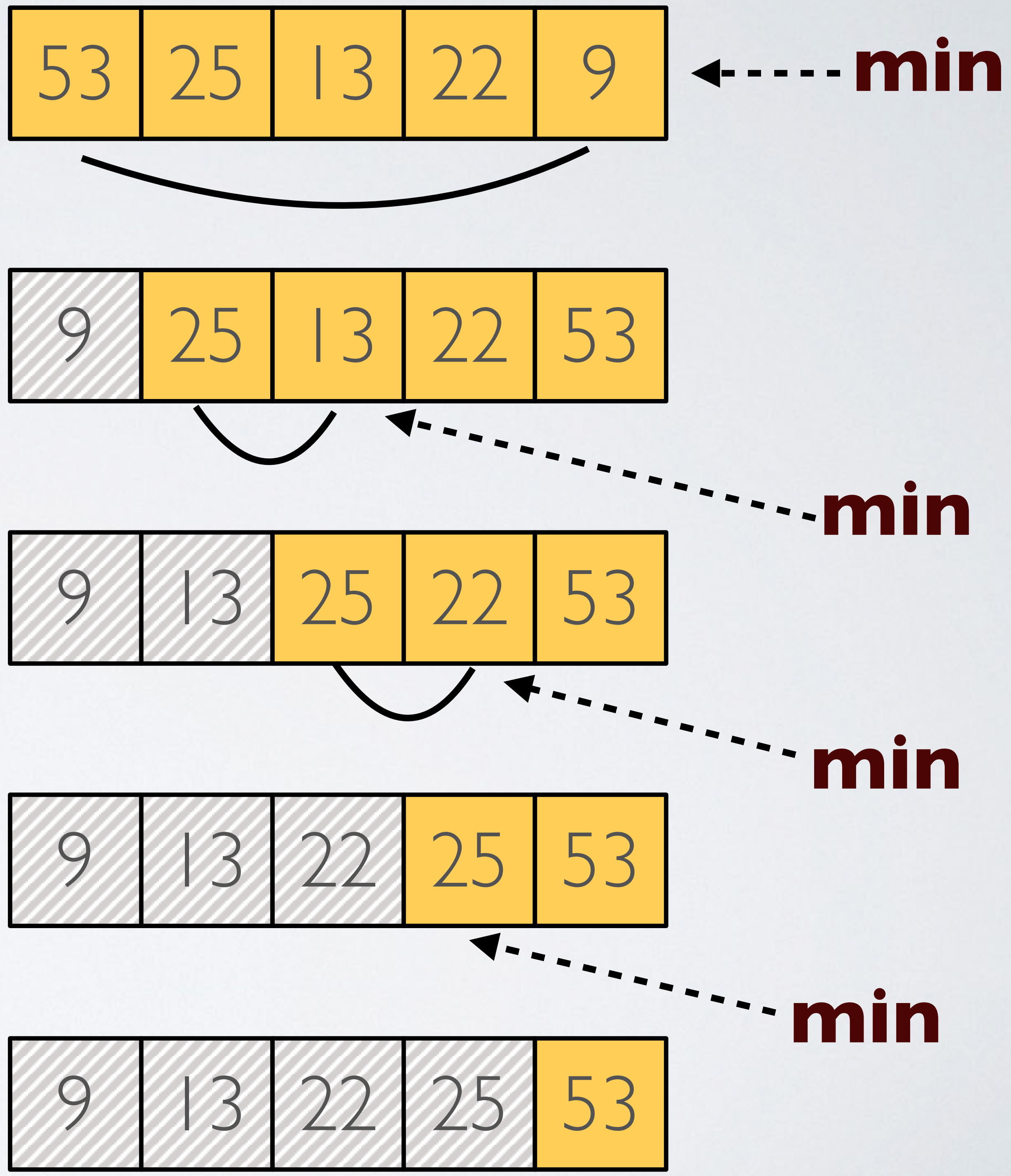
- Usually iterative and in-place
- Divides input array into two logical parts
  - elements already sorted
  - elements that still need to be sorted
- Selects smallest element & places it at index `0`
  - then selects second smallest & places it in index `1`
    - then the third smallest at index `2`,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

Selection Sort

- Advantages
  - Very simple
  - Memory efficient if in-place (swaps elements in array)
- Disadvantages
  - Slow

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]
```

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]
```

Runtime?

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]
```

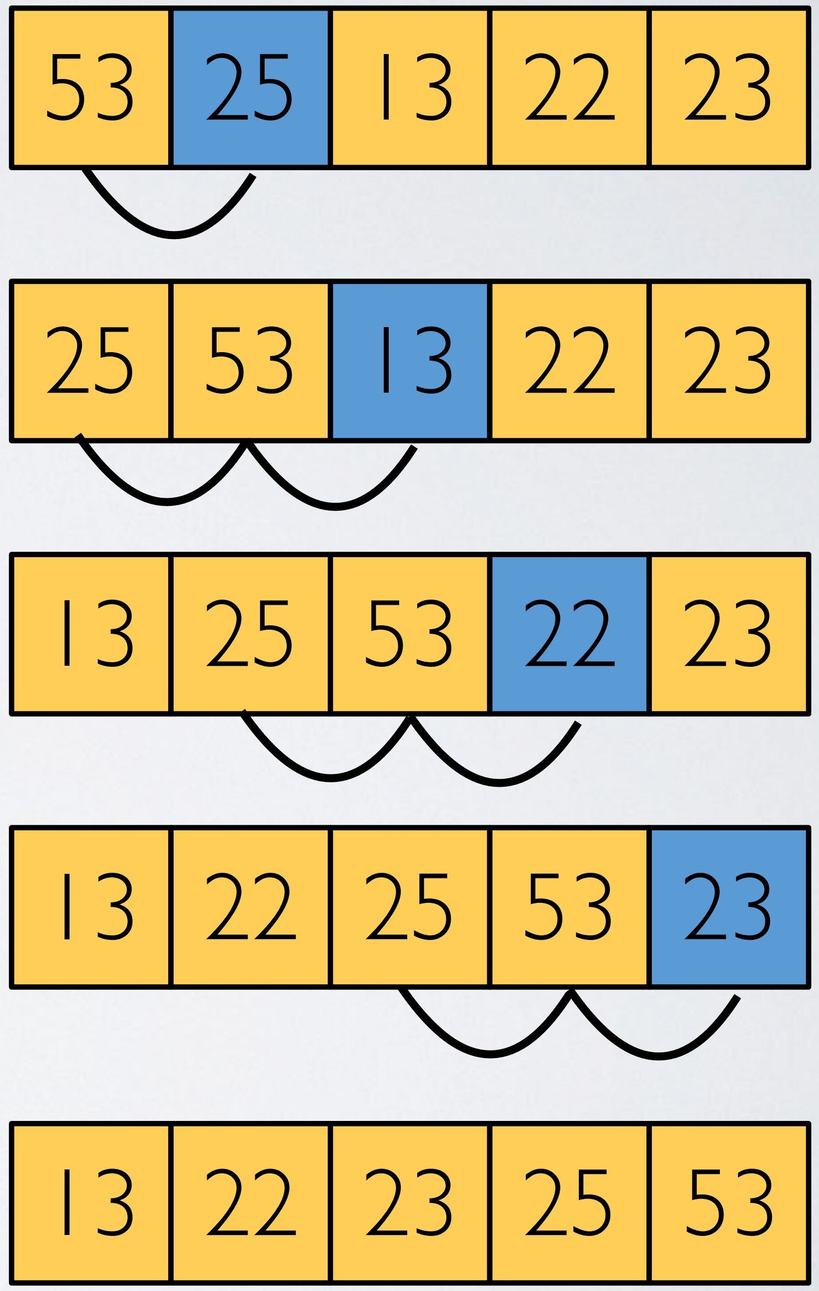
Runtime?
O(n^2)

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

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
```

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
```

Runtime?

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
```

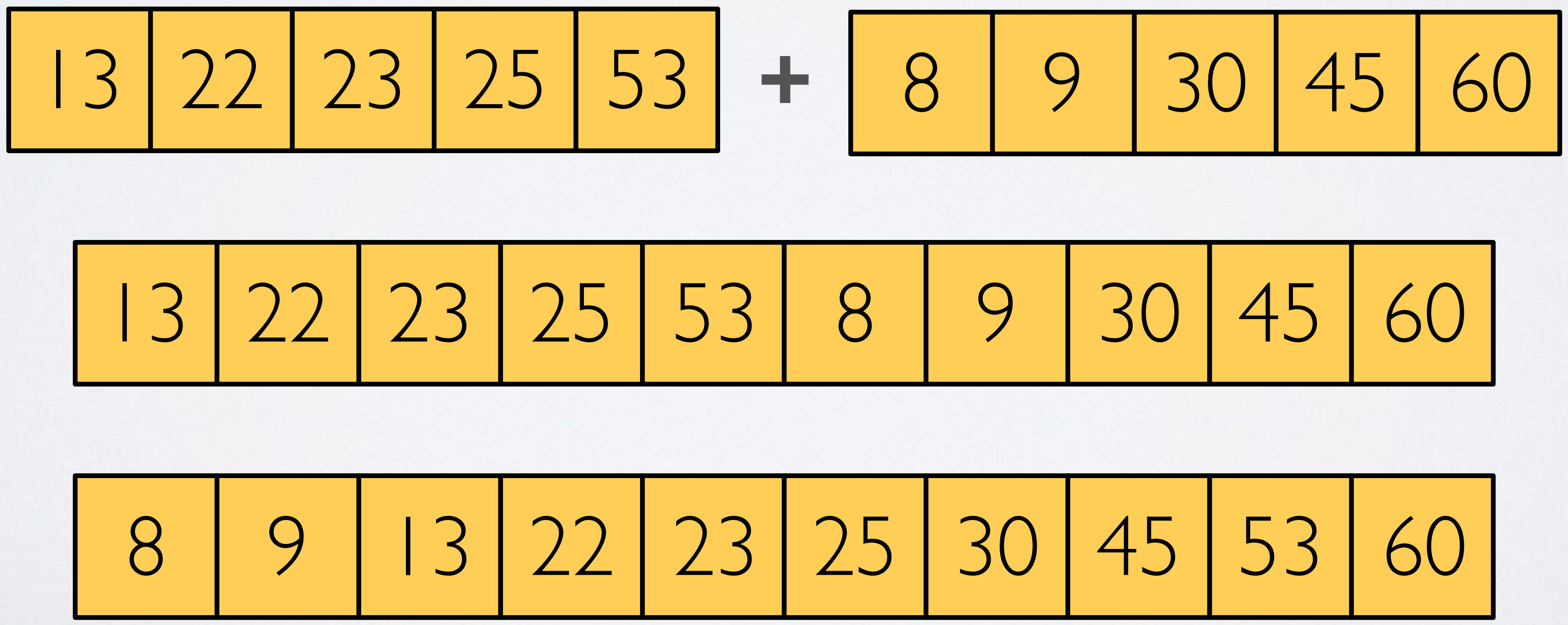
Runtime?

O(n^2)

Can we do better than O(n^2)?

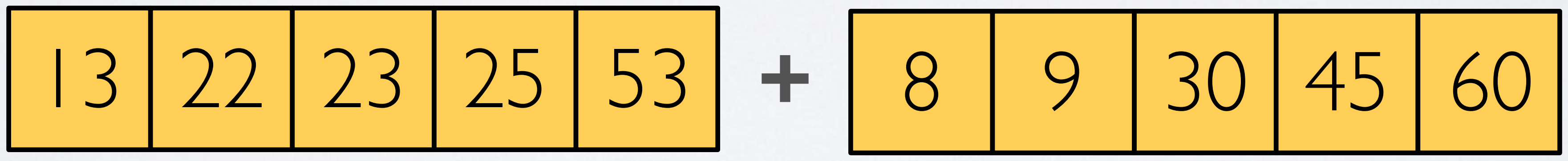
Merge

- Let's say we have two *sorted* lists
- How can we combine them to get one sorted list?
- Naive approach: combine then sort



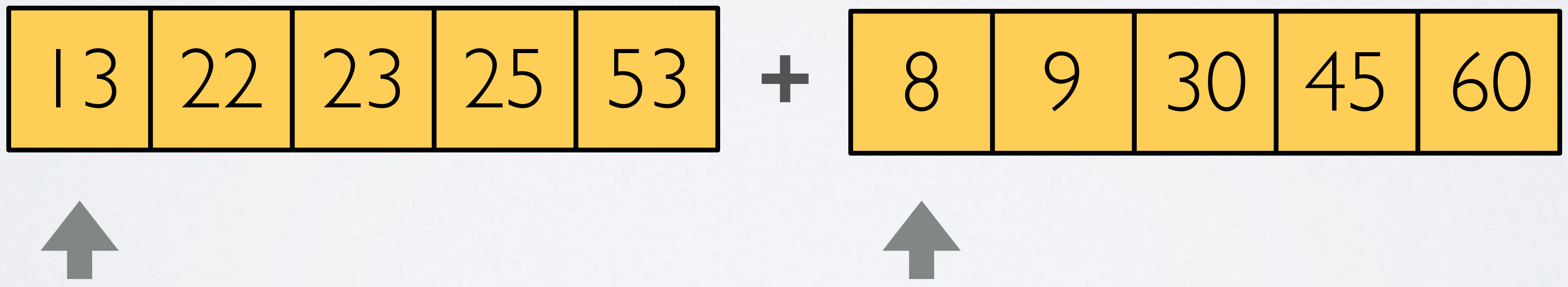
Better merge

- Go through both lists index by index
- Add smaller element to output list



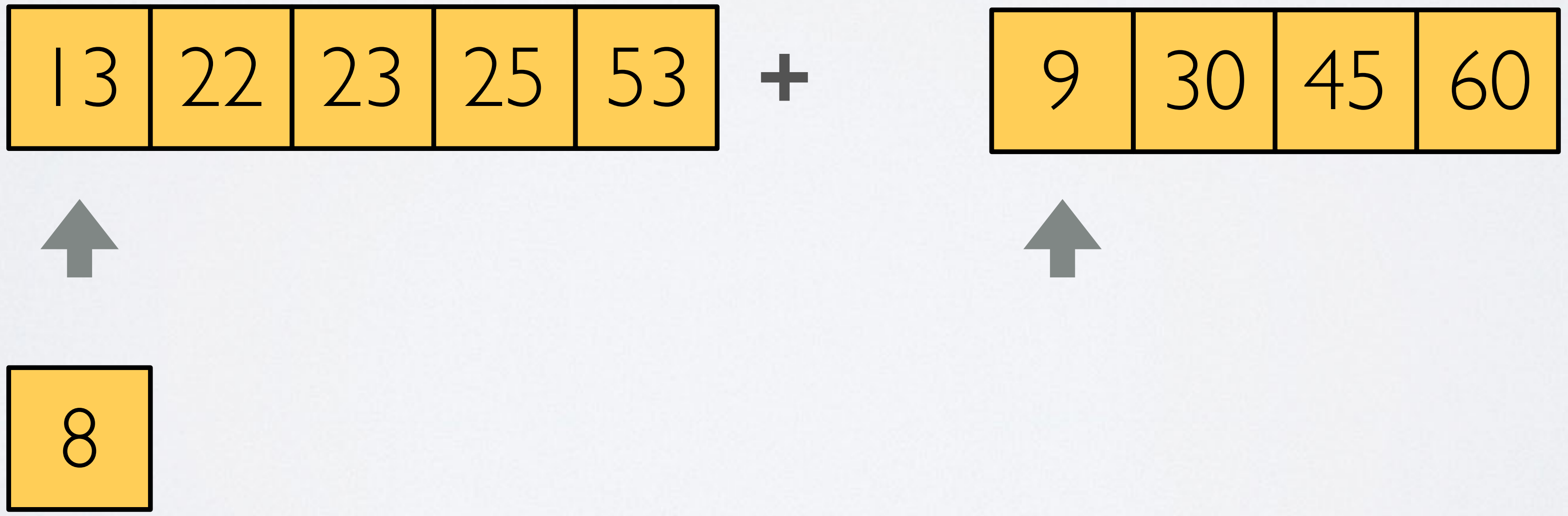
Better merge

- Go through both lists index by index
- Add smaller element to output list



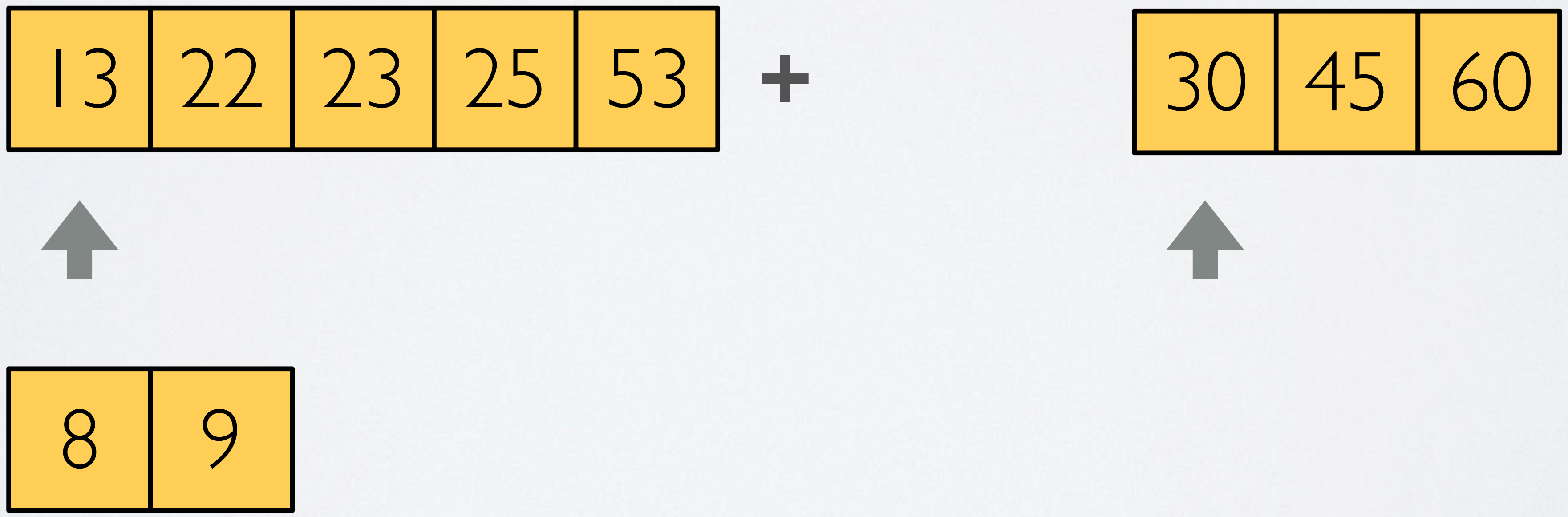
Better merge

- Go through both lists index by index
- Add smaller element to output list



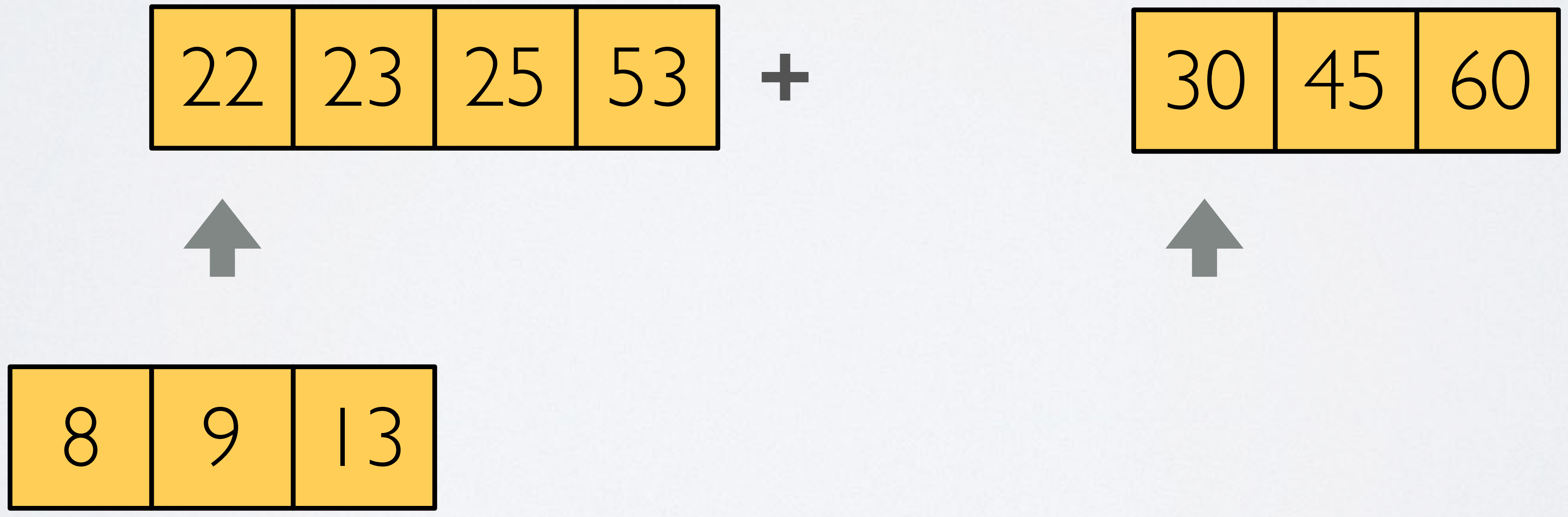
Better merge

- Go through both lists index by index
- Add smaller element to output list



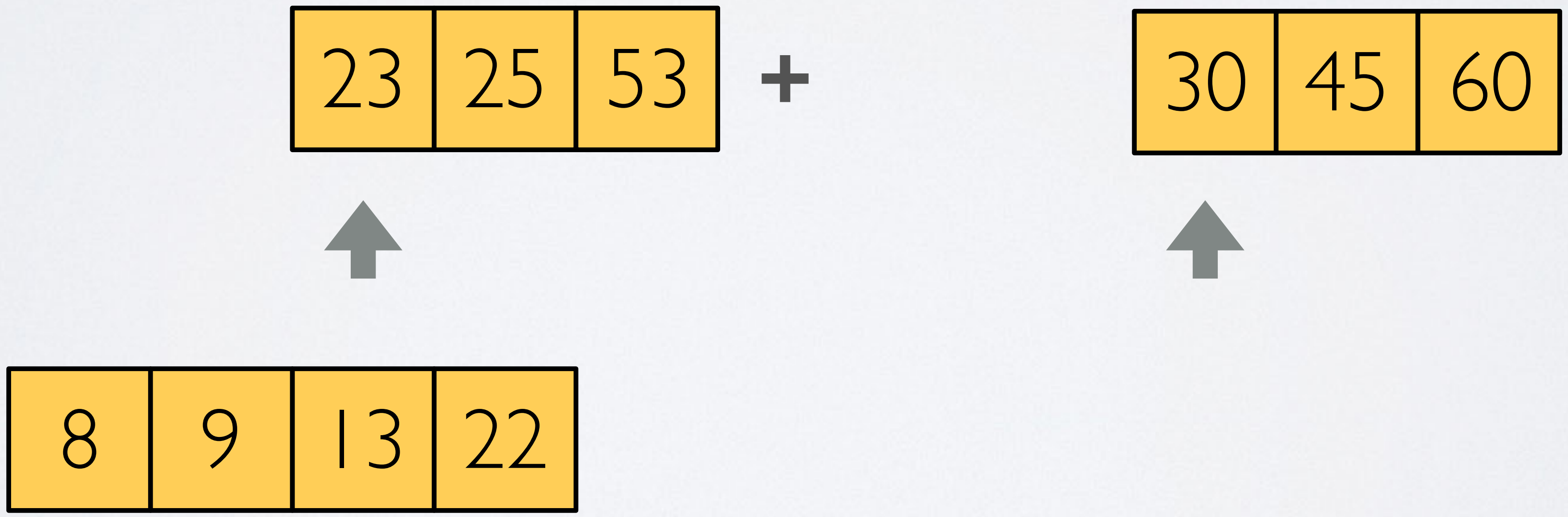
Better merge

- Go through both lists index by index
- Add smaller element to output list



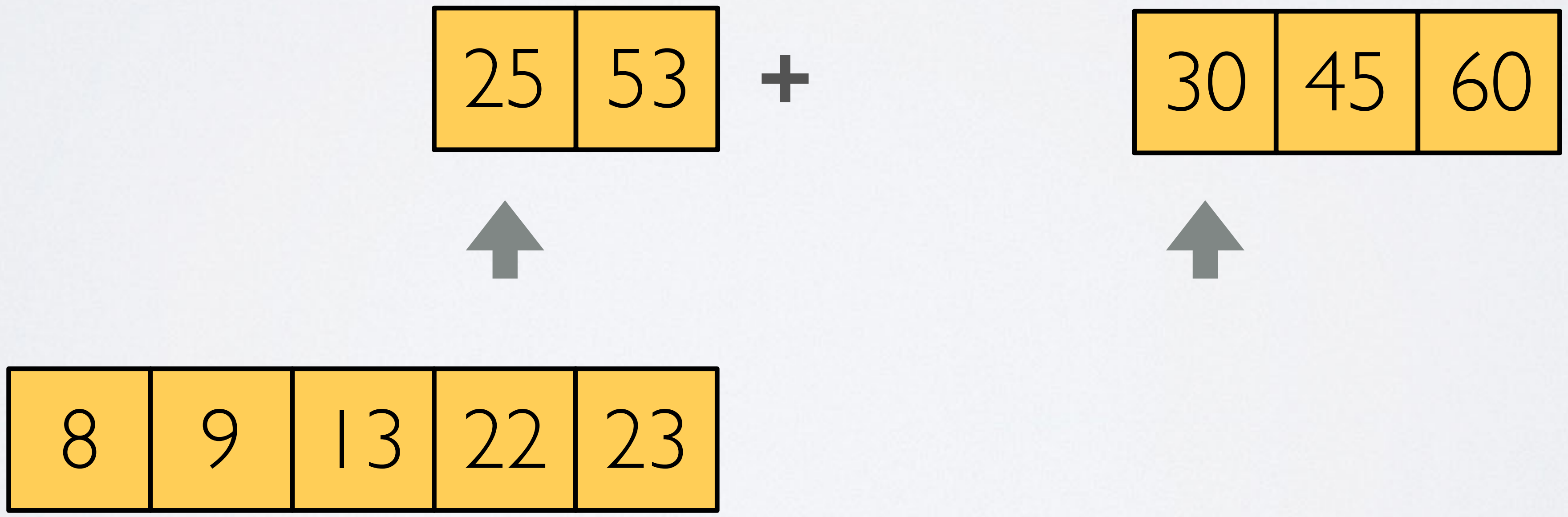
Better merge

- Go through both lists index by index
- Add smaller element to output list



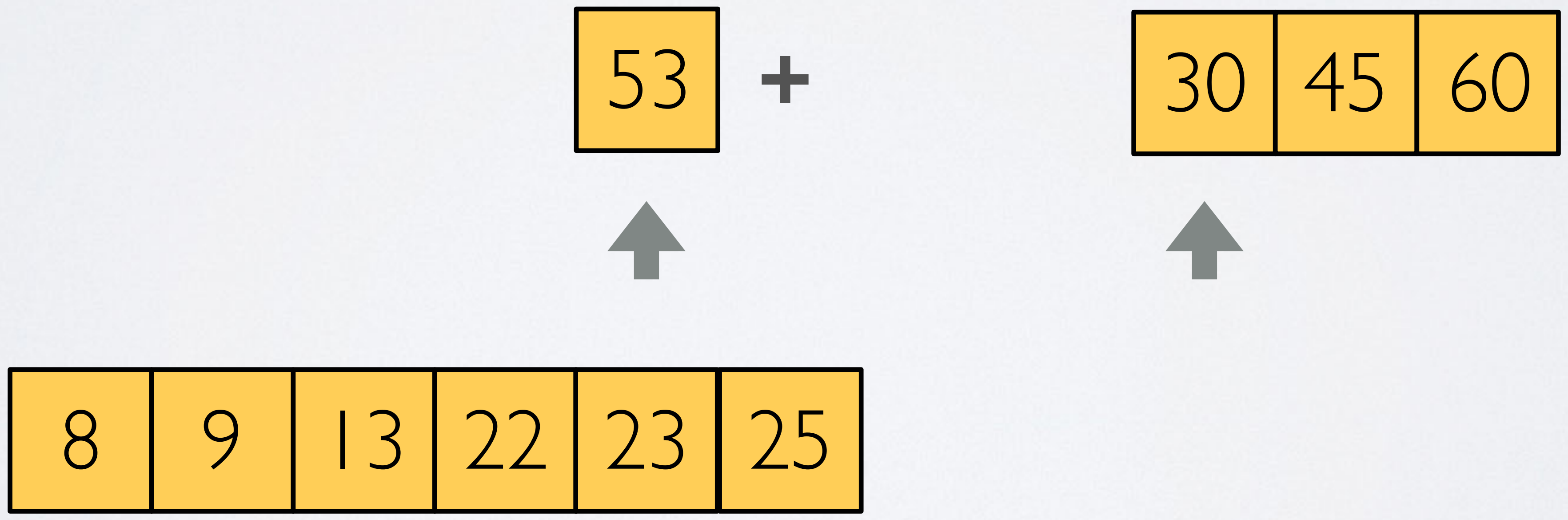
Better merge

- Go through both lists index by index
- Add smaller element to output list



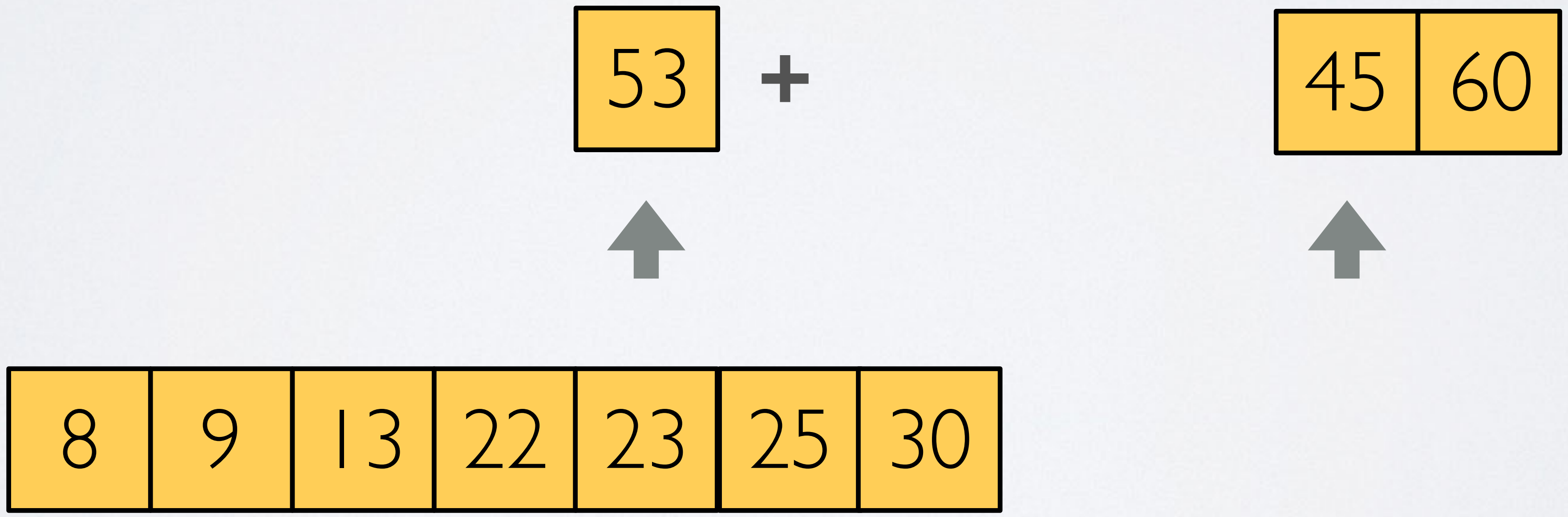
Better merge

- Go through both lists index by index
- Add smaller element to output list



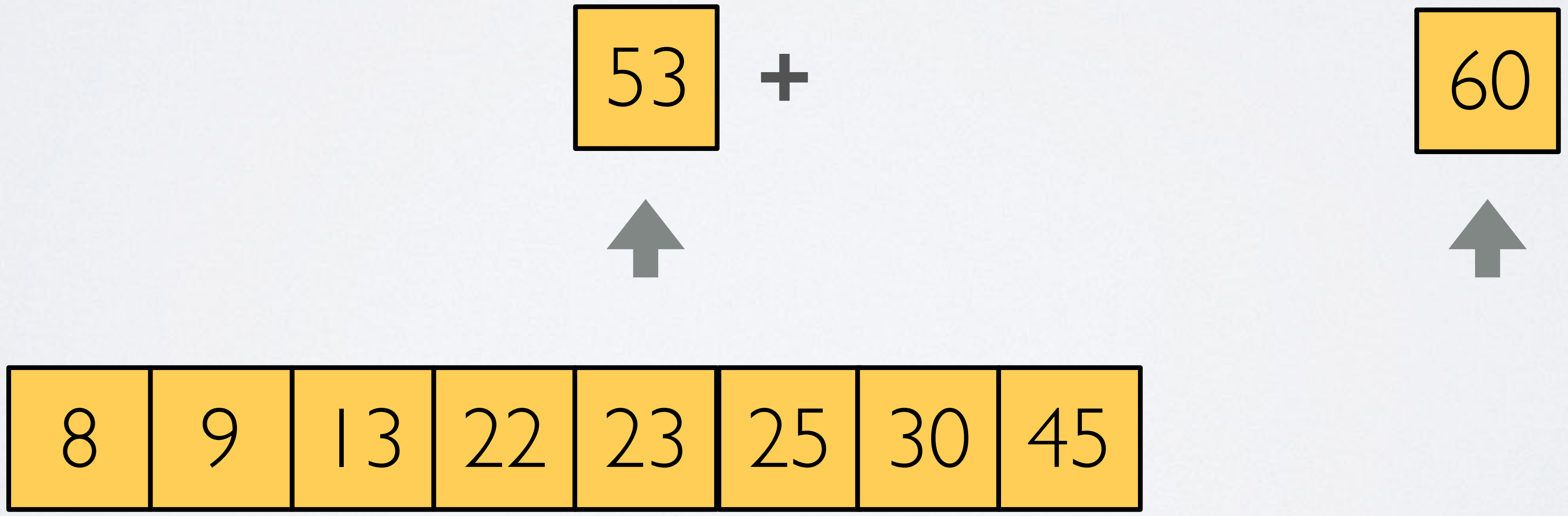
Better merge

- Go through both lists index by index
- Add smaller element to output list



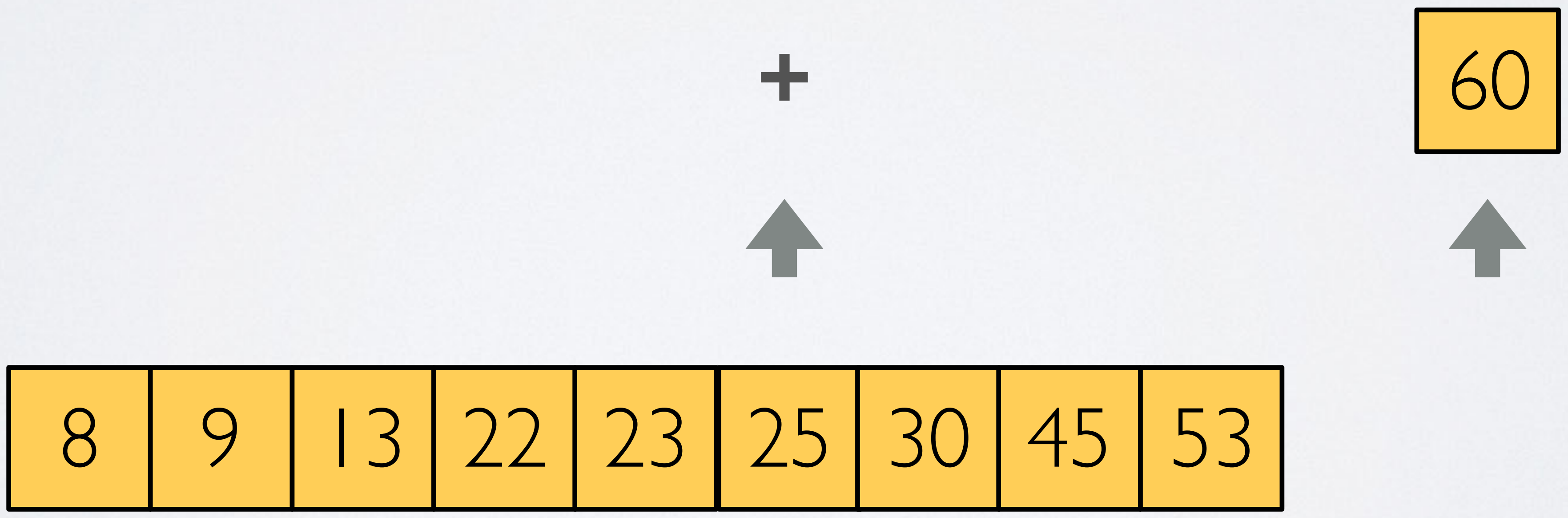
Better merge

- Go through both lists index by index
- Add smaller element to output list



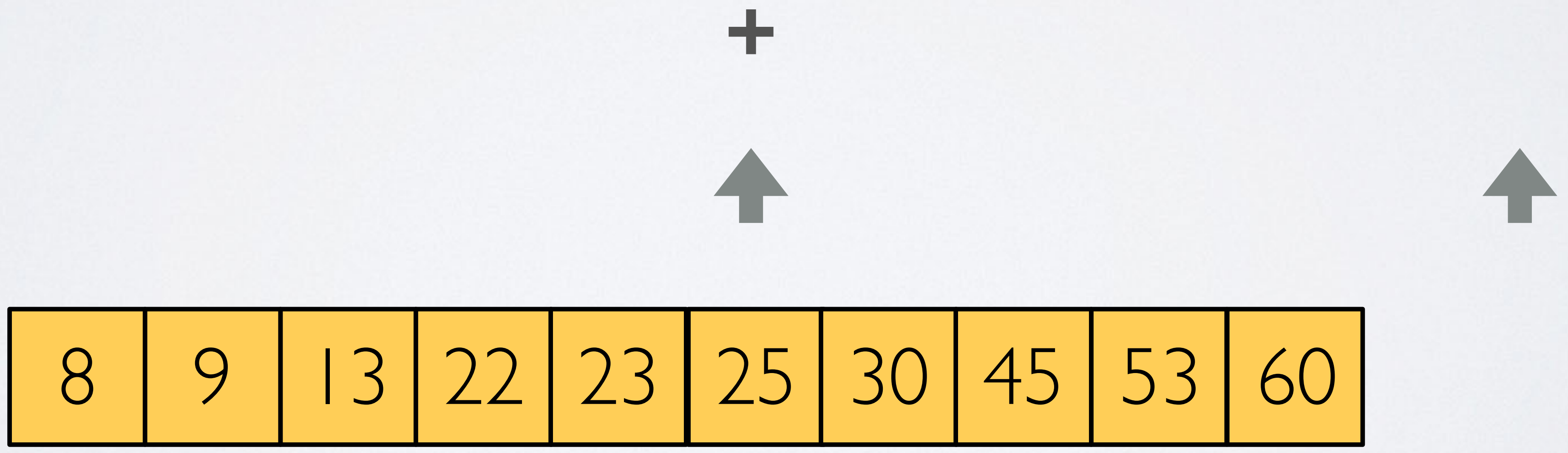
Better merge

- Go through both lists index by index
- Add smaller element to output list



Better merge

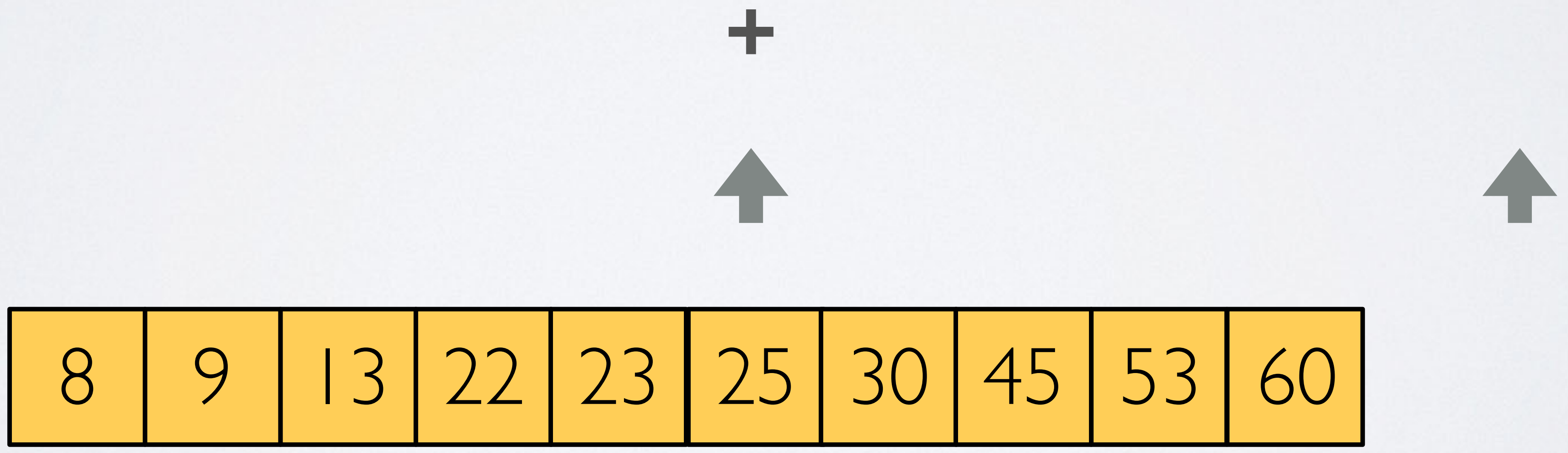
- Go through both lists index by index
- Add smaller element to output list



Better merge

- Go through both lists index by index
- Add smaller element to output list

Runtime if total # elements is n?

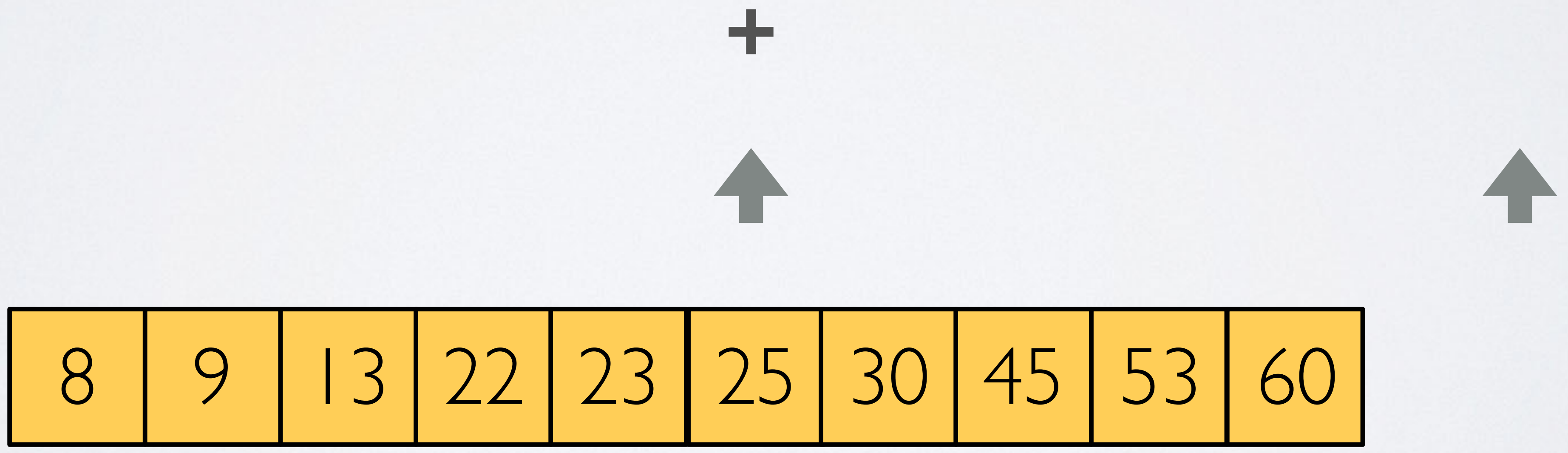


Better merge

- Go through both lists index by index
- Add smaller element to output list

Runtime if total # elements is n?

O(n)



Merge 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
```

Divide & Conquer

- Algorithmic design paradigm
  - divide: divide input `S` into disjoint subsets `S`$_1$`,…,S`$_k$
  - recur: solve sub-problems on `S`$_1$`,…,S`$_k$
  - conquer: combine solutions for `S`$_1$`,…,S`$_k$ into solution for `S`
- Base case is usually sub-problem of size `1` or `0`

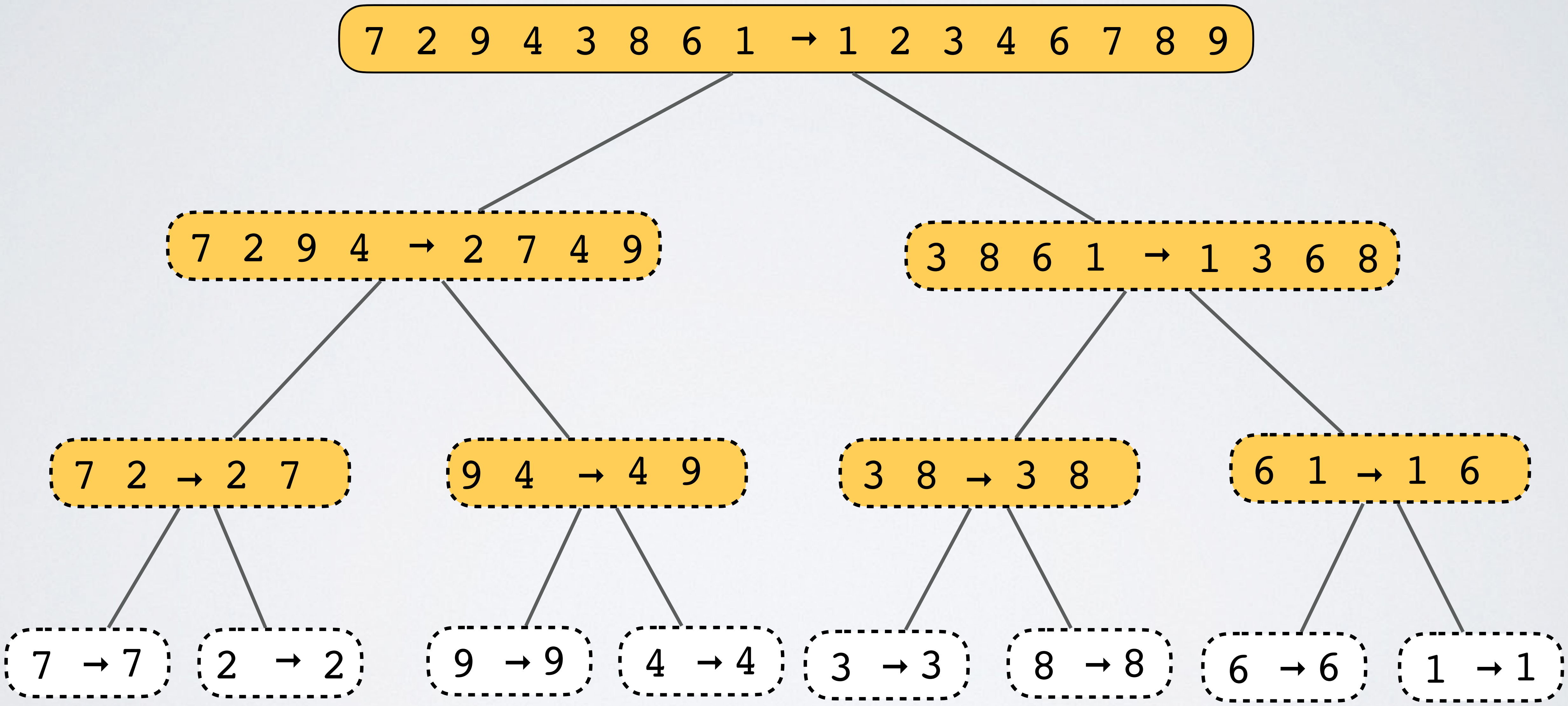
Merge Sort

- Sorting algorithm based on divide & conquer
- Like quadratic sorts
  - comparative
- Unlike quadratic sorts
  - recursive
  - not in-place (though in-place variants exist)
  - linearithmic `O(nlog n)`

Merge Sort

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

Merge Sort Recursion Tree



Merge Sort Pseudo-Code

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

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

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

$$
\begin{aligned}
T(1) &= c_1 \\
T(2) &= 2 \cdot T(1) + 2 = 2c_1 + 2 \\
T(4) &= 2 \cdot T(2) + 4 = 2(2c_1 + 2)4 = 4c_1 + 8 \\
T(8) &= 2 \cdot T(4) + 8 = 2(4c_1 + 8) + 8 = 8c_1 + 24 \\
T(16) &= 2 \cdot T(8) + 16 = 2(8c_1 + 24) + 16 = 16c_1 + 64
\end{aligned}
$$

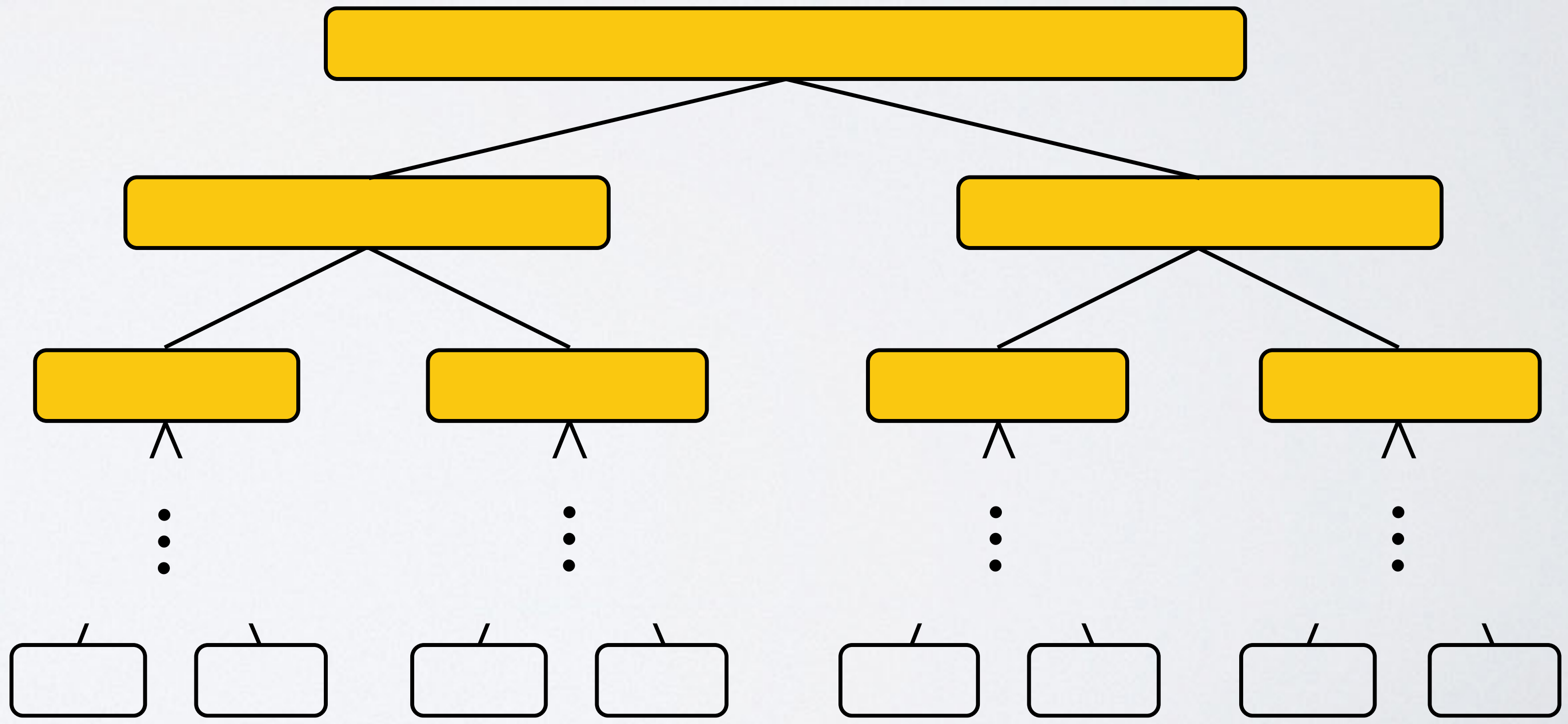
- Solution

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

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	sequences	size
0	2	n/2
1	4	n/4
2	8	n/4
⋮	⋮	⋮
k	2^{k+1}	$n/2^{k+1}$



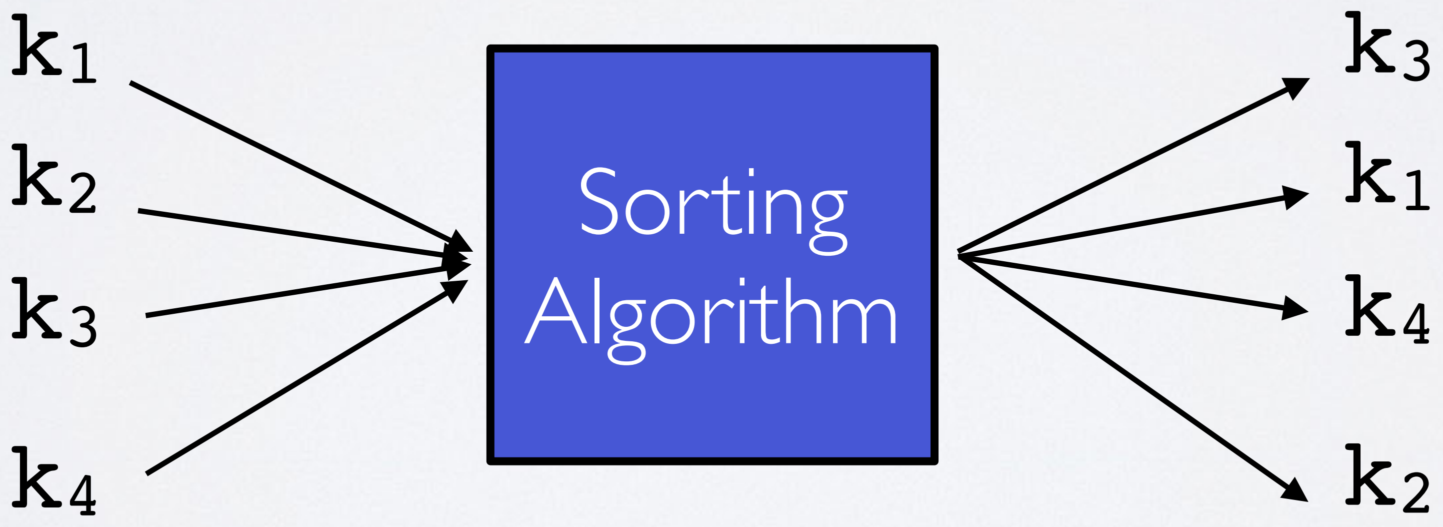
How Fast Can We Sort?

- Merge sort is `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

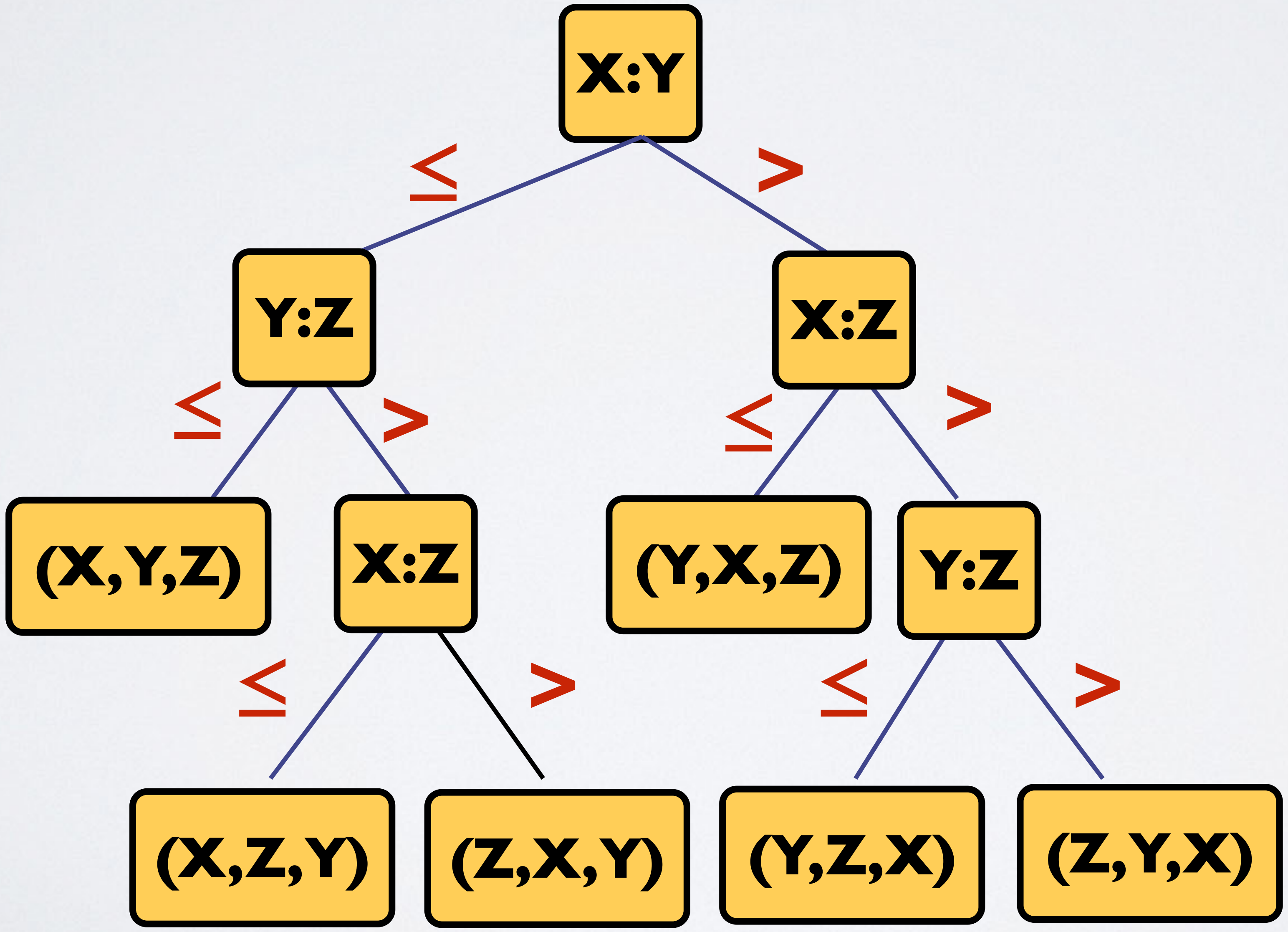


Lower Bound on Comparative Sorting

- The optimal (i.e., best possible) sorting algorithm can be modeled as
  - a *perfect binary decision tree* where
    - each internal node compares two keys
    - each leaf is a correct permutation of the input keys

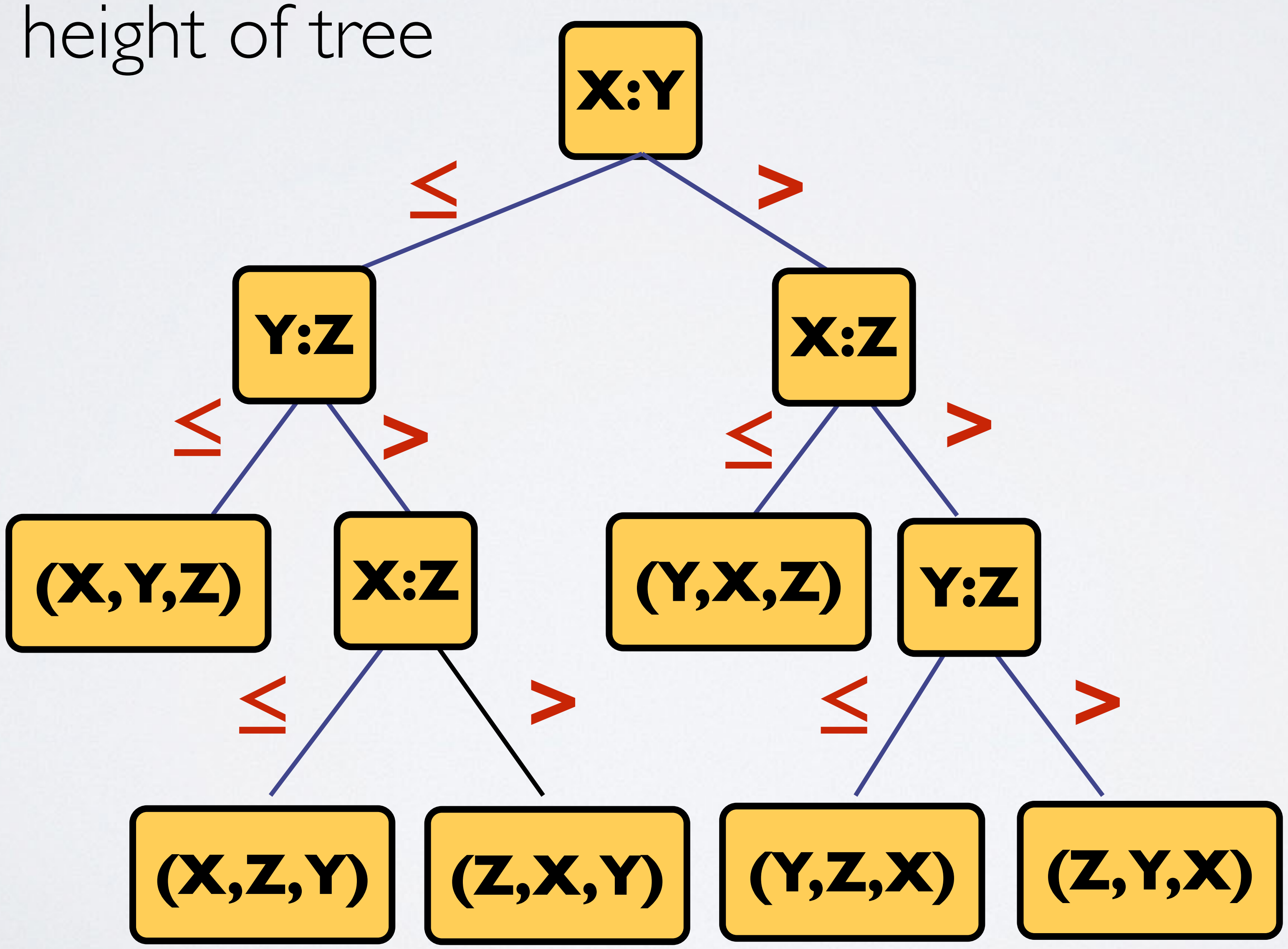
Lower Bound on Comparative Sorting

- Input is a sequence `X,Y,Z`
- All the possible sorted sequences are at the leaves



Lower Bound on Comparative Sorting

- To sort a sequence, we traverse tree
- What is the worst-case number of comparisons?
    - the height of tree



Lower Bound on Comparative Sorting

- What is the height of this binary (decision) tree?
- How many leaves does the tree have?
  - each leaf corresponds to a permutation of the input keys…
  - …and since are `n!` possible permutations of `n` keys, there are `n!` leaves
- A *perfect binary tree* with `L` leaves has height `log L`
  - So a 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 the height of the tree and worst-case number of comparisons is
  - `Ω(n log n)`

Non-Comparative Sorting

- Example: alphabetizing Doug’s record collection
  - Don't care about sorting within a letter
  - Dick Gaughan and Rhiannon Giddens can be in any order
- How could we do this in O(n) time?

Non-Comparative Sorting

- Have a pile for artist names beginning with A, B, C, …
  - Within a letter, albums are unsorted
- Dick Gaughan album: immediately go to “G” section
- Run the Jewels album: immediately go to “R” section
- Rhiannon Giddens album: immediately go to “G” section
- This is called “Radix sort”
  - More details in optional slides + readings

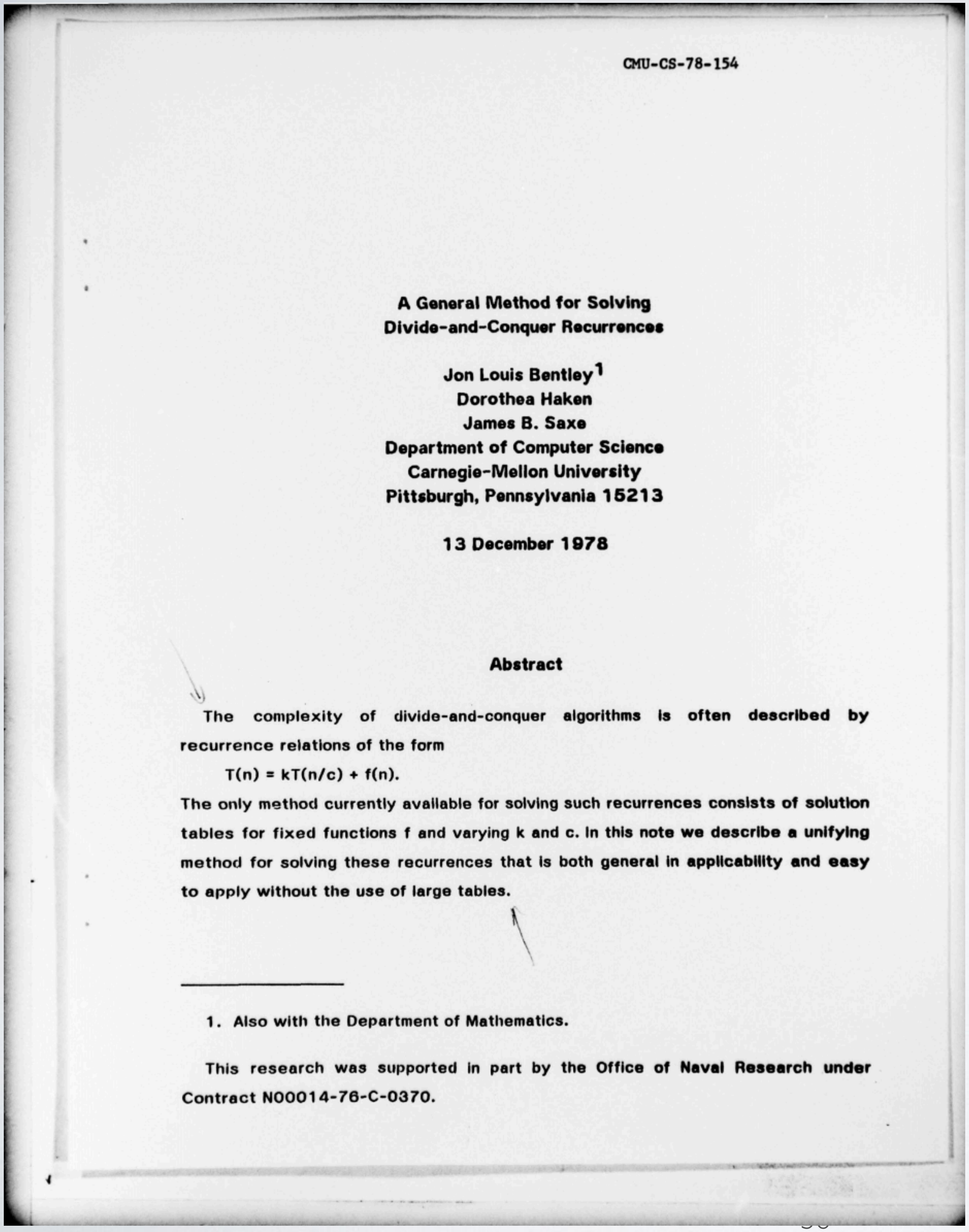
Sorting in practice

- Python and Java use TimSort, a combination of merge and insertion sort
  - Intuition: insertion sort great for sorted data, many real-world datasets are already "partially" sorted
- Quicksort, a different divide-and-conquer algorithm, also commonly used
  - More details on quicksort in optional slides + readings
- Sorting lots of data stored on many disks is still an area of active research!

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

Optional material



CMU-CS-78-154

A General Method for Solving
Divide-and-Conquer Recurrences

Jon Louis Bentley[1]
Dorothea Haken
James B. Saxe
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

13 December 1978

Abstract

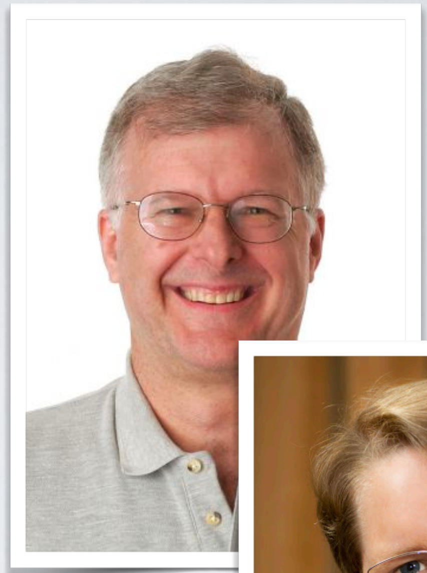
The complexity of divide-and-conquer algorithms is often described by recurrence relations of the form

$T(n) = kT(n/c) + f(n)$.

The only method currently available for solving such recurrences consists of solution tables for fixed functions f and varying k and c. In this note we describe a unifying method for solving these recurrences that is both general in applicability and easy to apply without the use of large tables.

1. Also with the Department of Mathematics.

This research was supported in part by the Office of Naval Research under Contract N00014-76-C-0370.



The Master Theorem

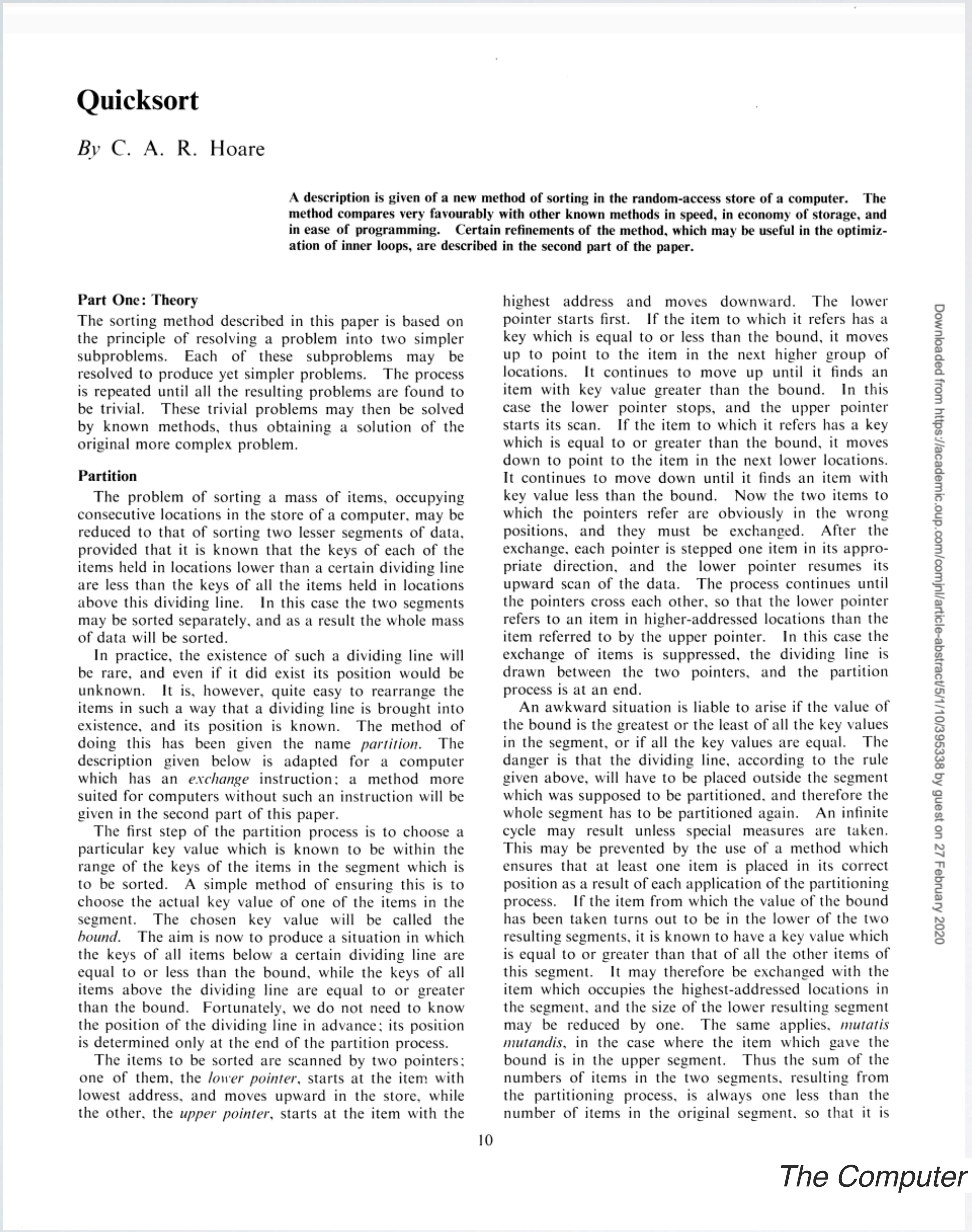
- Solves large class of recurrence relations
  - we will learn how to use it but not its proof
  - See Dasgupta et al. p. 58-60 for proof
- Let **`T(n)`** be a monotonically-increasing function of form

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

  - **`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 $T(n) = \Theta(n^d \log n)$
    $= \Theta(n^1 \log n)$
    $= \Theta(n \log n)$



Quicksort

By C. A. R. Hoare

A description is given of a new method of sorting in the random-access store of a computer. The method compares very favourably with other known methods in speed, in economy of storage, and in ease of programming. Certain refinements of the method, which may be useful in the optimization of inner loops, are described in the second part of the paper.

Part One: Theory

The sorting method described in this paper is based on the principle of resolving a problem into two simpler subproblems. Each of these subproblems may be resolved to produce yet simpler problems. The process is repeated until all the resulting problems are found to be trivial. These trivial problems may then be solved by known methods, thus obtaining a solution of the original more complex problem.

Partition

The problem of sorting a mass of items, occupying consecutive locations in the store of a computer, may be reduced to that of sorting two lesser segments of data, provided that it is known that the keys of each of the items held in locations lower than a certain dividing line are less than the keys of all the items held in locations above this dividing line. In this case the two segments may be sorted separately, and as a result the whole mass of data will be sorted.

In practice, the existence of such a dividing line will be rare, and even if it did exist its position would be unknown. It is, however, quite easy to rearrange the items in such a way that a dividing line is brought into existence, and its position is known. The method of doing this has been given the name *partition*. The description given below is adapted for a computer which has an *exchange* instruction; a method more suited for computers without such an instruction will be given in the second part of this paper.

The first step of the partition process is to choose a particular key value which is known to be within the range of the keys of the items in the segment which is to be sorted. A simple method of ensuring this is to choose the actual key value of one of the items in the segment. The chosen key value will be called the *bound*. The aim is now to produce a situation in which the keys of all items below a certain dividing line are equal to or less than the bound, while the keys of all items above the dividing line are equal to or greater than the bound. Fortunately, we do not need to know the position of the dividing line in advance; its position is determined only at the end of the partition process.

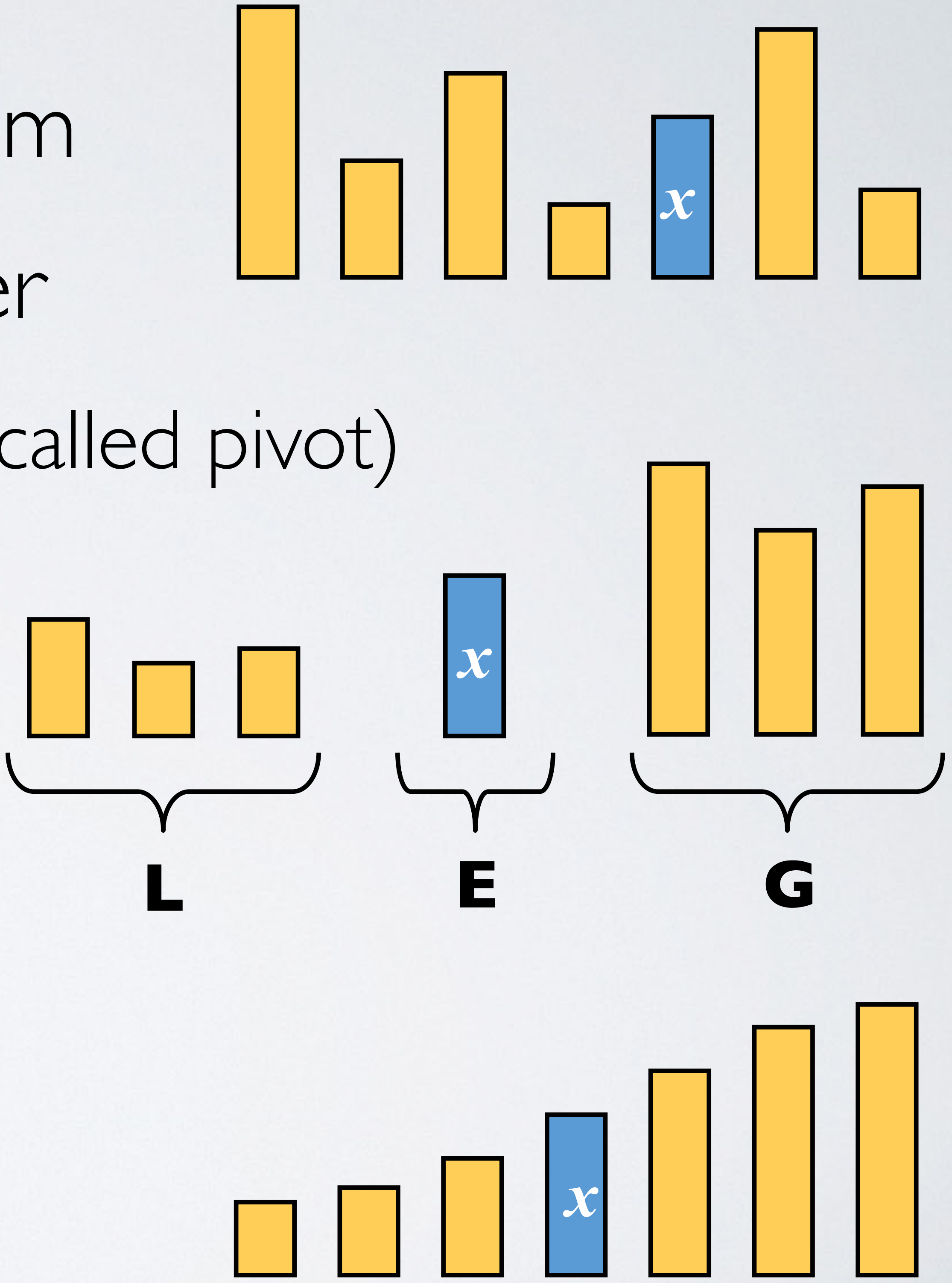
The items to be sorted are scanned by two pointers; one of them, the *lower pointer*, starts at the item with lowest address, and moves upward in the store, while the other, the *upper pointer*, starts at the item with the highest address and moves downward. The lower pointer starts first. If the item to which it refers has a key which is equal to or less than the bound, it moves up to point to the item in the next higher group of locations. It continues to move up until it finds an item with key value greater than the bound. In this case the lower pointer stops, and the upper pointer starts its scan. If the item to which it refers has a key which is equal to or greater than the bound, it moves down to point to the item in the next lower locations. It continues to move down until it finds an item with key value less than the bound. Now the two items to which the pointers refer are obviously in the wrong positions, and they must be exchanged. After the exchange, each pointer is stepped one item in its appropriate direction, and the lower pointer resumes its upward scan of the data. The process continues until the pointers cross each other, so that the lower pointer refers to an item in higher-addressed locations than the item referred to by the upper pointer. In this case the exchange of items is suppressed, the dividing line is drawn between the two pointers, and the partition process is at an end.

An awkward situation is liable to arise if the value of the bound is the greatest or the least of all the key values in the segment, or if all the key values are equal. The danger is that the dividing line, according to the rule given above, will have to be placed outside the segment which was supposed to be partitioned, and therefore the whole segment has to be partitioned again. An infinite cycle may result unless special measures are taken. This may be prevented by the use of a method which ensures that at least one item is placed in its correct position as a result of each application of the partitioning process. If the item from which the value of the bound has been taken turns out to be in the lower of the two resulting segments, it is known to have a key value which is equal to or greater than that of all the other items of this segment. It may therefore be exchanged with the item which occupies the highest-addressed locations in the segment, and the size of the lower resulting segment may be reduced by one. The same applies, *mutatis mutandis*, in the case where the item which gave the bound is in the upper segment. Thus the sum of the numbers of items in the two segments, resulting from the partitioning process, is always one less than the number of items in the original segment, so that it is

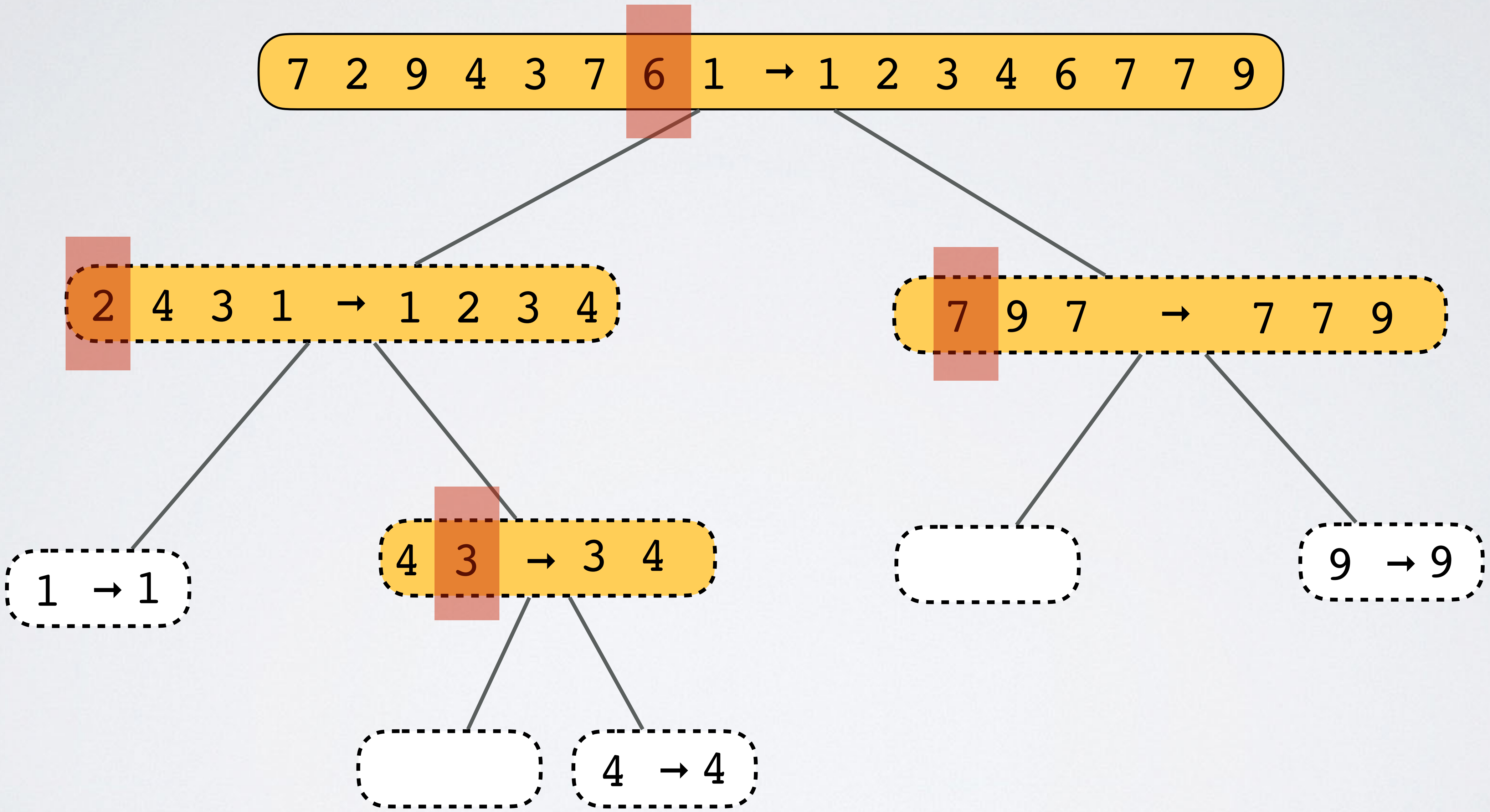
The Computer Journal, Volume 5, Issue 1, 1962, Pages 10–16,

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 Recursion Tree



Quicksort Pseudo-Code

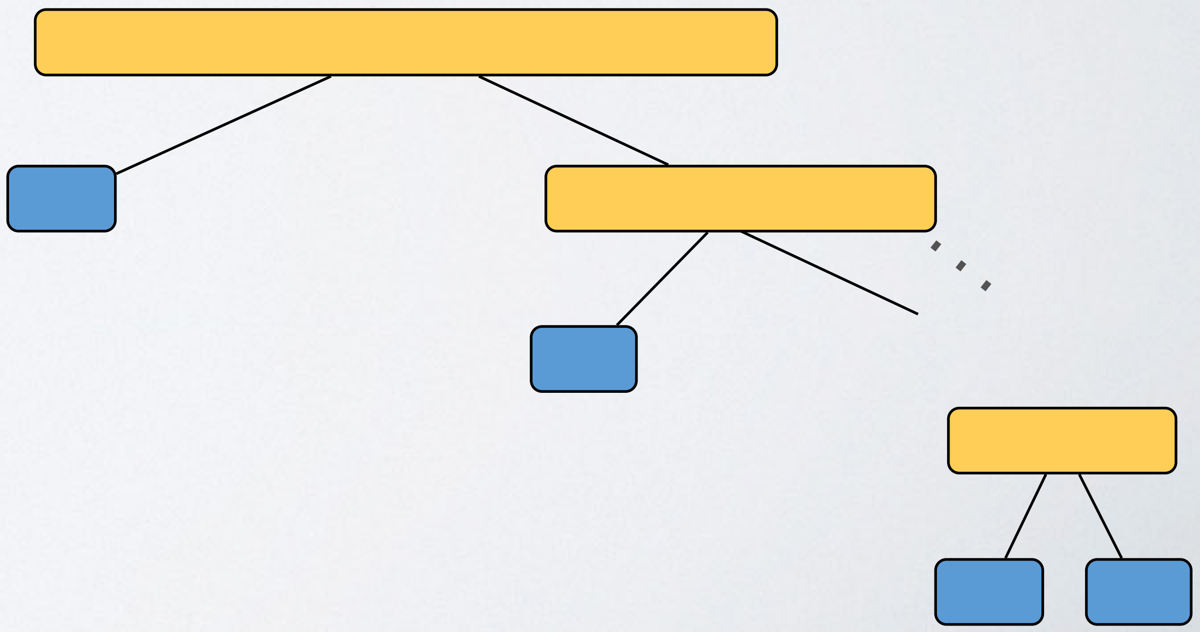
```
function quick_sort(A):
  if A.length ≤ 1
    return A

  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)
```

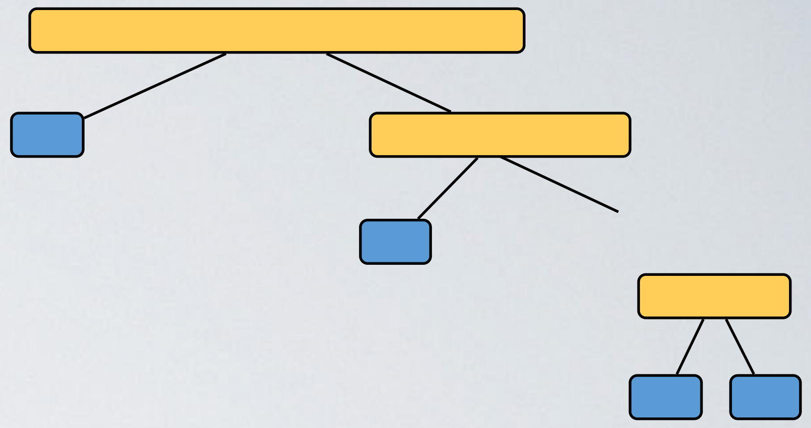
Worst-Case Running Time

- Worst-case for Quicksort
  - when pivot is (unique) min or max element
  - Either `L` or `G` has size `n-1` and the 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



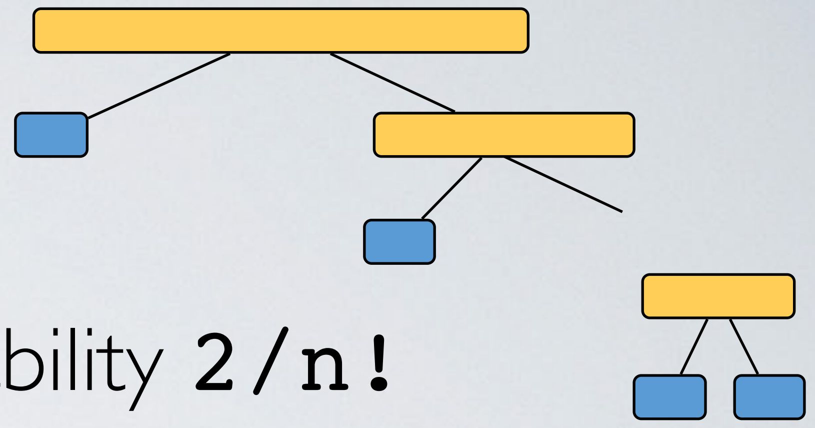
Worst-Case Running Time



- Worst-case runtime of Quicksort is `O(n²)`
  - but this only happens if we *always* pick the min (or max) element as a pivot
- How likely is that?
  - each time we pick a pivot, we have probability `1/n` of picking the min/max element
  - worst case happens if we keep picking min/max element at every level of the recursion (remember `n` gets smaller at each level)
  - since each pivot is chosen independently, the probability of *always* picking the min/max element is

$$\prod_{i=0}^{n-2} \frac{1}{n-i} = \frac{1}{n!}$$

Worst-Case Running Time



- Worst case is `O(n²)` but happens with probability `2/n!`
  - for array of size `n=100` this is `2/(9.332×10¹⁵⁷)`
  - So worst case is *very* unlikely
- So what is the *expected* runtime of Quicksort?
- The expected runtime of a randomized algorithm is

$$\sum_{\text{all rand choices c}} \Pr[\text{Alg makes choice c}] \cdot \text{Time(Alg with choice c)}$$

  - $\approx$ if we ran algo. a billion times & took the average runtime

Expected Runtime of Quicksort

- Assume there are no duplicates (if there are then are even less recursive calls)
- At each level of recursion, Quicksort can make **`n`** different & unique recursive calls depending on the choice of pivot
  - Each choice of pivot produces a "split" (i.e., a partition of sequence into **`L`** & **`G`** sets)
- The set of all possible splits are
  - split #1: `|L|=0` and `|G|=n-1` has cost `T(0)+T(n-1)`
  - split #2: `|L|=1` and `|G|=n-2` has cost `T(1)+T(n-2)`
  - …
  - split #n: `|L|=n-1` and `|G|=0` has cost `T(n-1)+T(0)`
- 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

$$\begin{aligned}\mathbb{E}[T(n)] &= n + \overset{\substack{\textbf{prob of first split}\\ \downarrow}}{\frac{1}{n}} \cdot \overset{\substack{\textbf{cost of first split}\\ \downarrow}}{\Big(T(0) + T(n-1)\Big)} + \cdots + \overset{\substack{\textbf{prob of last split}\\ \downarrow}}{\frac{1}{n}} \cdot \overset{\substack{\textbf{cost of last split}\\ \downarrow}}{\Big(T(n-1) + T(n-1-(n-1))\Big)} \\ &= n + \frac{1}{n} \cdot \sum_{i=0}^{n-1} \Big(T(i) + T(n-1-i)\Big)\end{aligned}$$

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

Don't need to know the proof of this.

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

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
  line = low
  for i from low to high – 1:
    if A[i] <= pivotValue :
      swap A[line] and A[i]
        line++
  swap A[line] and A[high]   # move the pivot back
  return line
```

Merge Sort vs. Quicksort

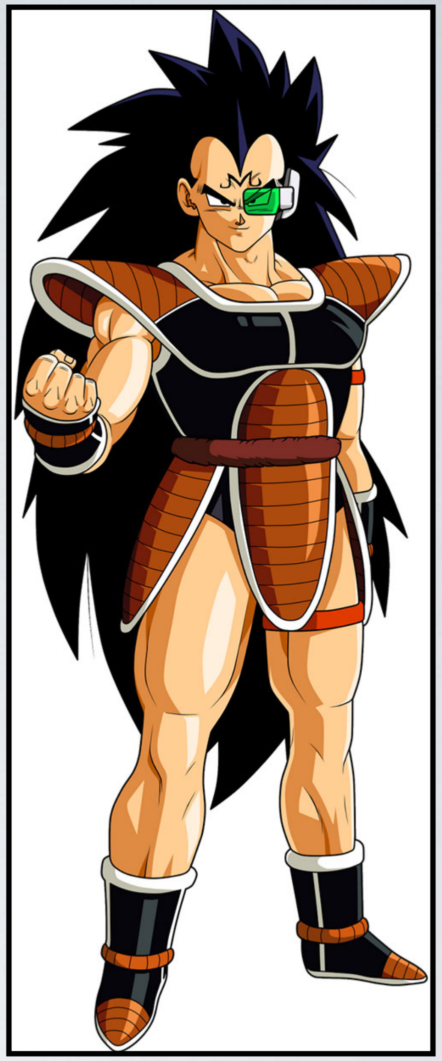
- Merge sort is worst-case `O(n log n)`
- Quicksort is expected `O(n log n)`
- Which is better?
- In practice quicksort is faster!
    - it also uses less space
    - constants are better

Non-Comparative Sorting

- Comparison-based sorting algorithms can be used on different types of inputs as long as we can *compare* them
    - Integers, floats, strings, arrays, other objects…
- 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`?
  - compare digit by digit
  - the 3 high order digits are same…
  - …so you keep going until you see that
  - … `3<4` so `258391` must be 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 an array of `10` buckets
  - for each number **`x`**, add it to the bucket at index **`x`**

0	1	2	3	4	5	6	7	8	9
	1	2	3		5	6			
	1								

  - Return concatenation of all buckets (in order)
    - print out `[1,1]+[2]+[3]+[5]+[6]`
- What is the runtime?
  - **`O(n)`**

This only works for single-digit numbers!

Radix Sort

- Radix sort combines both techniques so that it can work over multi-digit numbers
  - iterate from least significant to most significant digit
  - and use buckets to sort number by current 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 `[8271,0273,0279,8735,8736,7891]`
- 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 from least to most significant
    for each number in A
      digit = digit in number at place
      buckets[digit].append(number)
    A = concatenate all buckets in order
    empty all buckets
  return A
```

- Very efficient!
  - `O(nd)`, where `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 then radix sort with two buckets but
    - number of digits will dominate runtime
    - for long sequences will be very slow

Summary of Sorting Algorithms

Algorithm	Time	Notes
Selection sort	`O(n²)`	in-place slow (good for small inputs)
Insertion sort	`O(n²)`	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