Lecture 15
Sorting

Department of Computer Science
Hofstra University

Acknowledgement: Lecture slides based on UofW Course on Data Structures

Warm Up

If I handed you a stack of papers and asked you to sort them by author name alphabetically, how would you do it?

Selection Sort – Flip through the stack from front to back looking for the first name, then pull it to the front. Then I would flip through again looking for the second name and put it behind the first and so on until all were sorted

Insertion Sort – Look at the first two papers and put them in sorted order, then look at the third and put it in sorted order with the previous two and continue until the whole stack is in sorted order

Merge Sort – Spread the papers out on the ground and break them into subsections, sort the subsections section by section then put them all back together

Bucket Sort – Put the papers into groups based on the first letter of the author's name until I had 26 piles, then sort within those piles and put them all back together

Intro to Sorting

Types of Sorts

Comparison Sort

Compare two elements at a time

General sort, works for most types of elements

What does this mean?
compareTo() works for your elements

- And for our running times to be correct, compareTo() must run in O(1) time

Specialized Sorts ("Niche Sorts")

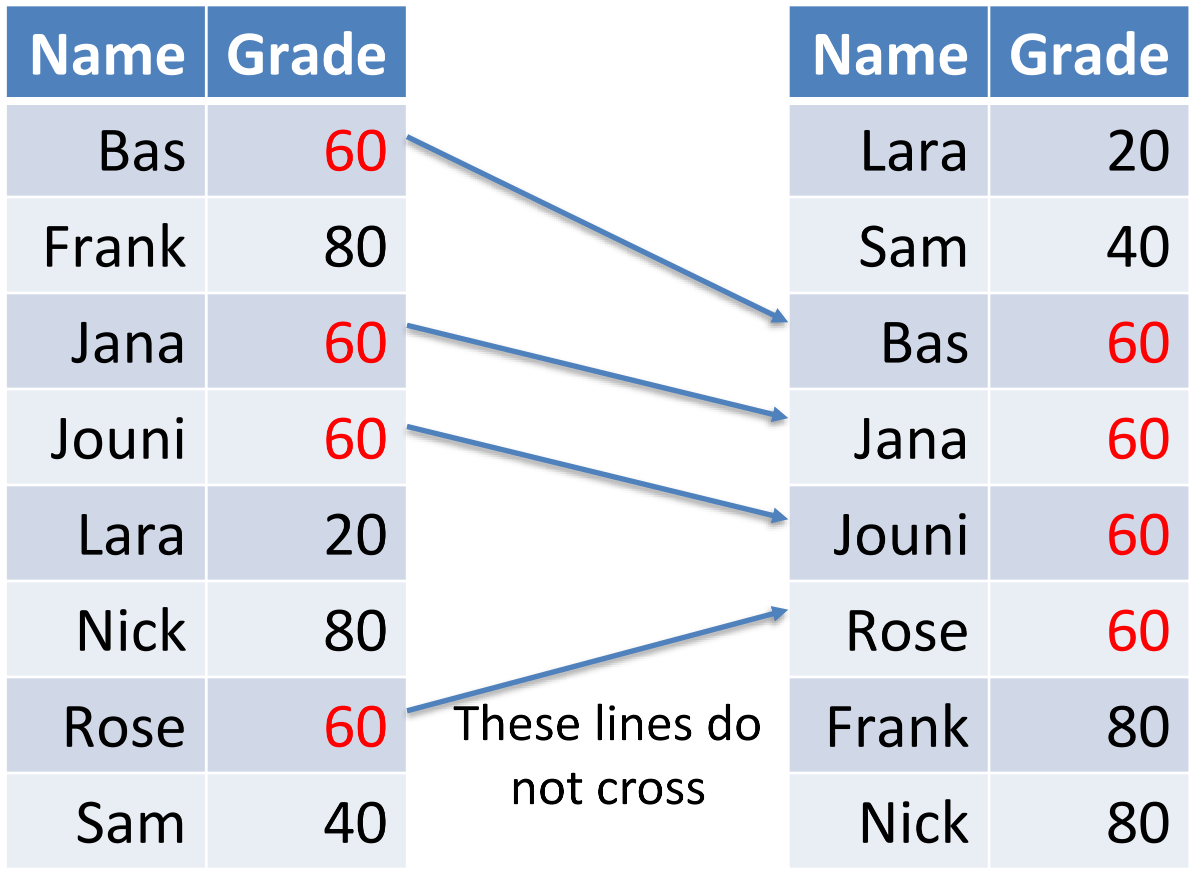
Leverages specific properties about the items in the list to achieve faster runtimes

Typically runs in O(*n*) time

e.g., sorting integers or strings where we sort by each individual digit or character

Stable Sort, In-Place Sort

- A stable sorting algorithm is one that maintains the relative order of elements with equal keys in the sorted output as they appeared in the input
  - e.g., Insertion Sort, Merge Sort, Radix Sort
- Stability is important when multiple sorting operations are performed on data with multiple keys. For example, if you first sort a list of students by name and then by grade, a stable sort will ensure that students with the same grade remain sorted by name. This characteristic is crucial in scenarios where secondary attributes need to be preserved after sorting by primary attributes.
- In-place sort: A sorting algorithm is in-place if it modifies input array and does not allocate extra memory. Useful for minimizing memory usage



Name	Grade
Bas	60
Frank	80
Jana	60
Jouni	60
Lara	20
Nick	80
Rose	60
Sam	40

Name	Grade
Lara	20
Sam	40
Bas	60
Jana	60
Jouni	60
Rose	60
Frank	80
Nick	80

Sorting by the Grade attribute (the key) maintains the relative order of the Name attribute for persons with equal Grade

Principle 1: Iterative Improvement

Invariants/Iterative improvement

- Step-by-step, make one more part of the input your desired output

We’ll write iterative algorithms to satisfy the following invariant:

After *k* iterations of the loop, the first *k* elements of the array will be sorted.

Insertion Sort

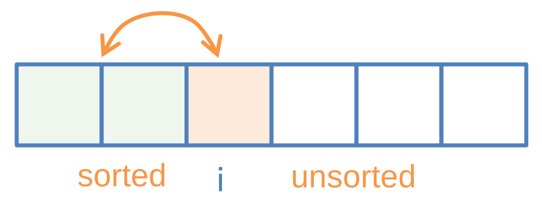
Insertion Sort

- Insertion sort works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.
  - Start with second element of the array as first element in the array is assumed to be sorted.
  - Compare second element with the first element and check if the second element is smaller then swap them.
  - Move to the third element and compare it with the first two elements and put at its correct position
  - Repeat until the entire array is sorted.
- Time complexity: O(n^2), as there are two nested loops:
  - Outer loop to select each element in the unsorted group one by one, with O(n) complexity
  - Inner loop to insert that element into the sorted group, with O(n) complexity
- Insertion Sort | GeeksforGeeks
  - https://www.youtube.com/watch?v=OGzPmgsI-pQ

Insertion Sort: Basic Algorithm

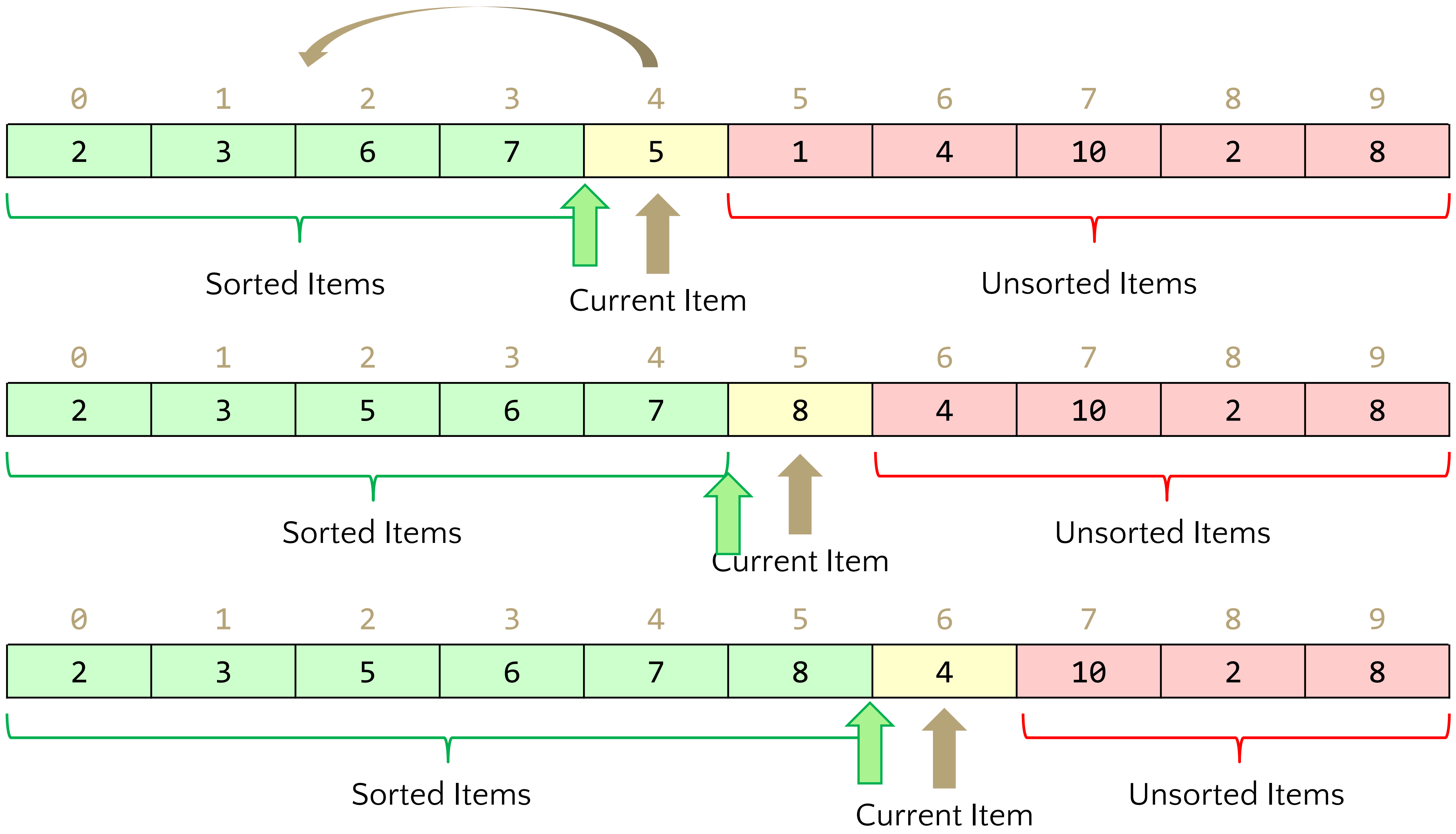
For each **position i** from **1** to **length-1**

Swap successive pairs to put value in **position i** in correct location relative to earlier values

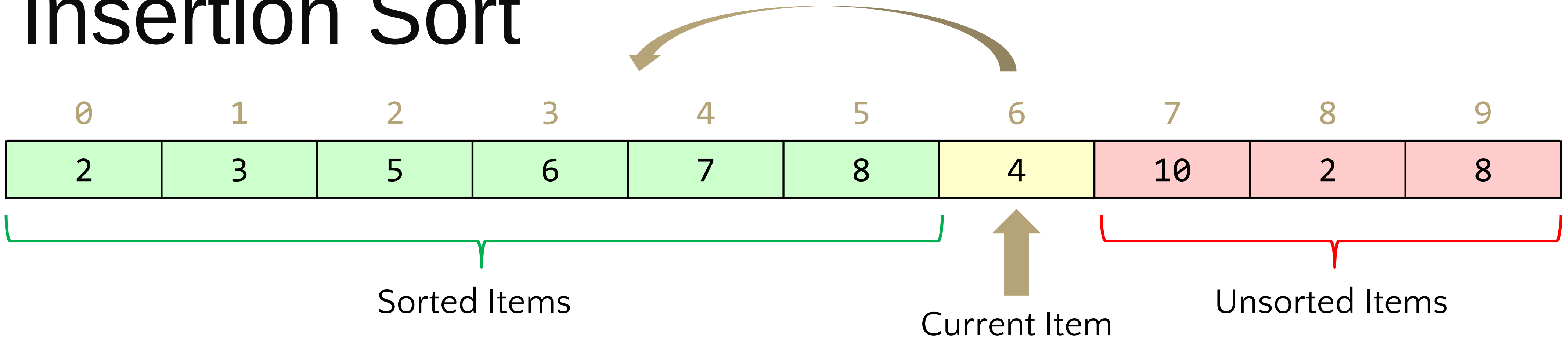


1	8	4	3	7	2	pos 1
1	4	8	3	7	2	pos 2
1	3	4	8	7	2	pos 3
1	3	4	7	8	2	pos 4
1	2	3	4	7	8	pos 4

Insertion Sort



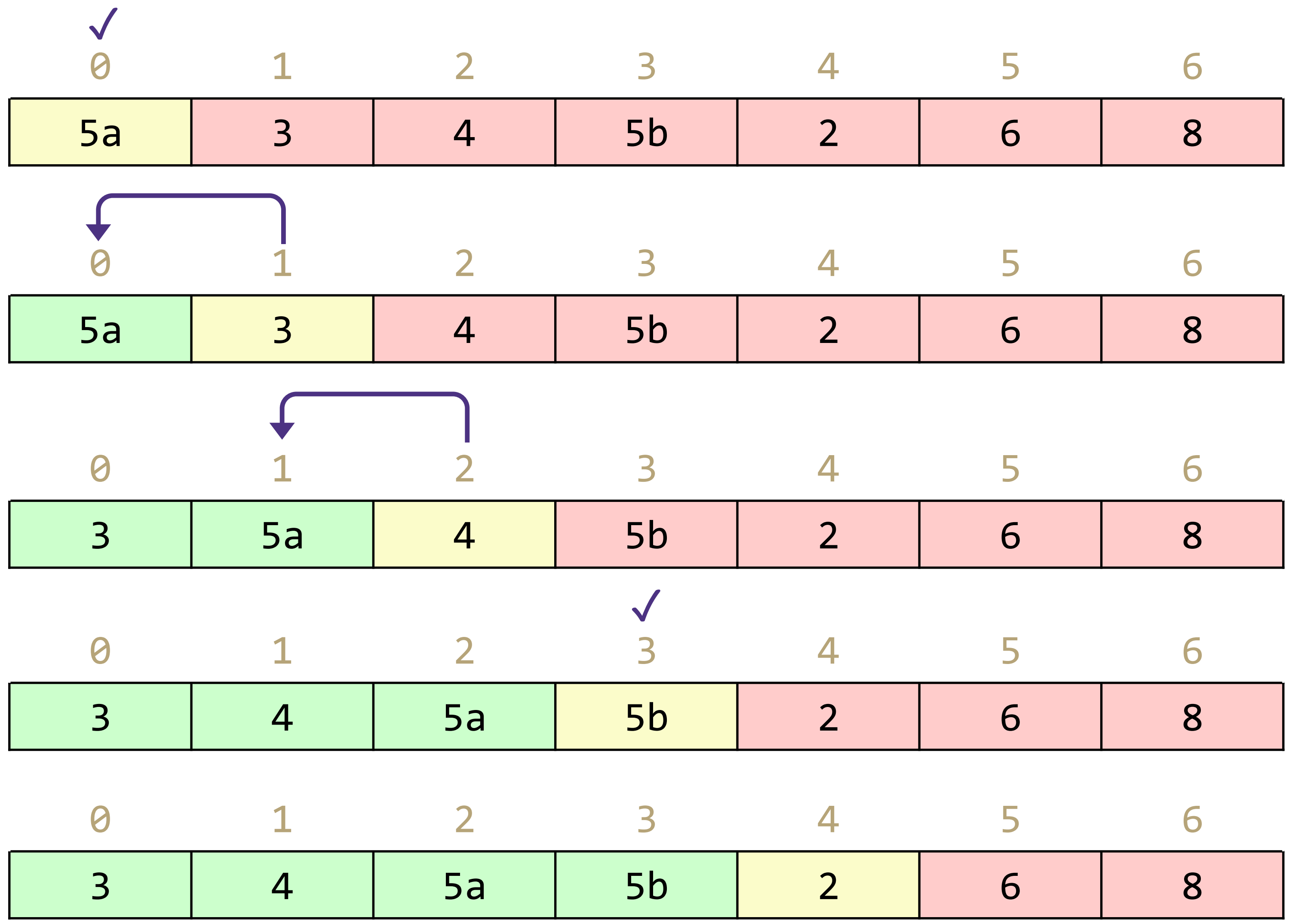
Insertion Sort



```
public void insertionSort(collection) {
   for (entire list)
      if(currentItem is smaller than largestSorted)
         int newIndex = findSpot(currentItem);
         shift(newIndex, currentItem);
}
public int findSpot(currentItem) {
   for (sorted list going backwards)
      if (spot found) return
}
public void shift(newIndex, currentItem) {
   for (i = currentItem > newIndex)
      item[i+1] = item[i]
   item[newIndex] = currentItem
}
```

Worst case runtime?	$\Theta(n^2)$
Best case runtime?	$\Theta(n)$
Average runtime?	$\Theta(n^2)$
Stable?	Yes
In-place?	Yes
Useful for:	Mostly sorted collections of primitives

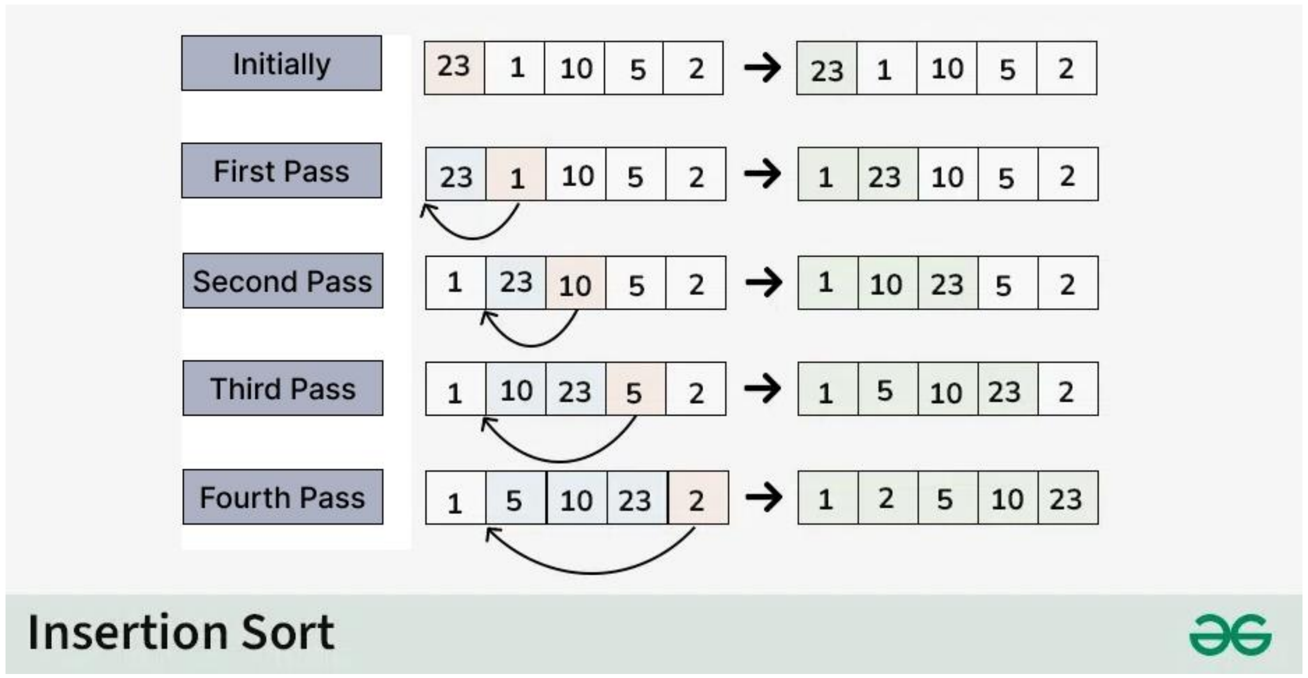
Insertion Sort Stability



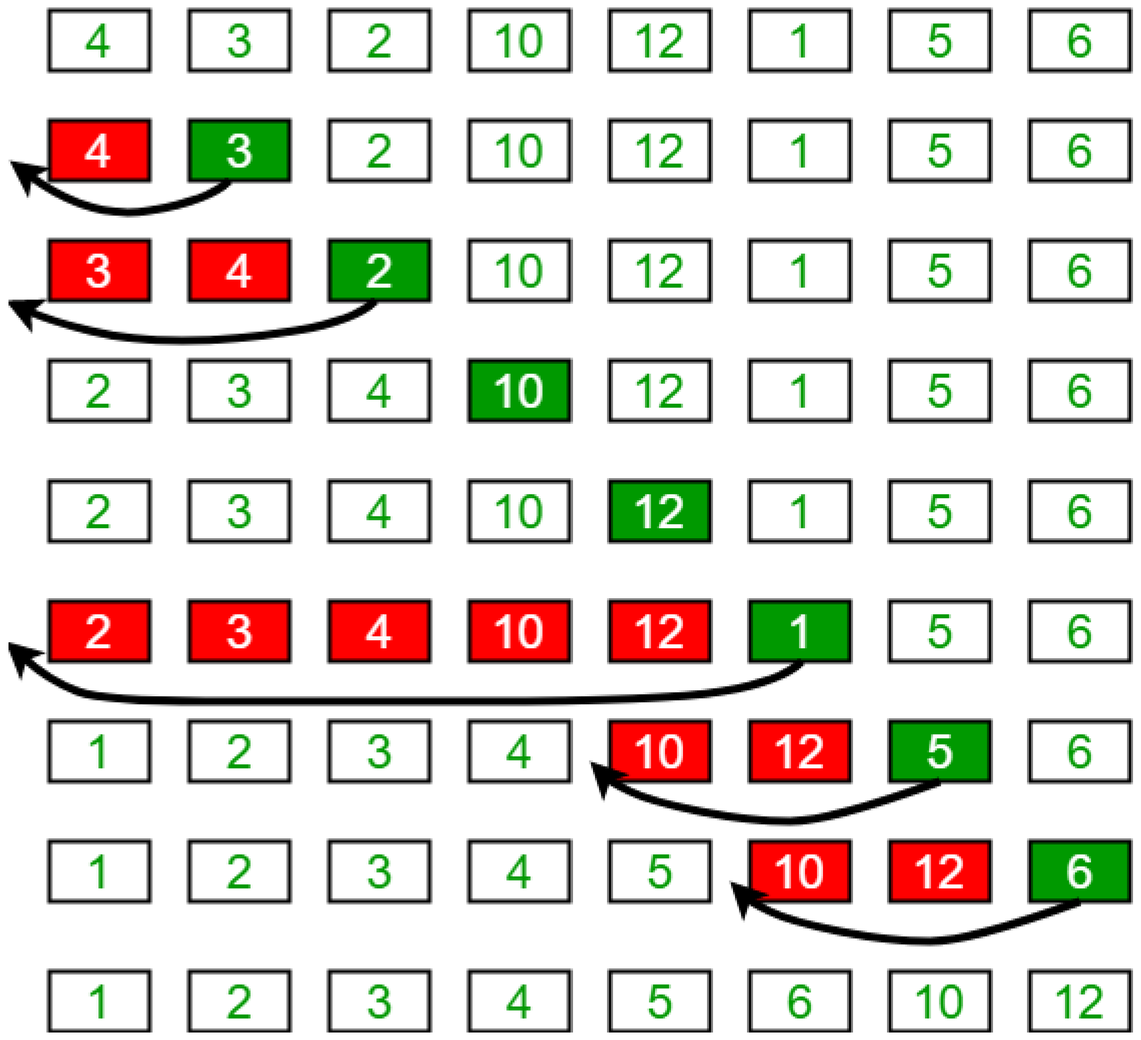
Insertion sort is stable

- All swaps happen between adjacent items to get current item into correct relative position within sorted portion of array

Insertion Sort: More Examples



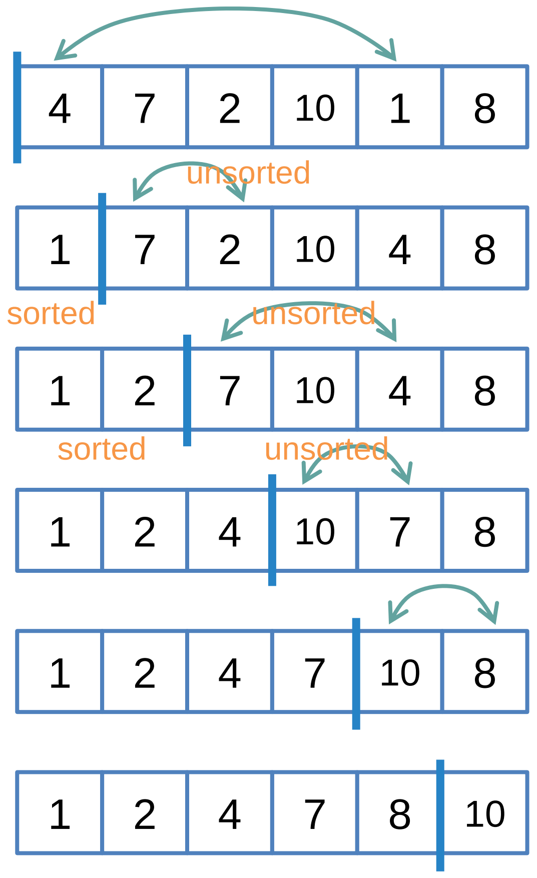
Insertion Sort Execution Example



Selection Sort

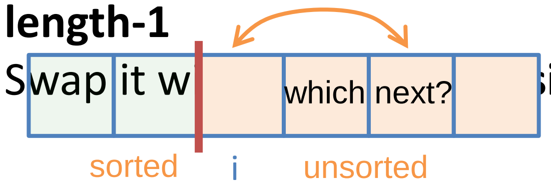
Selection Sort

- Selection Sort works by repeatedly selecting the smallest element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted
    - First find the smallest element and swap it with the first element. This way we get the smallest element at its correct position
    - Then find the smallest among remaining elements (or second smallest) and move it to its correct position by swapping
    - After k iterations of the loop, the k smallest elements of the array are (sorted) in indices 0, … , k-1
    - Keep going until all elements are sorted.
- Time complexity: $O(n^2)$, as there are two nested loops:
    - Outer loop to select each element one by one with O(n) complexity
    - Inner loop to compare that element with every other element with O(n) complexity
- Selection Sort | GeeksforGeeks
    - https://www.youtube.com/watch?v=xWBP4lzkoyM

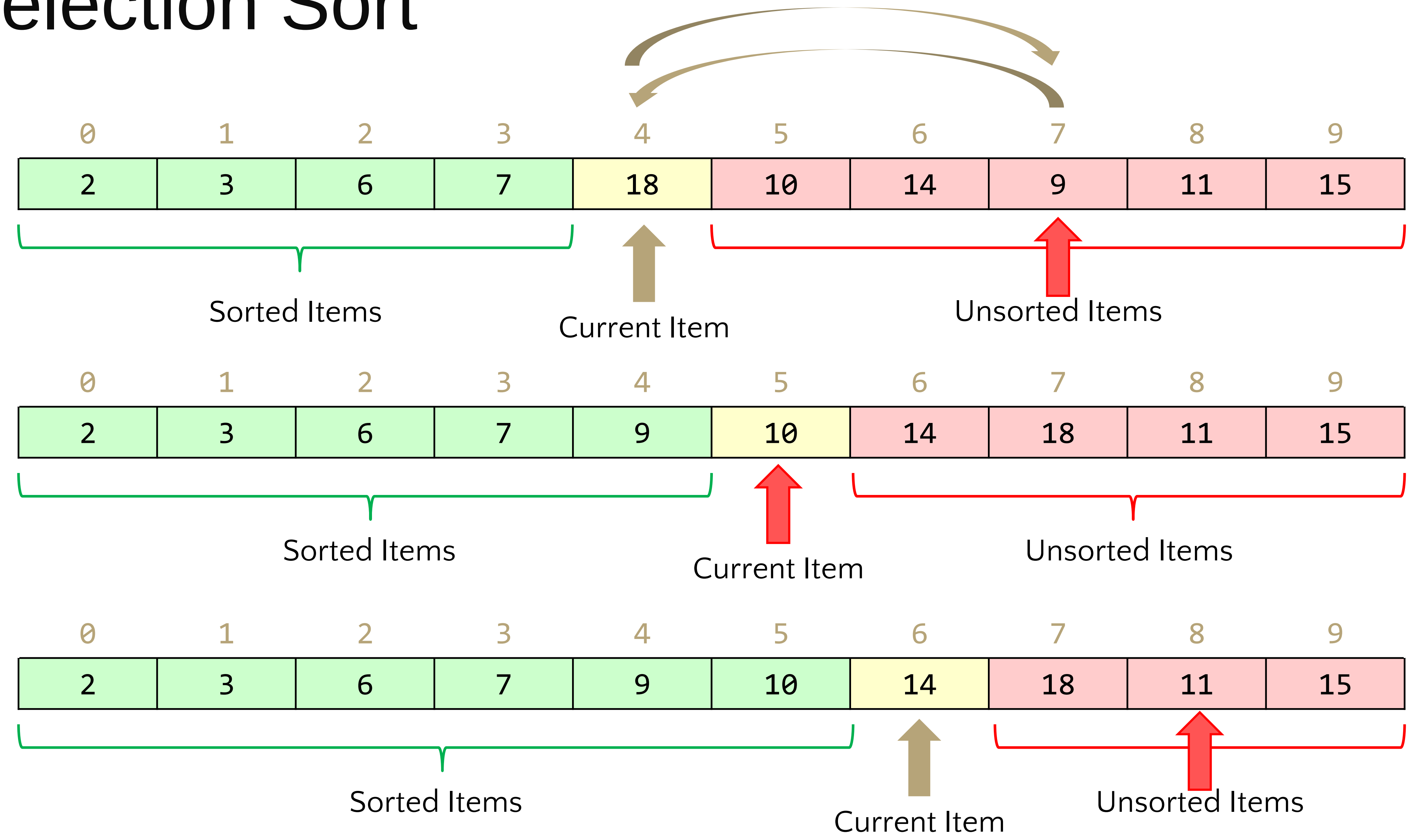


<u>Selection Sort: Basic Algorithm</u>

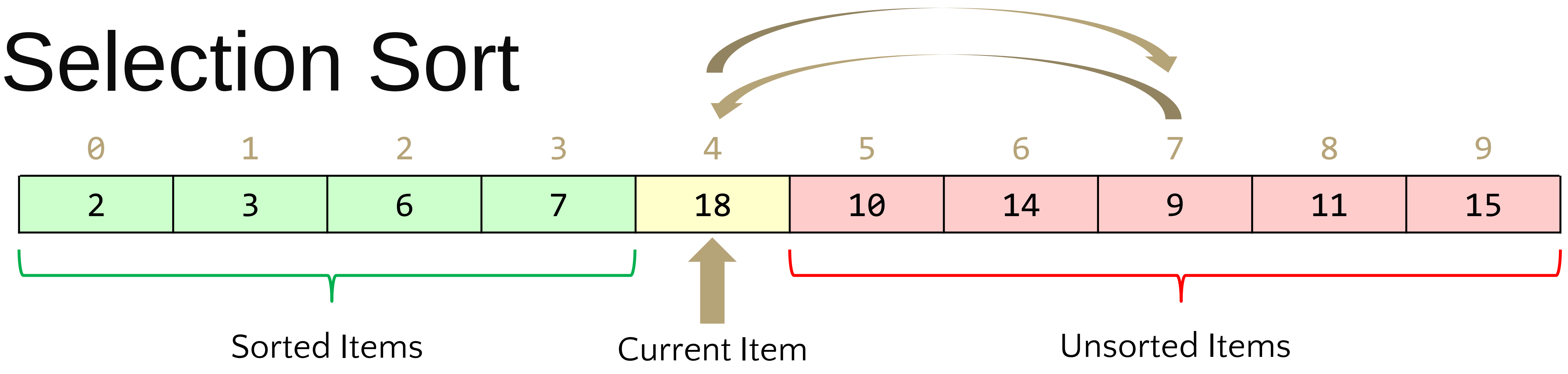
```
For each **position i** from **0** to **length-2**
    Find smallest element in **positions i** to **length-1**
    Swap it w                     ition i
```



Selection Sort



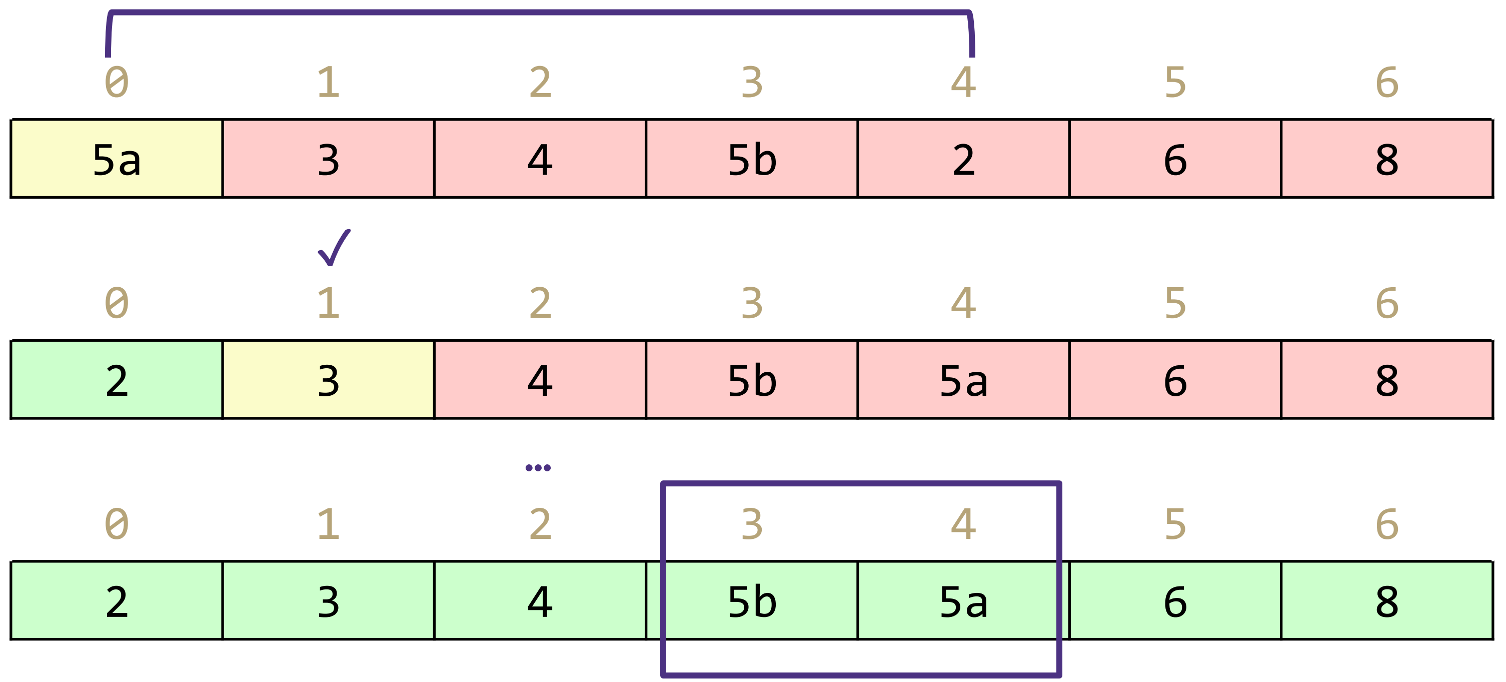
Selection Sort



```
public void selectionSort(collection) {
   for (entire list)
      int newIndex = findNextMin(currentItem);
      swap(newIndex, currentItem);
}
public int findNextMin(currentItem) {
   min = currentItem
   for (unsorted list)
      if (item < min)
         min = currentItem
   return min
}
public int swap(newIndex, currentItem) {
   temp = currentItem
   currentItem = newIndex
   newIndex = currentItem
}
```

Worst case runtime?	$\Theta(n^2)$
Best case runtime?	$\Theta(n^2)$
Average runtime?	$\Theta(n^2)$
Stable?	No
In-place?	Yes
Useful for:	Top K sort without needing extra space

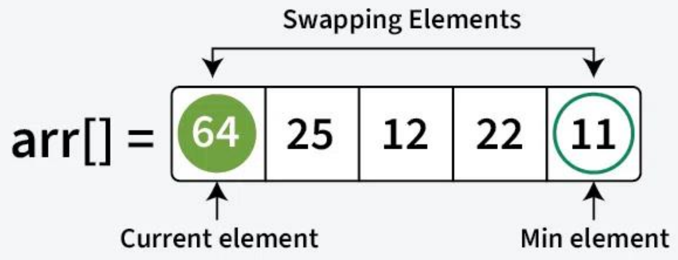
Selection Sort Stability



*Swapping non-adjacent items can result in instability of sorting algorithms

01 Step

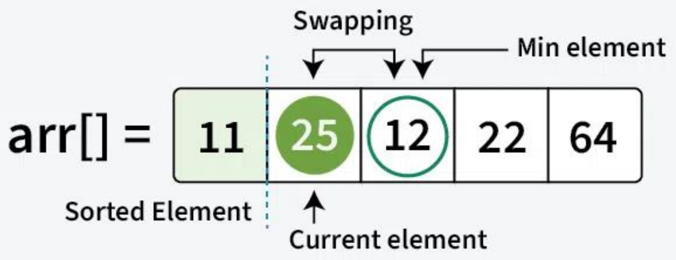
Start from the first element at index 0, find the smallest element in the rest of the array which is unsorted, and swap (11) with current element(64).



Selection Sort Algorithm

02 Step

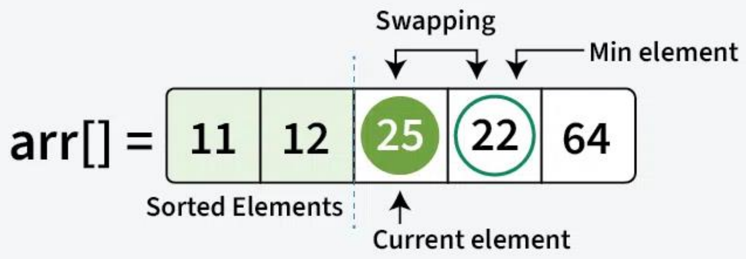
Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).



Selection Sort Algorithm

03 Step

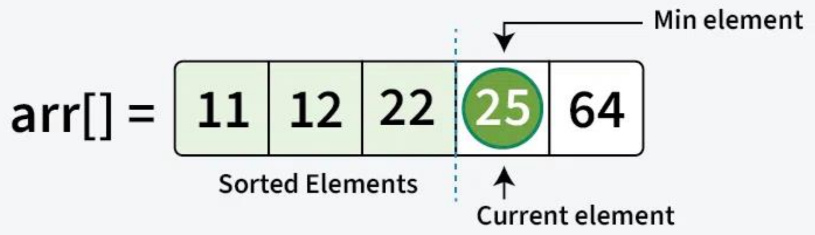
Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).



Selection Sort Algorithm

04 Step

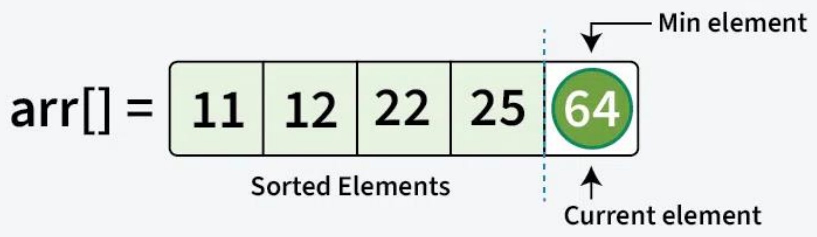
Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).



Selection Sort Algorithm

05 Step

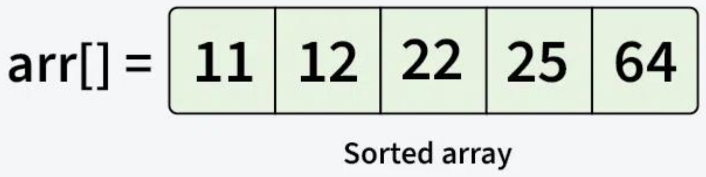
Move to element at index 4 (64), find the minimum from unsorted subarray and swap (64) with current element (64).



Selection Sort Algorithm

06 Step

We get the sorted array at the end.



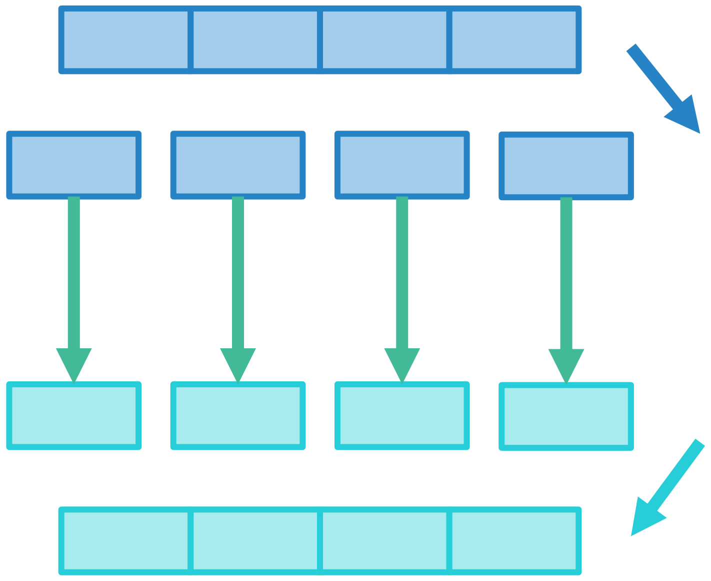
Selection Sort Algorithm

Principle 2: Divide and Conquer

General recipe:

1. **Divide** your work into smaller pieces (subproblems) recursively
2. **Conquer** the recursive subproblems
   - In many algorithms, conquering a subproblem requires no extra work beyond recursively dividing and combining it!
3. **Combine** the results of your recursive calls

Examples: Merge Sort, Quick Sort

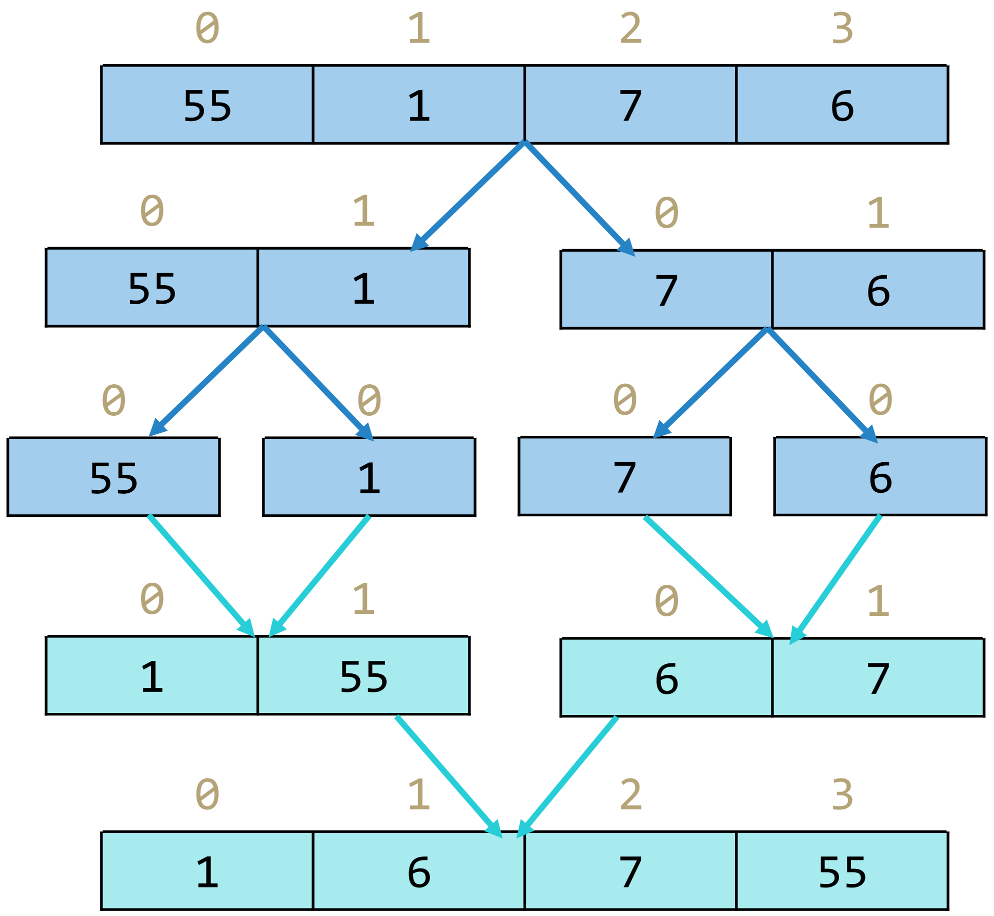


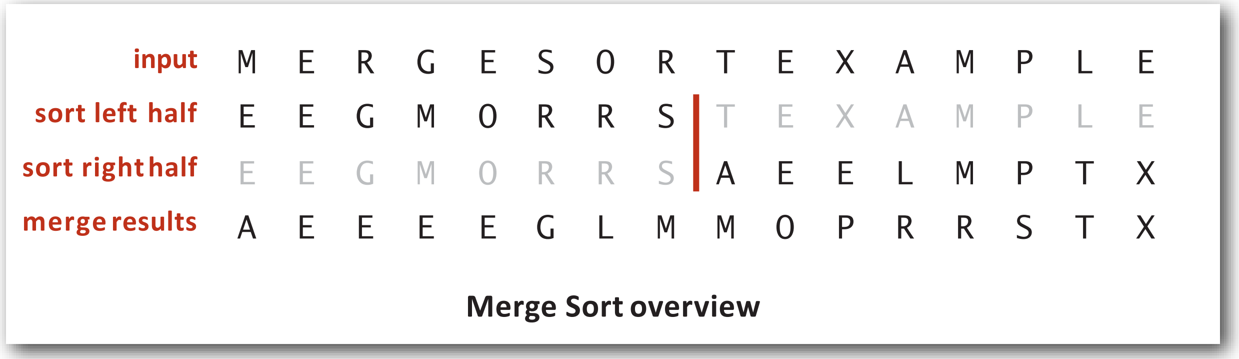
```
divideAndConquer(input) {
  if (small enough to solve):
    conquer, solve, return results
  else:
    divide input into a smaller pieces
    recurse on smaller pieces
    combine results and return
}
```

Merge Sort

Merge Sort

- Merge sort
  - Divide array into two halves. Recursively sort each half. Merge two halves
- Merge Sort Algorithm: A Step-by-Step Visualization, Quoc Dat Phung
  - https://www.youtube.com/watch?v=ho05egqcPl4





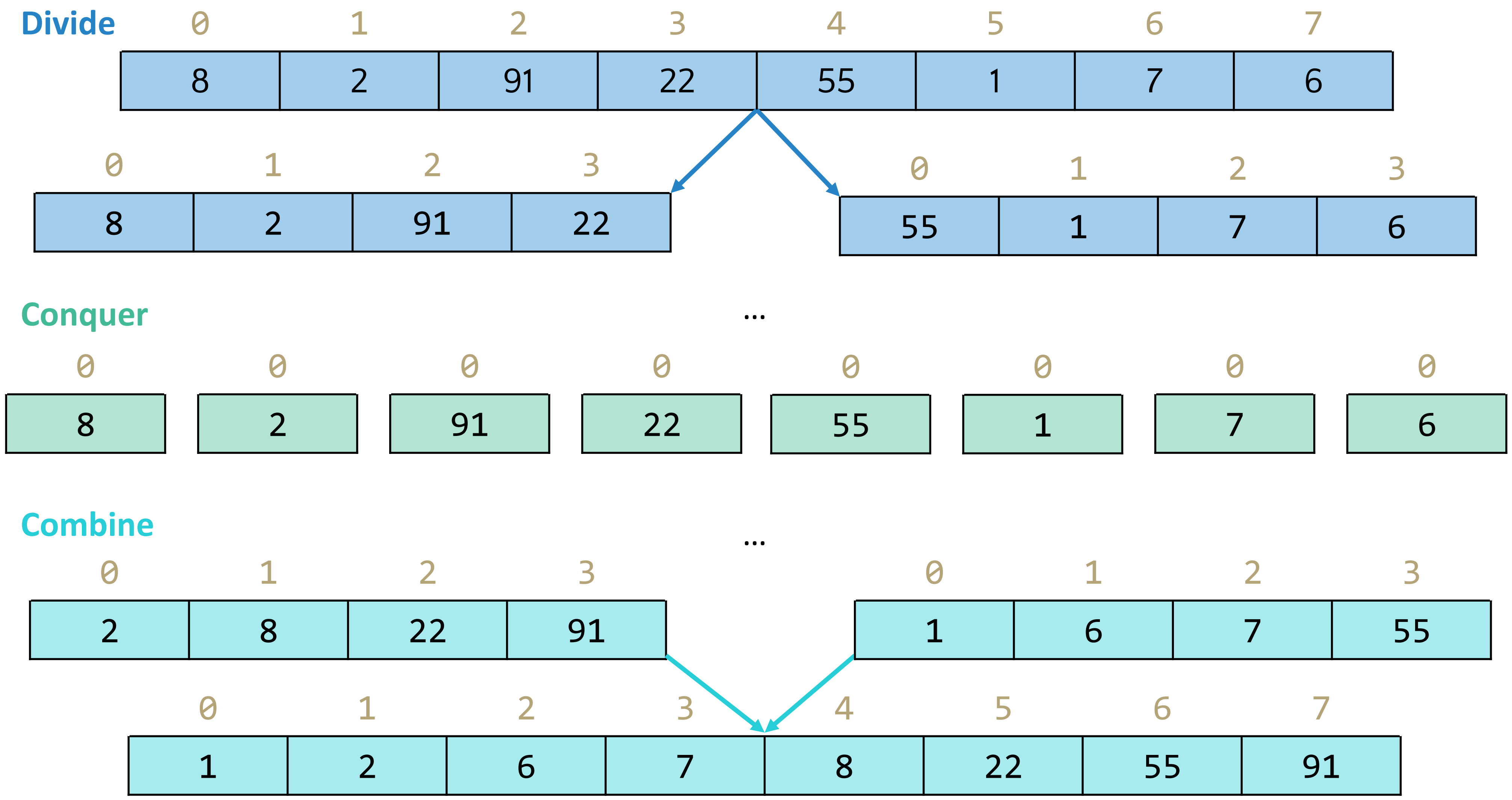
input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Merge Sort overview

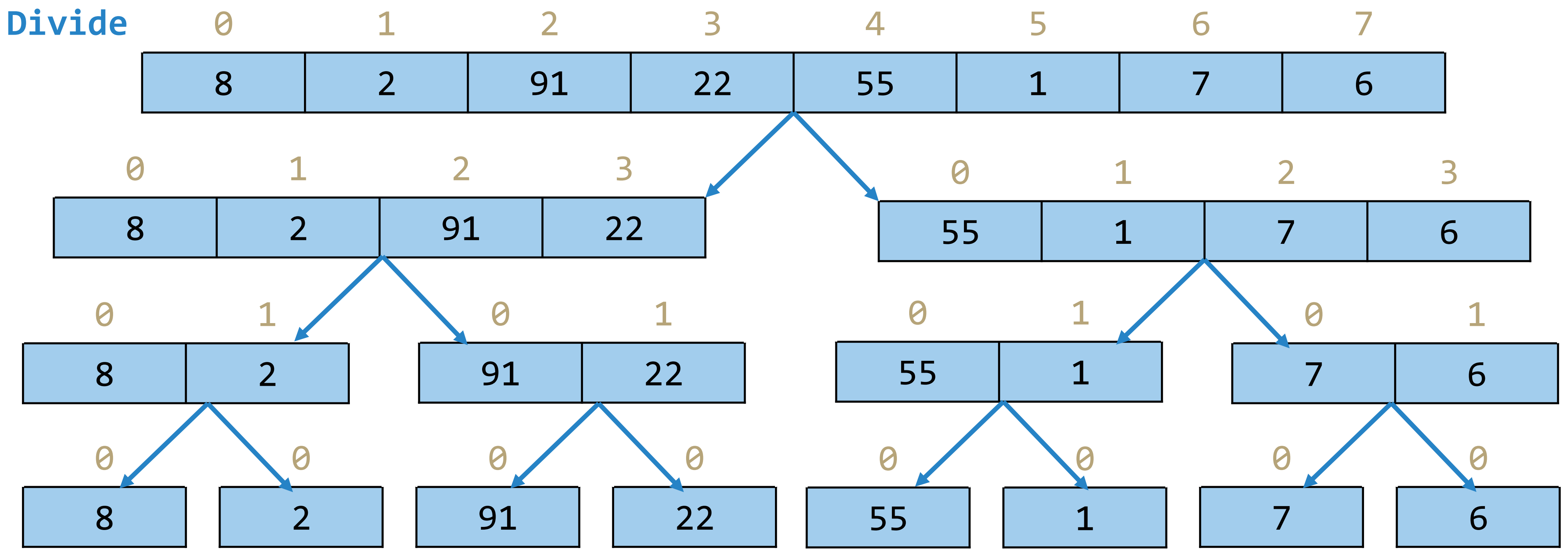
Merge Sort

Divide in half each time

Actual sorting happens here



Merge Sort: Divide Step

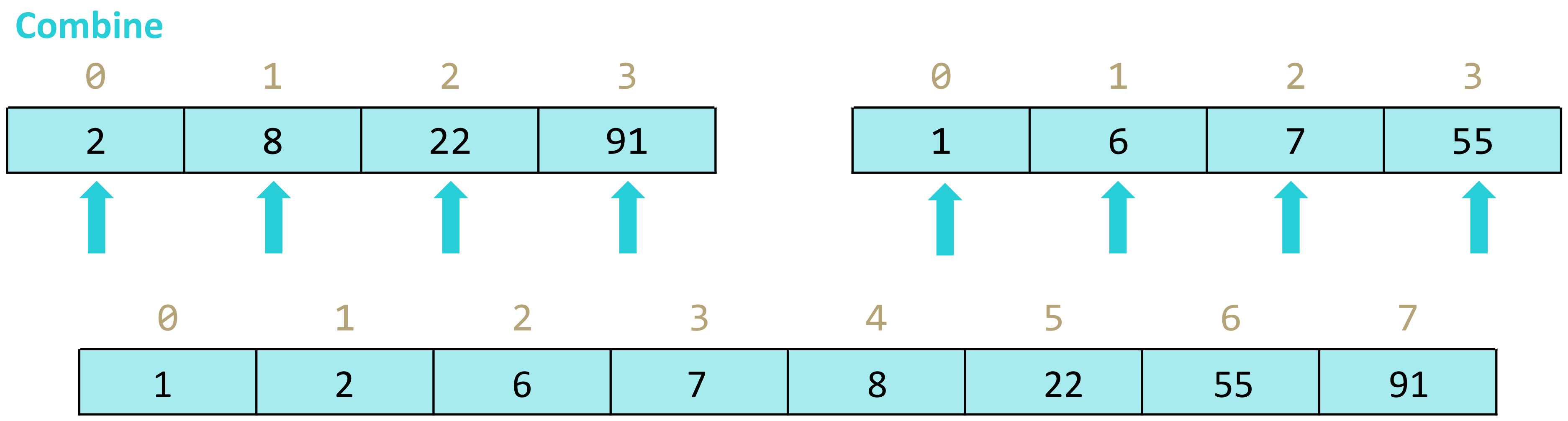


Recursive Case: split the array in half and recurse on both halves

When array hits size 1, stop dividing.

Sort the pieces through recursion

Merge Sort: Combine Step



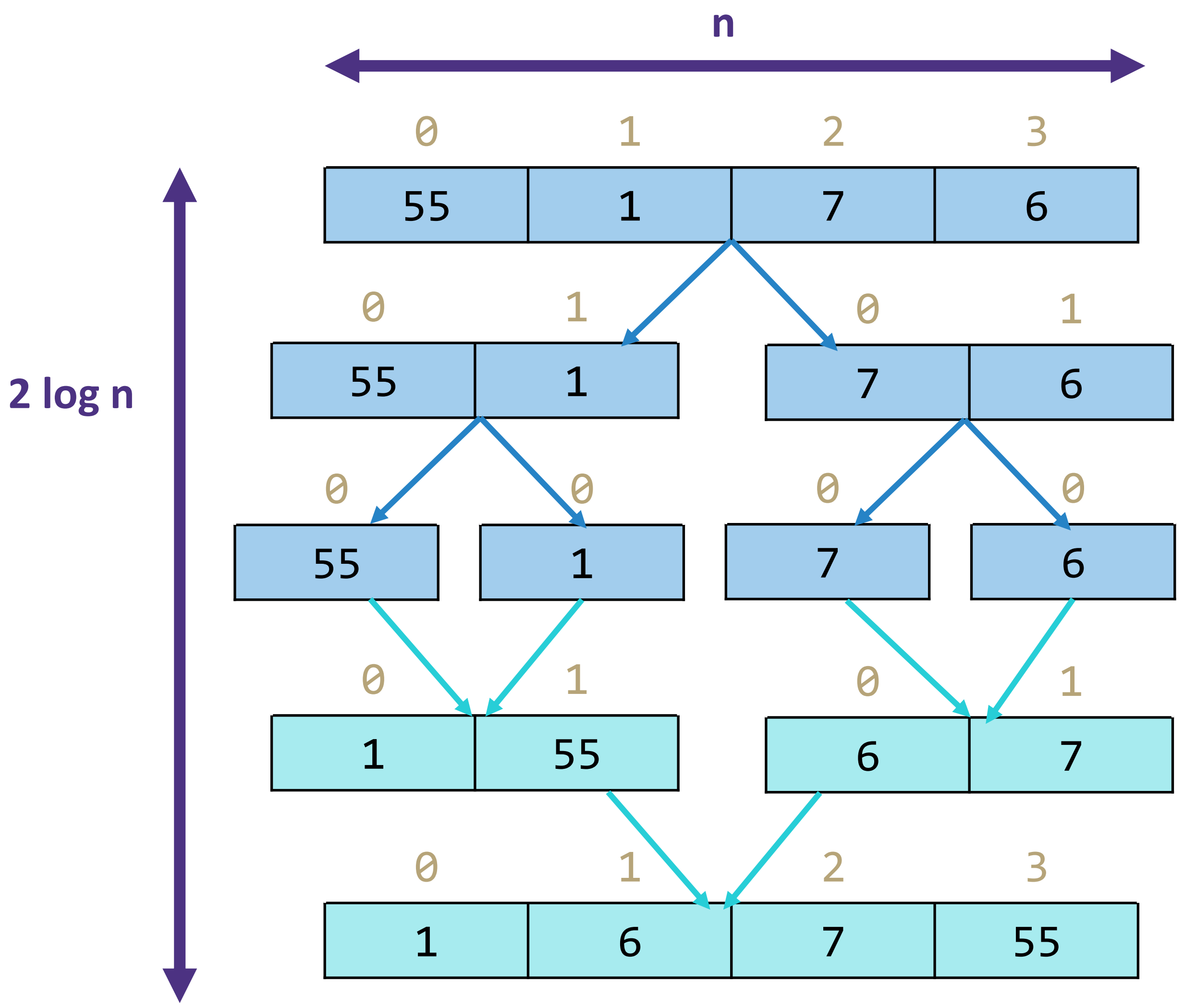
Combining two *sorted* arrays:

1. Initialize two pointers to start of both arrays
2. Repeat until all elements are added:
    1. Add the smaller element of the two pointers to the result array
    2. Move that pointer forward one spot

Merge Sort

```
mergeSort(list) {
   if (list.length == 1):
      return list
   else:
      smallerHalf = mergeSort(new [0, ..., mid])
      largerHalf = mergeSort(new [mid + 1, ...])
      return merge(smallerHalf, largerHalf)
}
```

Worst case runtime?	$O(n \log n)$
Best case runtime?	Same
In Practice runtime?	Same
Stable?	Yes
In-place?	No



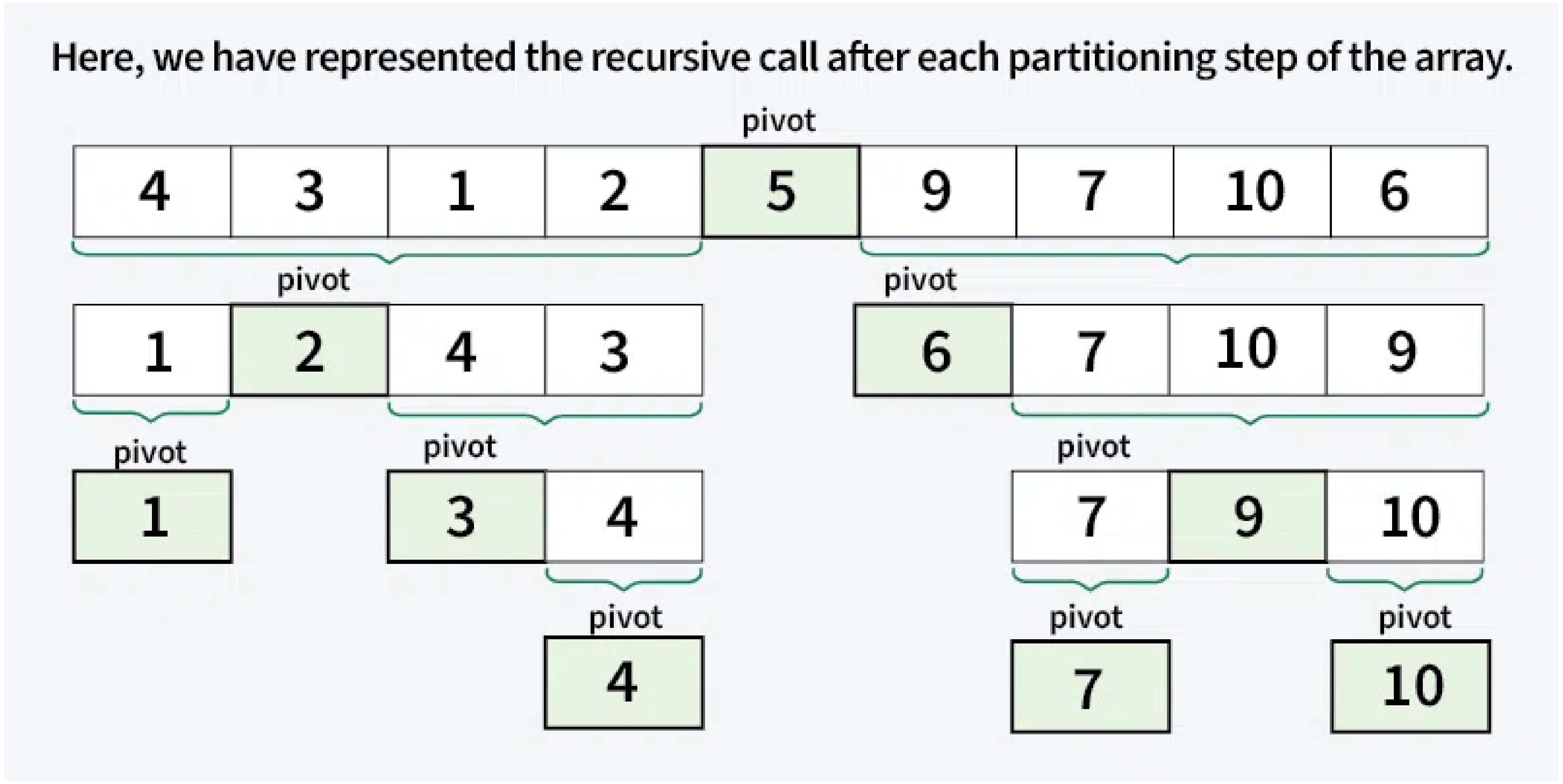
Quick Sort

Quick Sort

- **Choose a Pivot**: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median)
- **Partition the Array**:
  - 1. The pivot is compared with each element in the array
  - 2. Elements smaller than the pivot are moved to its left
  - 3. Elements larger than the pivot are moved to its right
  - 4. The pivot is placed in its final sorted position
- **Recursively Call**: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
- **Base Case**: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.
- Quick sort in 4 minutes (recommended)
  - https://www.youtube.com/watch?v=Hoixgm4-P4M

Quick Sort Example

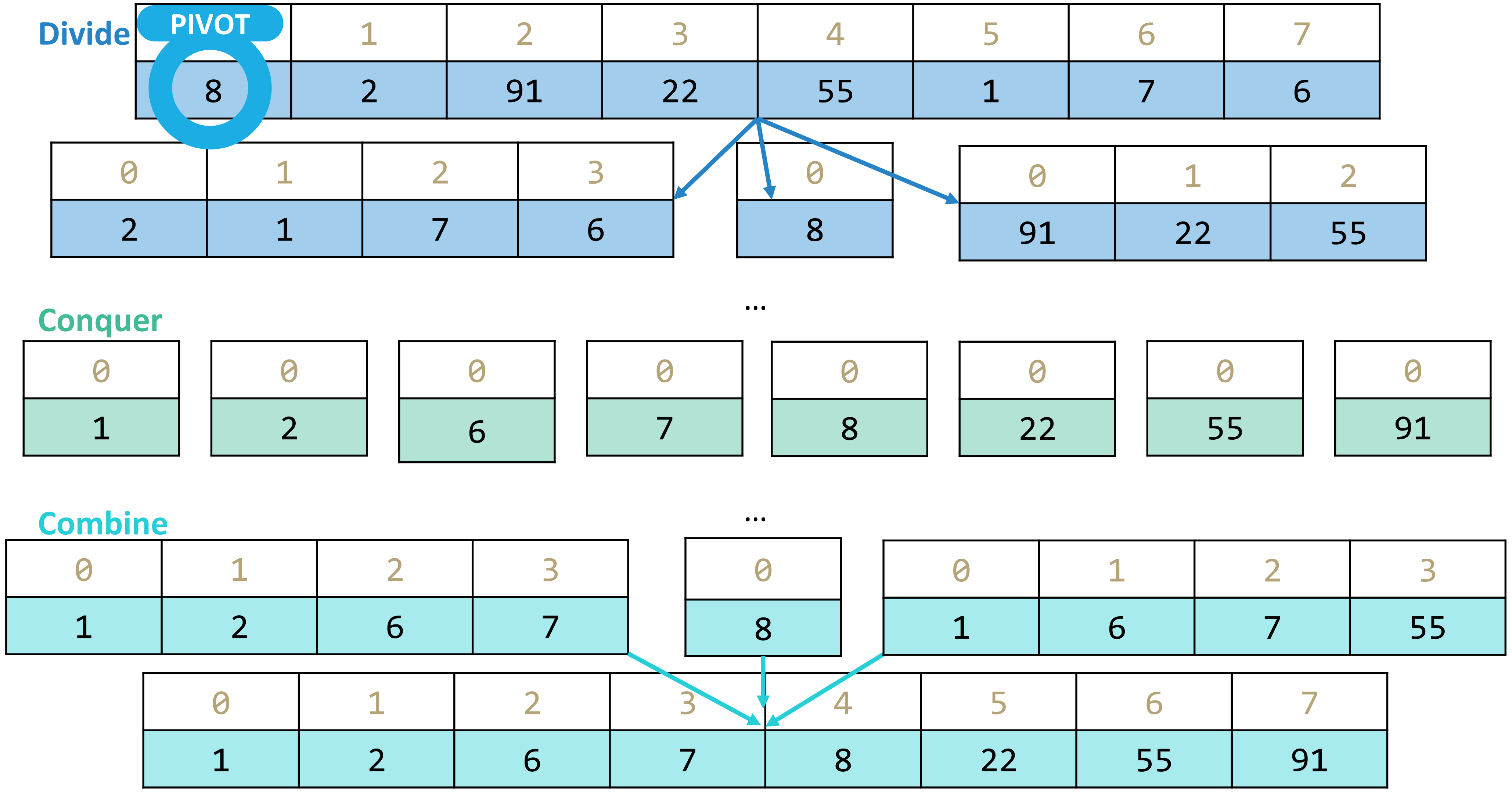
- Input array [4, 3, 9, 7, 1, 2, 10, 6, 5]
- Choose the last element as pivot



Quick Sort (v1)

Divide: Choose a "pivot" element, partition array relative to it

Combine: Concatenate the now-sorted arrays



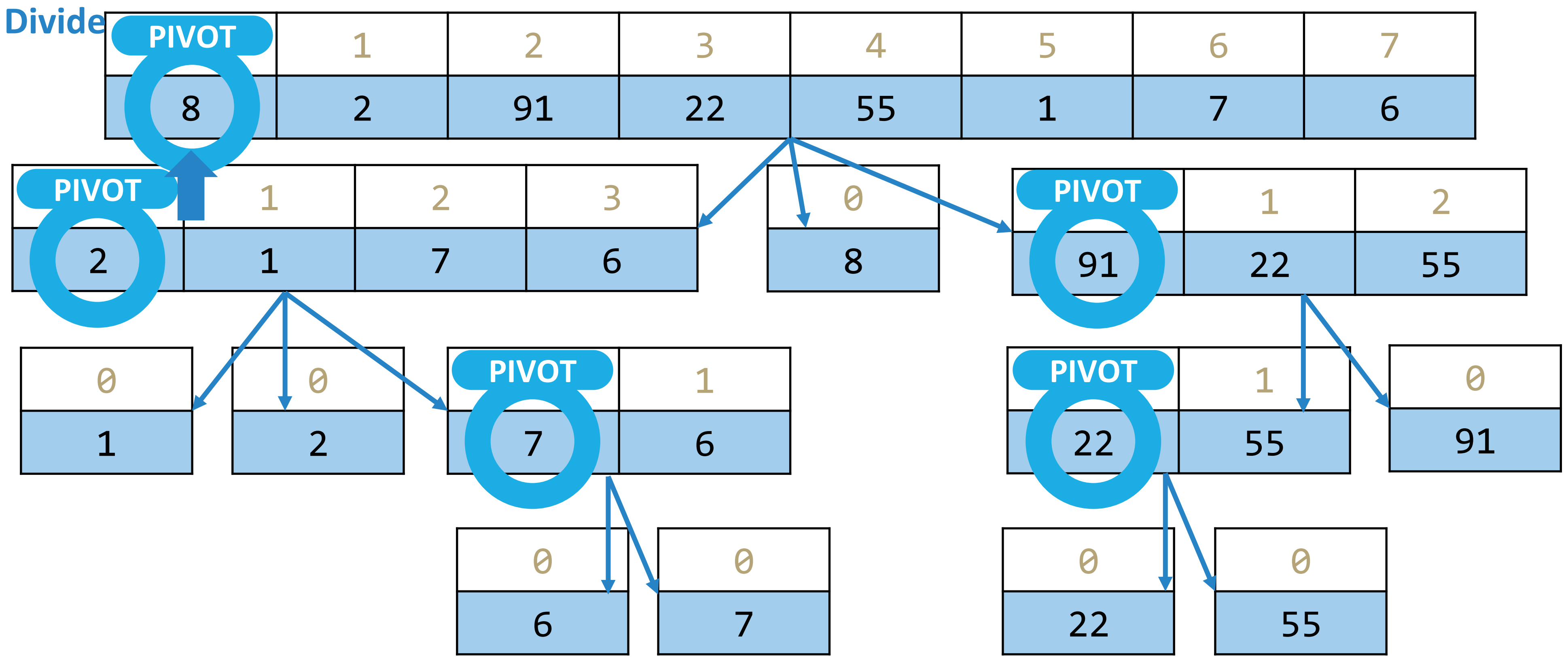
Quick Sort (v1): Divide Step

Recursive Case:

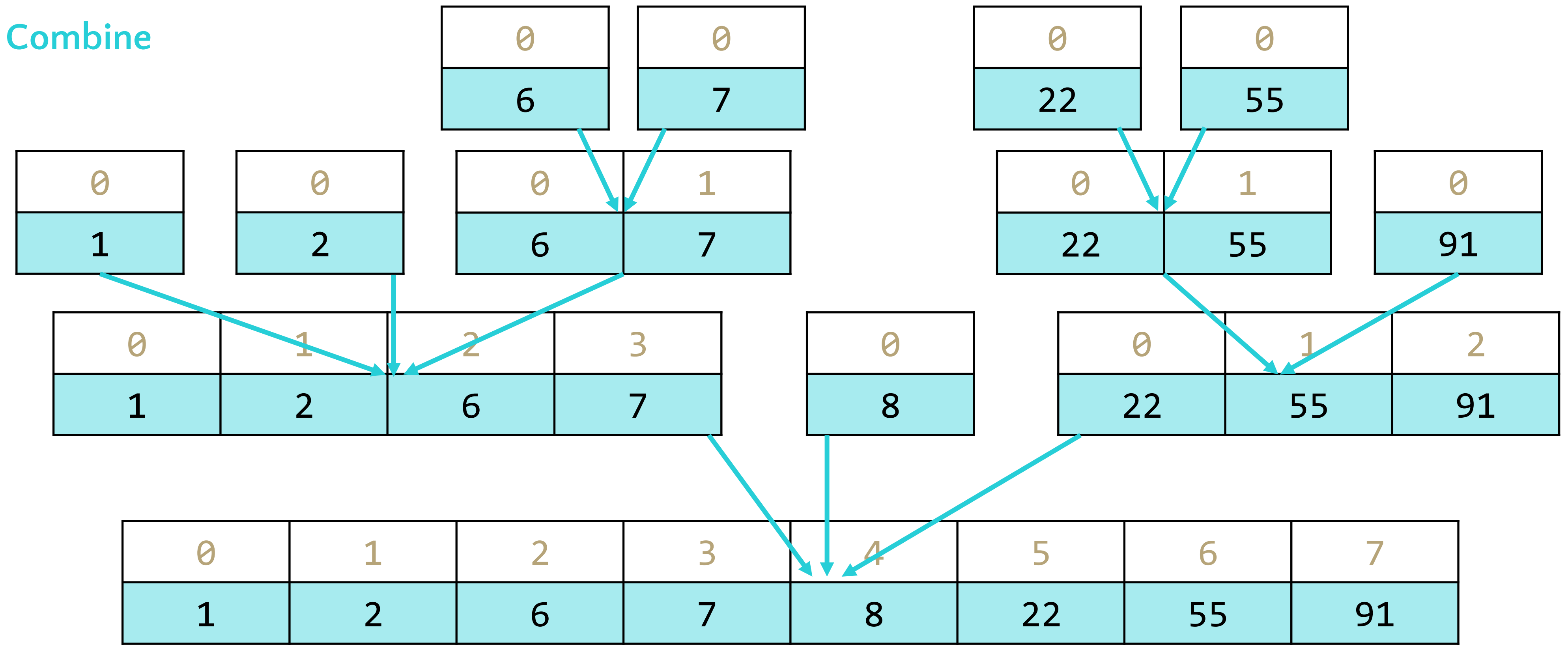
- Choose a “pivot” element
- Partition: linear scan through array, add smaller elements to one array and larger elements to another
- Recursively partition

Base Case:

- When array hits size 1, stop dividing



Quick Sort (v1): Combine Step



Simply concatenate the arrays that were created earlier. Partition step already left them in order

Quick Sort (v1)

```
quickSort(list) {
   if (list.length == 1):
      return list
   else:
      pivot = choosePivot(list)
      smallerHalf = quickSort(getSmaller(pivot, list))
      largerHalf = quickSort(getBigger(pivot, list))
      return smallerHalf + pivot + largerHalf
}
```

Worst case runtime? $O(n^2)$

Number of compares is quadratic:
$n + (n-1) + (n-2) + \cdots + 1$
$= O(n^2)$

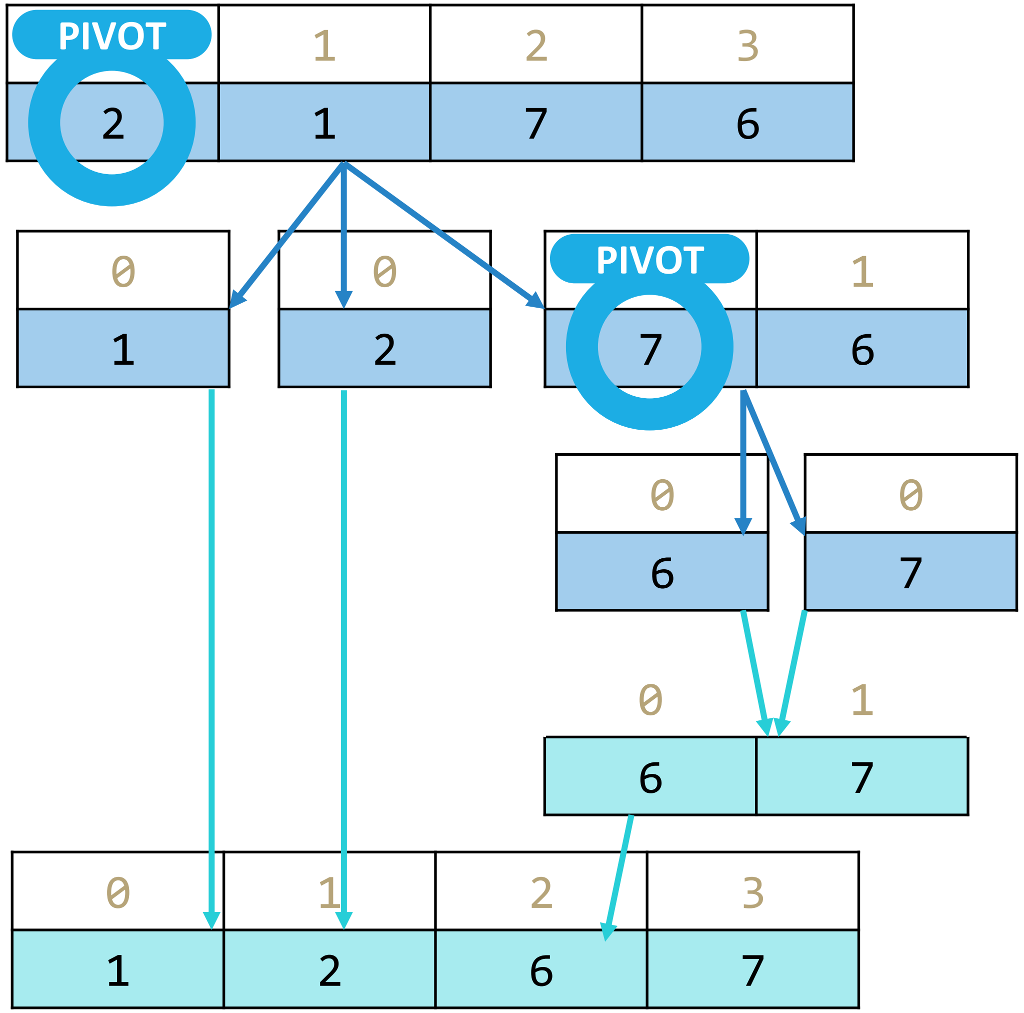
Best case runtime? $O(n \log n)$

Average runtime? $O(n \log n)$
(non-trivial derivation)

Stable? No

In-place? Can be

Worst case: Pivot only chops off one value
Best case: Pivot divides each array in half



Strategies for Choosing a Pivot

Just take the first element

- Very fast
- But has worst case: for example, sorted lists have $\Omega(n^2)$ runtime

Take the median of the full array

- Can find the median in $O(n)$ time (QuickSelect). It’s complicated
- Worst case is $O(n \log n)$ … but the constant factors are large. No one does quicksort this way.

Take the median of the first, last, and middle element

- Makes pivot slightly more content-aware, at least won’t select very smallest/largest
- Worst case is still $O(n^2)$, but on real-world data tends to perform well!

Pick a random element

- Get $O(n \log n)$ runtime with probability at least $\mathbf{1-1/}\boldsymbol{n}^{\mathbf{2}}$
- No simple worst-case input (e.g. sorted, reverse sorted)

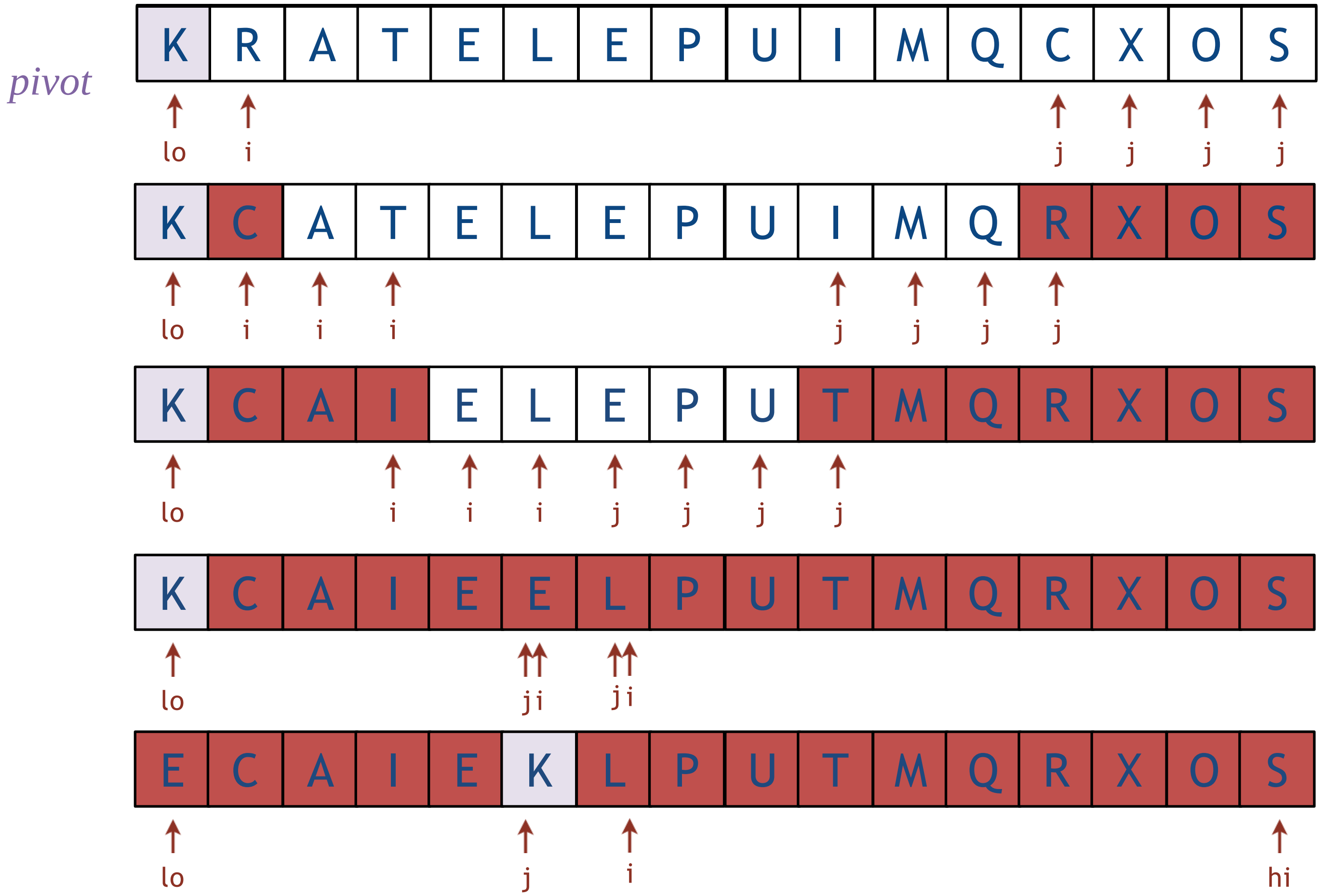
Quick Sort (v2: In-Place) Example I

Repeat until i and j pointers cross

- Scan i from left to right so long as (a[i] < a[lo]).
- Scan j from right to left so long as (a[j] > a[lo]).
- Exchange a[i] with a[j].

When pointers cross.

- Exchange a[lo] with a[j].



pivot
K R A T E L E P U I M Q C X O S
lo i j j j j
K C A T E L E P U I M Q R X O S
lo i i i j j j j
K C A I E L E P U T M Q R X O S
lo i i i j j j j
K C A I E E L P U T M Q R X O S
lo j i j i
E C A I E K L P U T M Q R X O S
lo j i hi

Quick Sort (v2: In-Place) Example II

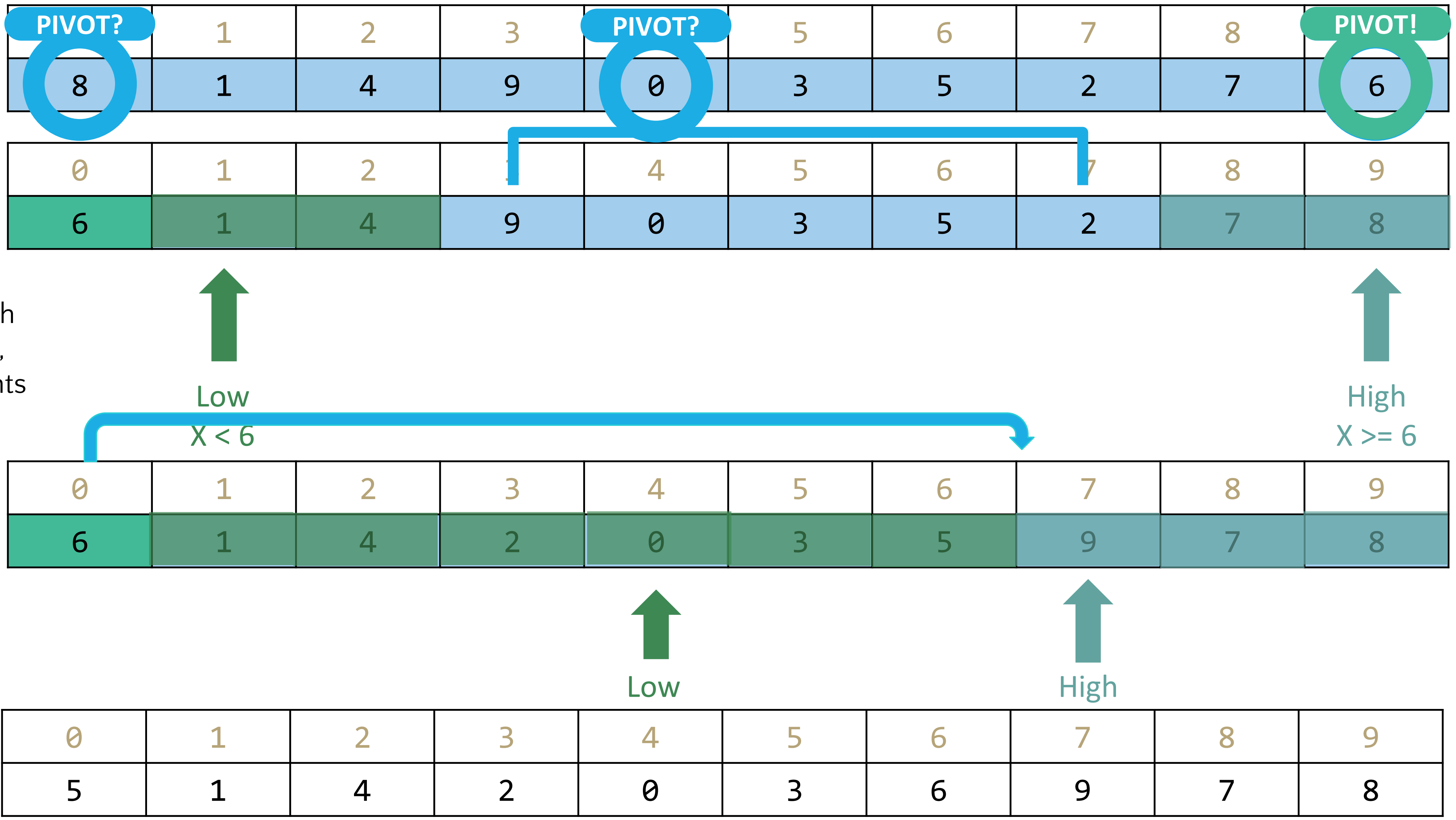
Divide

Select a pivot

Move pivot out of the way

Bring low and high pointers together, swapping elements if needed

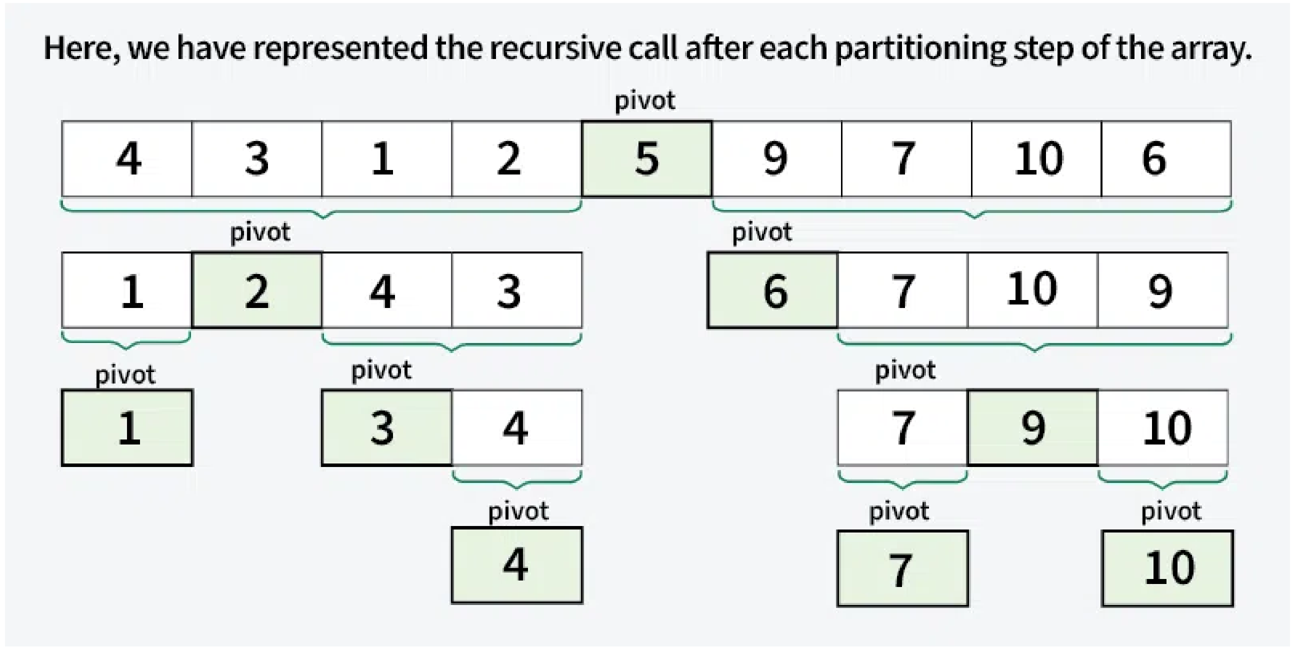
Meeting point is where pivot belongs; swap in. Now recurse on smaller portions of same array!

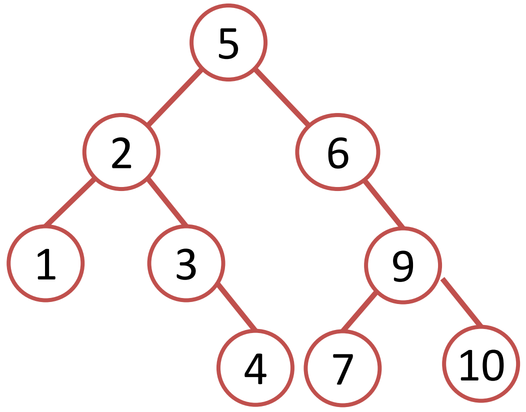


Quick Sort is Equivalent to Sorting by BST

Key idea: compareTo calls are same for Binary Search Tree (BST) insert and Quick Sort.

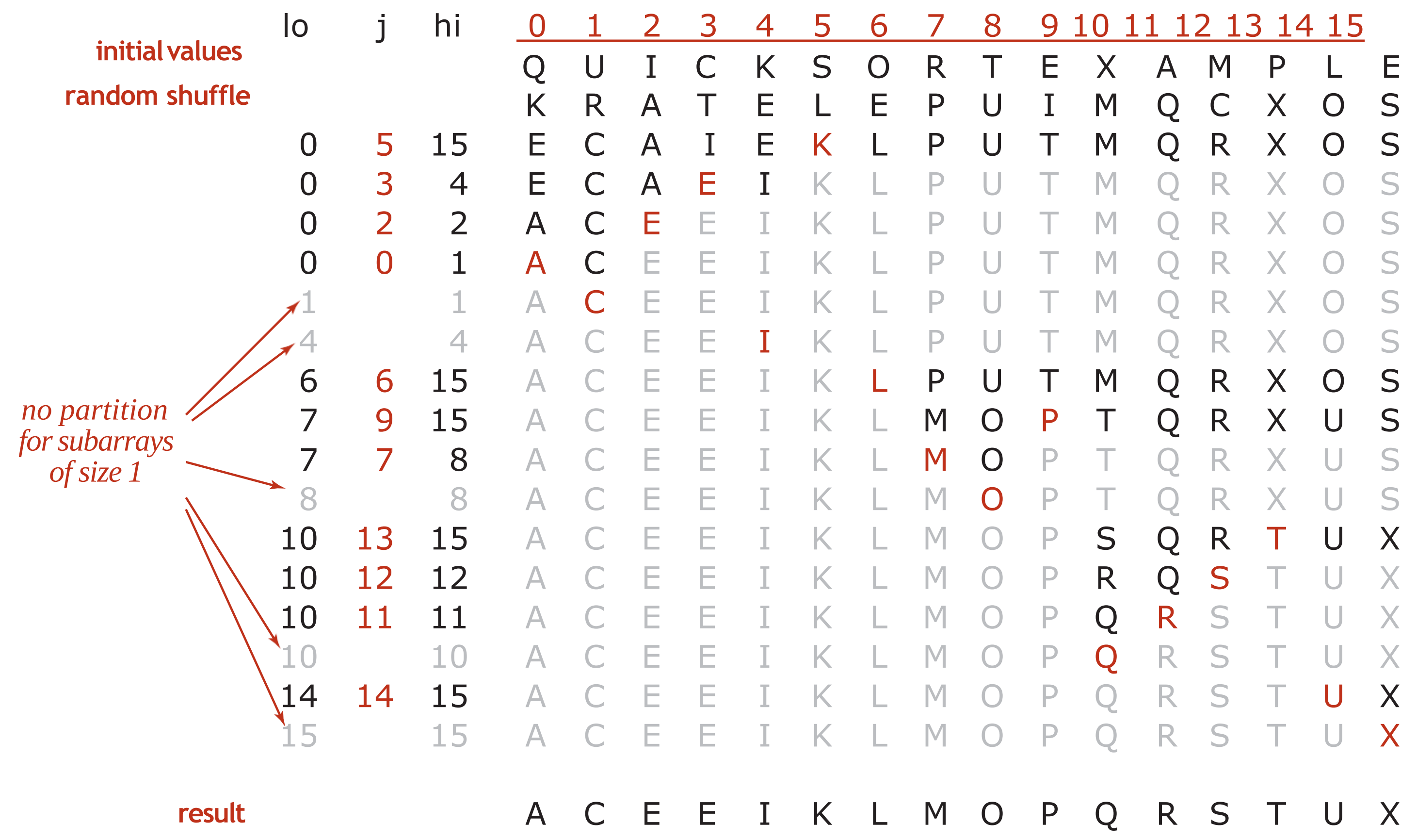
- Every number gets compared to 5; 1, 3, 4 get compared to only 2.
- Recall: Insertion into a BST has average-case complexity O(N log N)


Here, we have represented the recursive call after each partitioning step of the array.



BST In-order traversal gives sorted list [1,2,3,4,5,6,7,9,10]

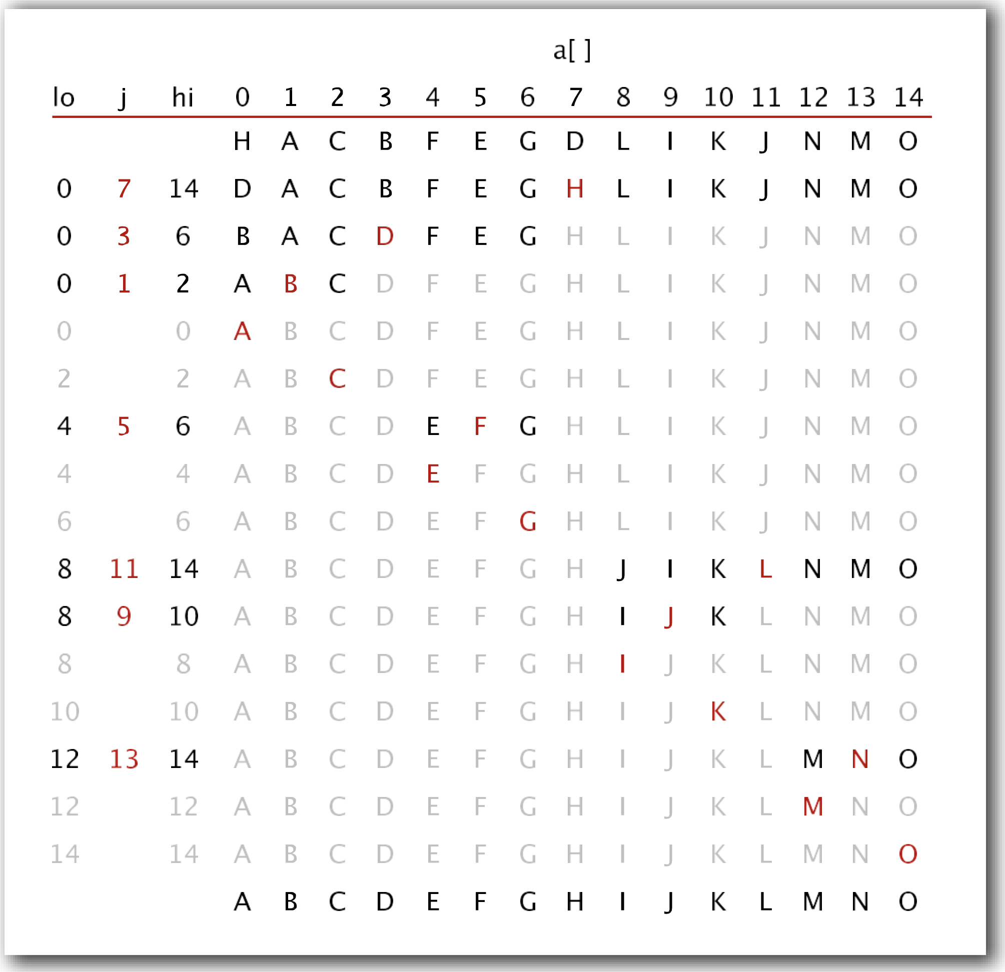
Trace of a Quick Sort Example


no partition for subarrays of size 1

	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

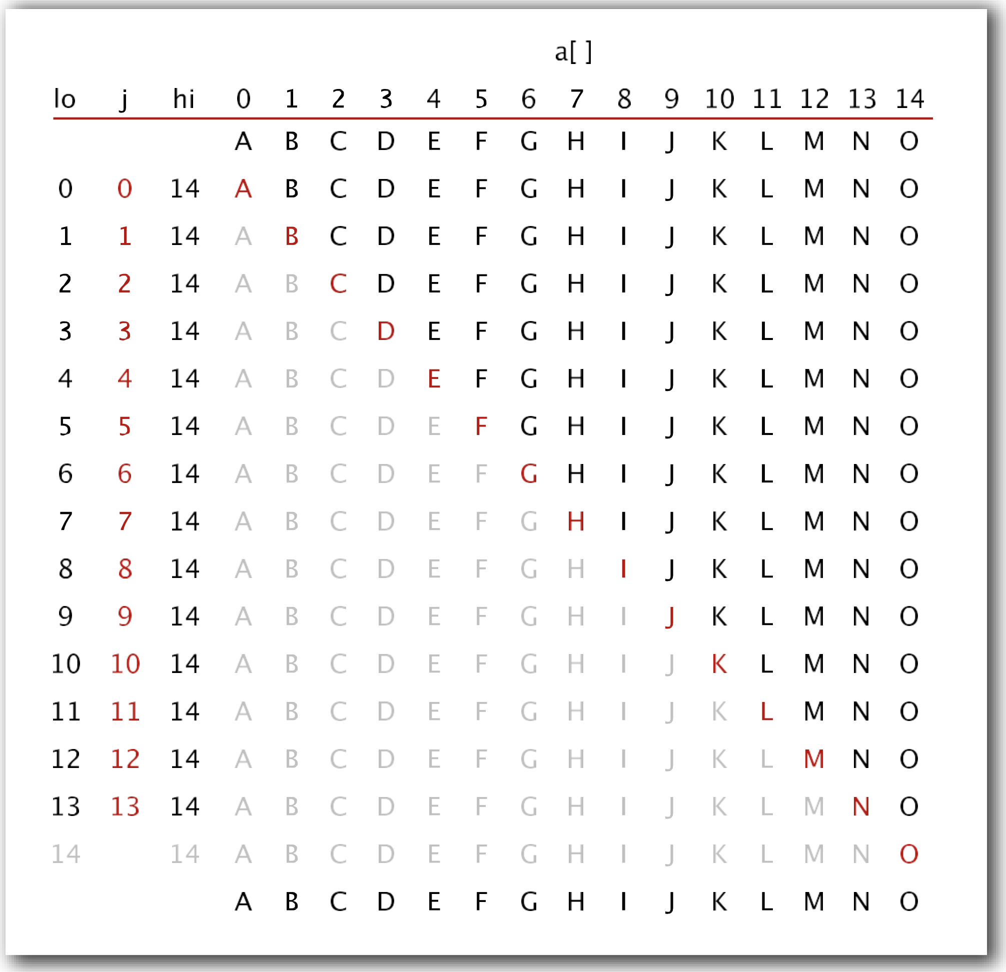
Quick Sort trace (array contents after each partition)

Best-Case vs. Worst-Case



a[ ]

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O



a[ ]

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

When the list is randomly shuffled, and if we happen to always pick the median element as the pivot, then Quick Sort has the best-case complexity of O(n log n). This corresponds to a balanced BST

When the list is already sorted, and if we always pick the first element as the pivot, then Quick Sort has the worst-case complexity of $O(n^2)$. This corresponds to a extremely unbalanced BST.

Heap Sort

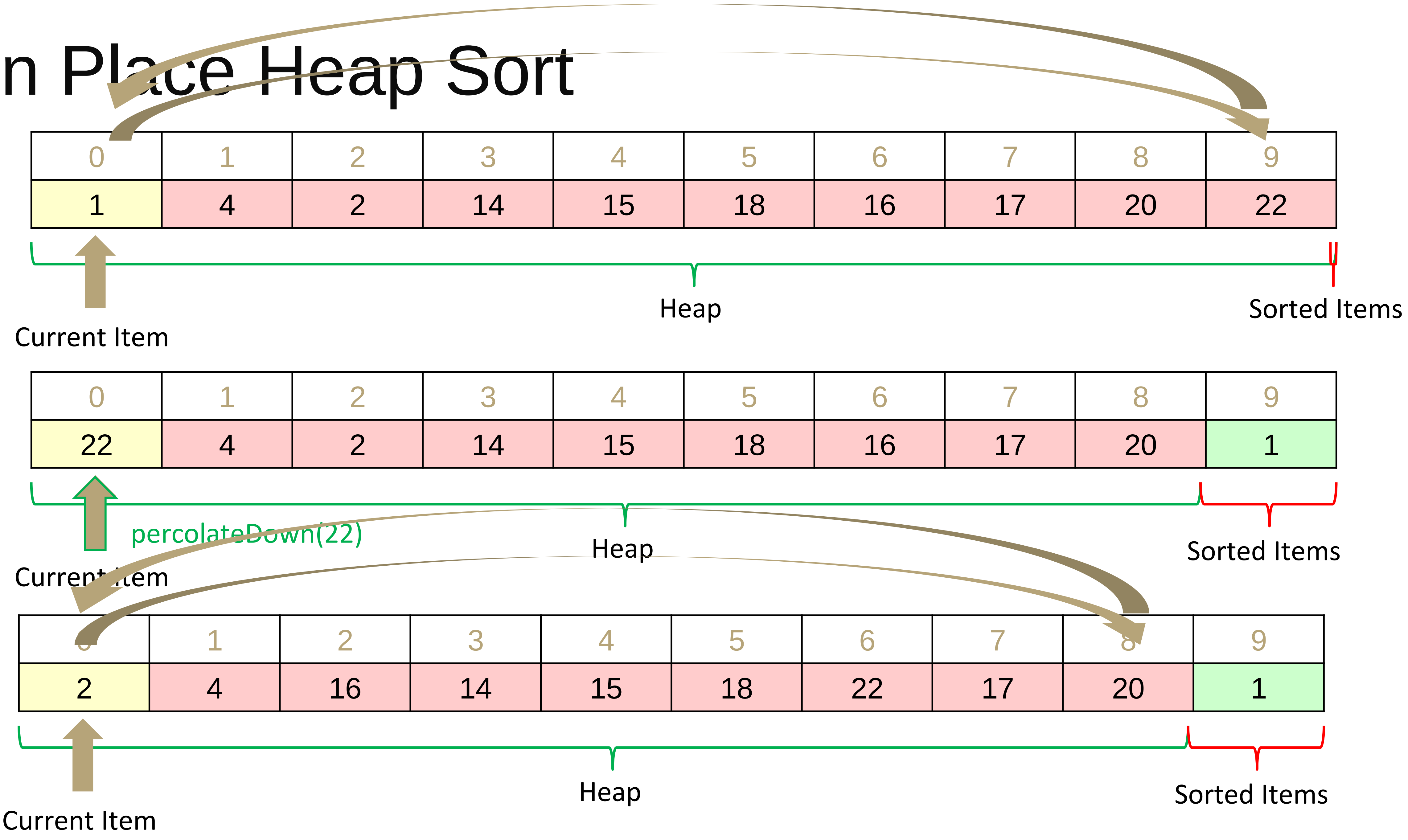
Heap Sort

1. Run Floyd's buildHeap
2. Call removeMin n times

```
public void heapSort(input) {
   E[] heap = buildHeap(input)
   E[] output = new E[n]
   for (n)
      output[i] = removeMin(heap)
}
```

Worst case runtime?	$O(n \log n)$
Best case runtime?	$O(n)$
Average runtime?	$O(n \log n)$
Stable?	No
In-place?	Yes

In Place Heap Sort

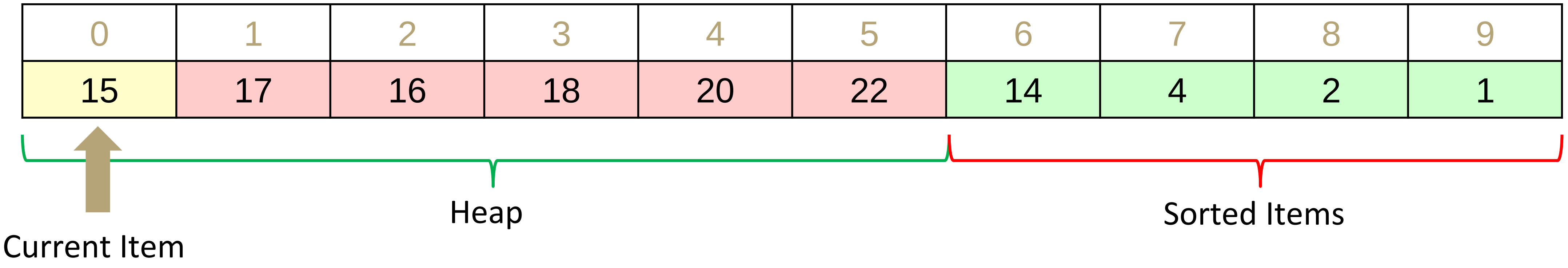


0	1	2	3	4	5	6	7	8	9
1	4	2	14	15	18	16	17	20	22

0	1	2	3	4	5	6	7	8	9
22	4	2	14	15	18	16	17	20	1

0	1	2	3	4	5	6	7	8	9
2	4	16	14	15	18	22	17	20	1

In Place Heap Sort



```
public void inPlaceHeapSort(input) {
   buildHeap(input) // alters original array
   for (n : input)
      input[n – i - 1] = removeMin(heap)
}
```

Complication: final array is reversed! Lots of fixes:

- Run reverse afterwards O(*n*)
- Use a max heap
- Reverse compare function to emulate max heap

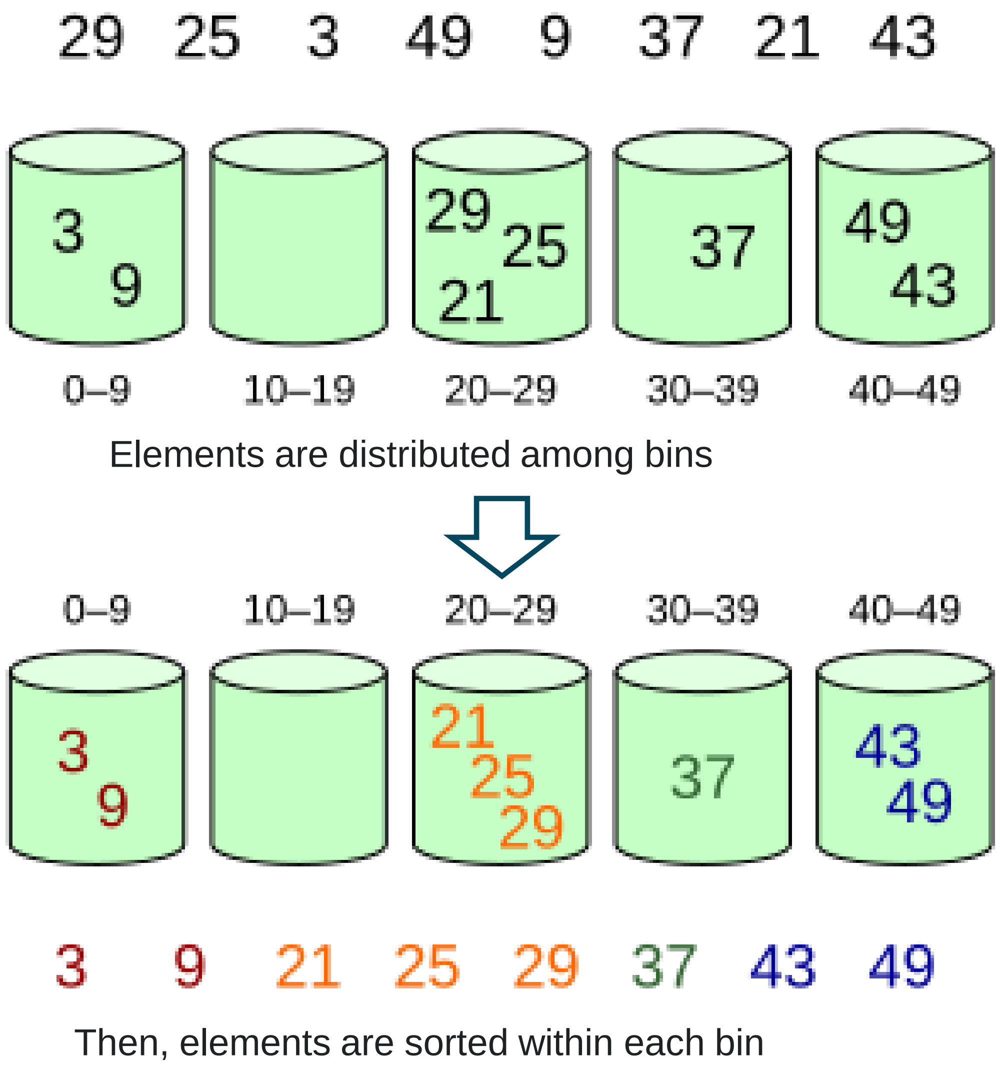
Worst case runtime?	$O(n \log n)$
Best case runtime?	$O(n)$
Average runtime?	$O(n \log n)$
Stable?	No
In-place?	Yes

Bucket Sort

Bucket Sort

- Bucket sort is a comparison sort algorithm that works by distributing the elements of an array into a number of buckets and then each bucket is sorted individually using a stable sorting algorithm, e.g., Insertion Sort or Merge Sort.
- This algorithm is efficient when the input is uniformly distributed over a range.

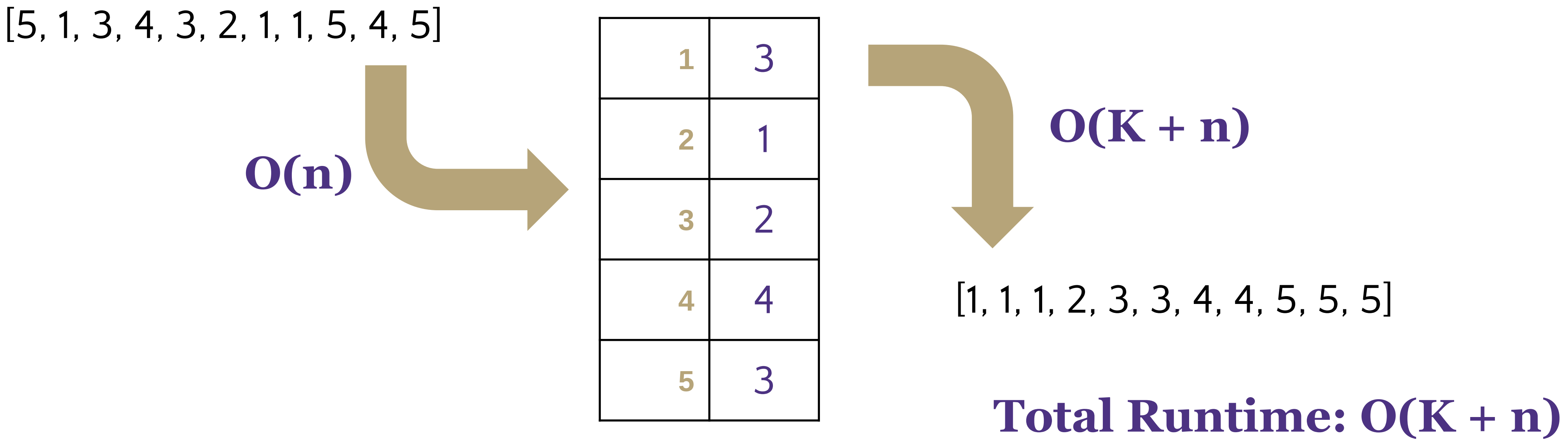
Bucket Sort | GeeksforGeeks
https://www.youtube.com/watch?v=VuXbEb5ywrU



https://en.wikipedia.org/wiki/Bucket_sort

Bucket Sort (aka Bin Sort)

- If all values are ints known to be in the range of 1 - K
- Create array of size K and put each element in its proper bucket ("scatter")
  - If elements are only ints simply store count of ints in each bucket
- Output results via linear pass through array of buckets ("gather")



[5, 1, 3, 4, 3, 2, 1, 1, 5, 4, 5]

O(n)

1	3
2	1
3	2
4	4
5	3

O(K + n)

[1, 1, 1, 2, 3, 3, 4, 4, 5, 5, 5]

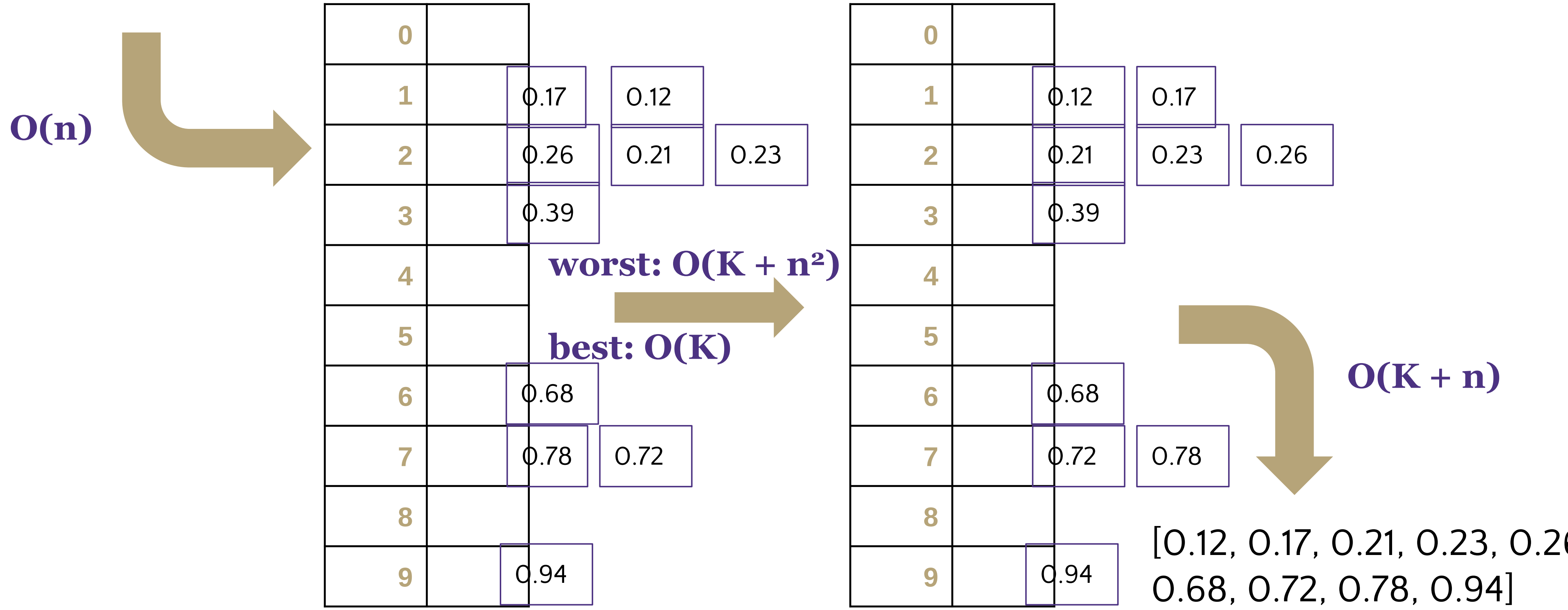
Total Runtime: O(K + n)

Bucket Sort | GeeksforGeeks
https://www.youtube.com/watch?v=VuXbEb5ywrU

Bucket Sort with Data

- Make buckets of array of lists
- Put items into bucket, use **insertion sort** to sort individual buckets

[0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]



[0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]

Bucket Sort

```
function bucketSort(array, k) is
    buckets ← new array of k empty lists
    M ← 1 + the maximum key value in the array
    for i = 0 to length(array) do
        insert array[i] into buckets[floor(k × array[i] / M)]
    for i = 0 to k do
        nextSort(buckets[i])
    return the concatenation of buckets[0], ...., buckets[k]
```

Worst case runtime?	O(K + n) for ints O(K + n^2) for data if insertion sort is used
Best case runtime?	O(n)
Average runtime?	O(n) if K ≅ n, always for ints, and if values are evenly distributed for data
Stable?	Can be because insertion sort
In-place?	No
Useful for:	When range, K, is smaller or not much larger than n (not many duplicates) Not good when K >> N, wasted space

Specialized Sorts (“Niche Sorts”)

So far we’ve learned about comparison sorts

- work on any comparable object
- have a best case lower bound of $O(n \log n)$

This is because to sort using comparisons requires all elements to be compared against one another

- n runtime to process all values into some ordered structure (tree)
- O(log n) runtime to remove items from structure in sorted order

What if we didn’t need to compare each element, what if we built a sort based on inherent knowledge about the ordering of numbers?

Specialized Sorts: Sorting algorithms that only work on data types with ordering already known to computer logic: numbers

- Bucket Sort for ints
- Radix Sort

Radix and Radix Sort

- Radix = “The base of a number system”
  - Number of unique digits, including the digit zero, used to represent numbers
- Radix of numbers:
  - Binary numbers have a radix of 2
  - decimals have a radix of 10
  - hexadecimals have a radix of 16
- Radix sort was first used in 1890 U.S. census by Hollerith
- Efficient O(n) complexity
- Not in-place sorting
  - May use more space than other sorting algorithms
- Basic idea: Bucket sort on each digit, from least significant digit to most significant digit.

Radix Sort Algorithm

```
radix_sort(A, n, k) {
    /* A: array; n: number of elements; k: number of digits in the
    largest number */
    create buckets  (buckets can be arrays or lists)
    for (d = 0; d <k; d++) {
        /* sort A using digit position d as the key. */
        for (i = 0; i<n; i++) {
            p = the d-th digit  (from right) of A[i]
            Add A[i] to bucket p
        }
        A = Join the buckets
    }
}
```

Time complexity O(n)

Bucket Sort as used in Radix Sort

- Use bucket array of size R for radix of R
- Put elements into the correct bucket in the array
- R = 5; unique digits (0,1,2,3,4); list = (0,1,3,4,3,2,1,1,0,4,0)

Buckets	
= 0	0,0,0
= 1	1,1,1
= 2	2
= 3	3,3
= 4	4,4

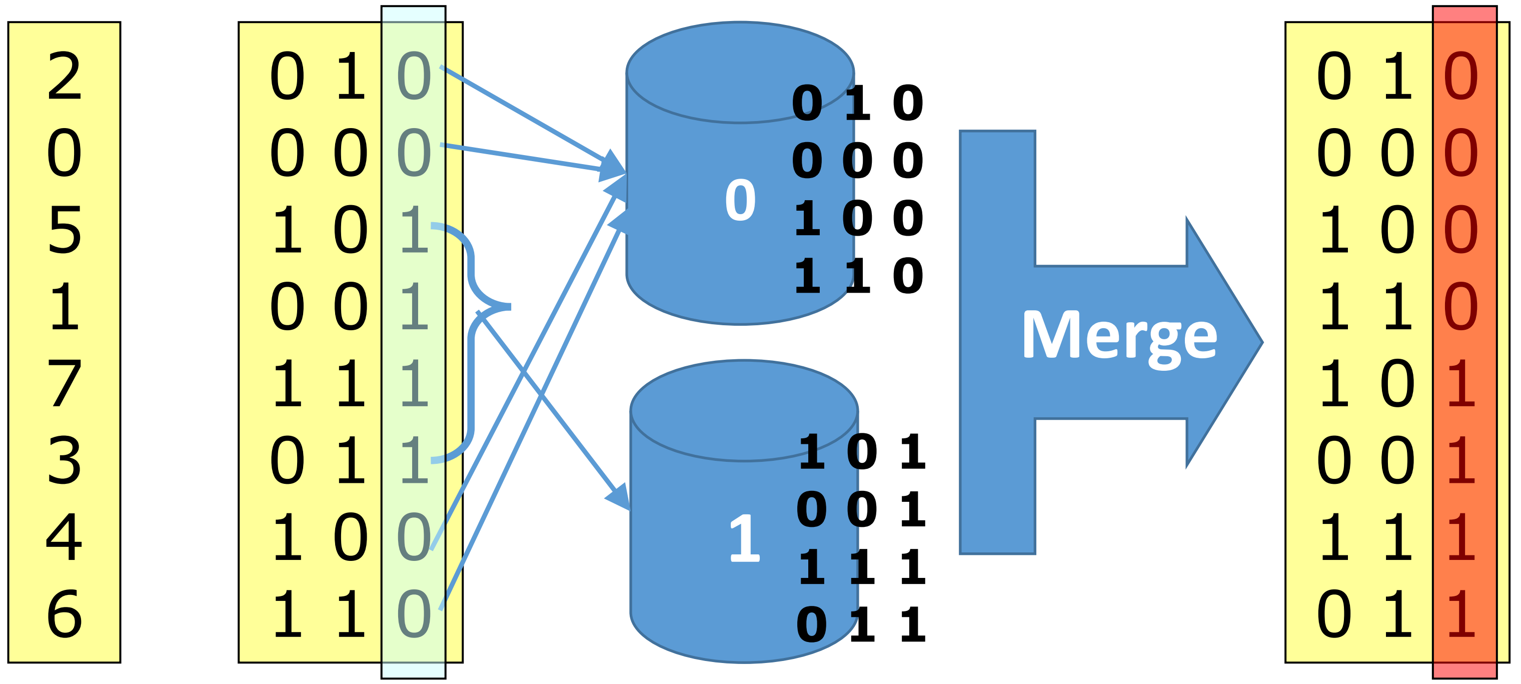
Sorted list:
0,0,0,1,1,1,2,3,3,4,4

Radix Sort: bucket sort on every digit/bit

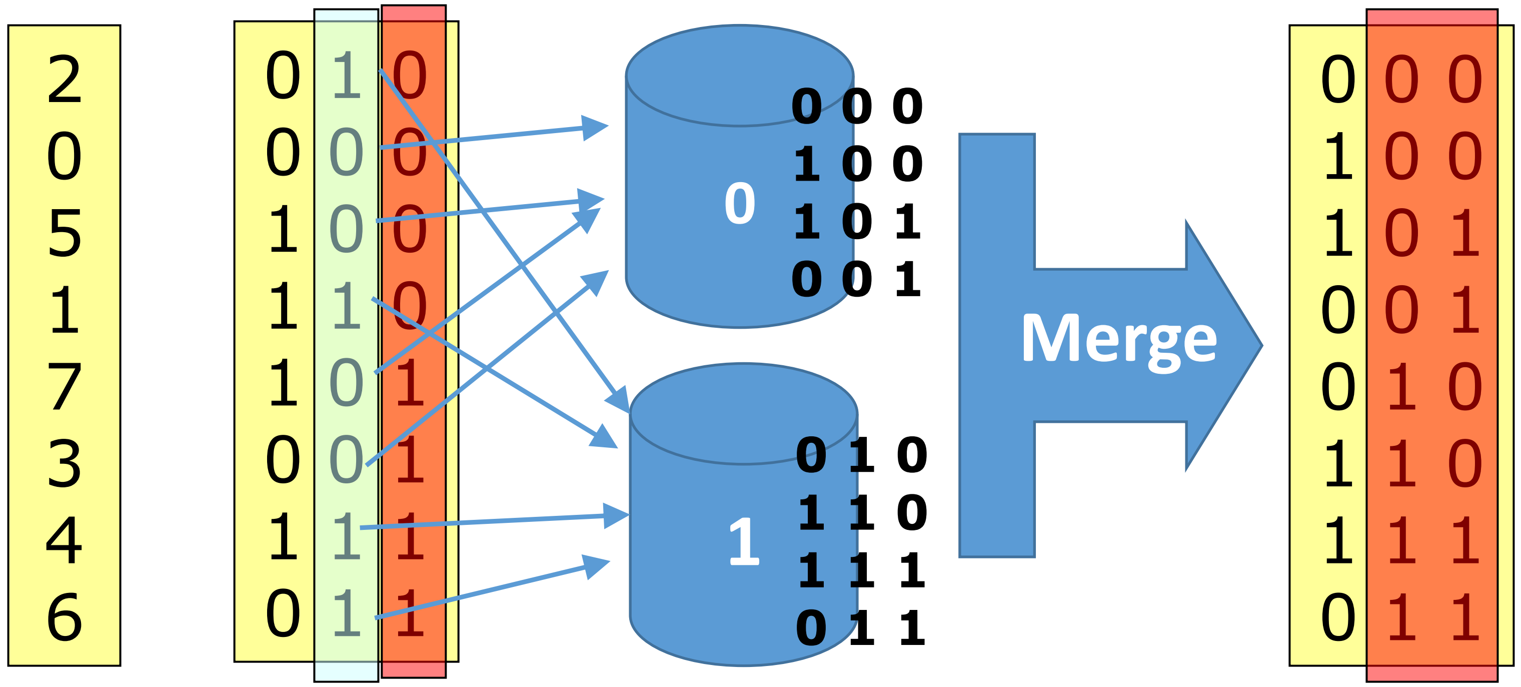
- For N elements between (L, H), using H-L+1 buckets can sort the elements in one round
- Problem: the range (L, H) may be too large.
    - Sorting 4-byte unsigned integers, range is [0, 2^{32}-1] → 2^{32} buckets
- Solution(radix sort): apply bucket sort on every digit/bit

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

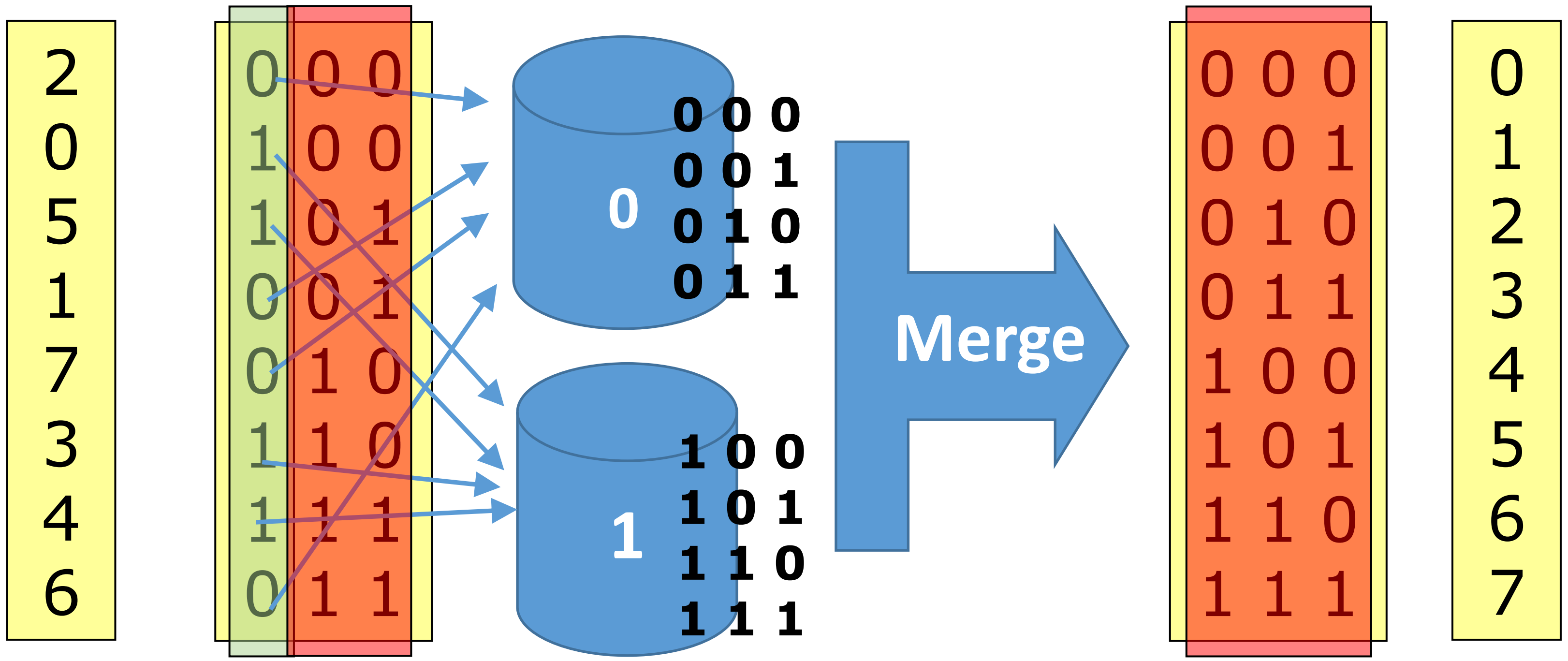
Use two
buckets 0 and 1



Step 1: Sort by the least significant bit



Step 2. Sort by the middle bit

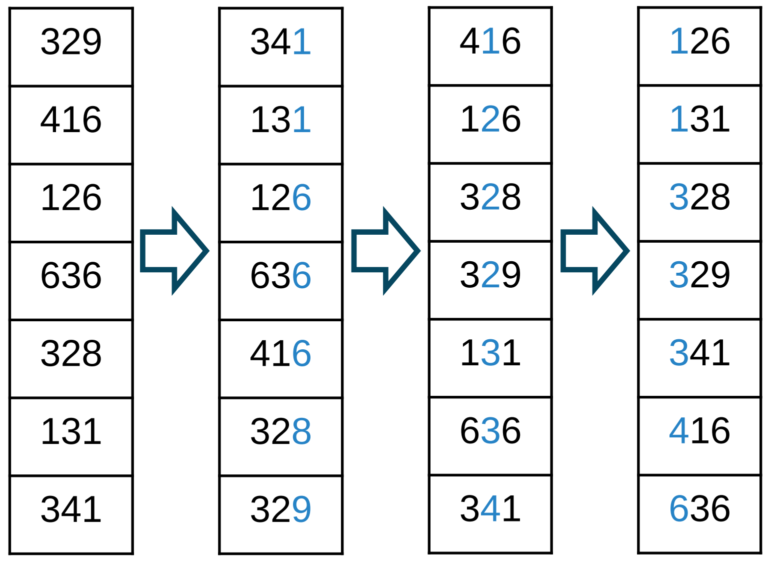


Step 3. Sort by the most significant bit

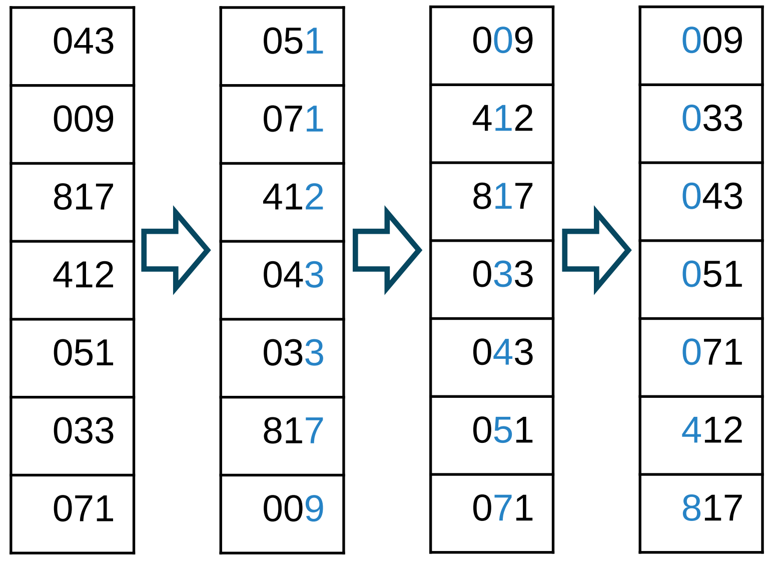
Radix Sort Algorithm Introduction in 5 Minutes, CS Dojo
https://www.youtube.com/watch?v=XiuSW_mEn7g
Radix Sort Animations | Data Structure | Visual How
https://www.youtube.com/watch?v=Om4BljCs_qE

You can choose an appropriate radix value

- Numbers in different formats
  - decimal whole numbers: (126, 328, 636, 341, 416, 131, 328)
  - Binary numbers: (0 001 111 110, 0 101 001 000, 1 001 111 100, 0 101 010 101, 0 110 100 000, 0 010 000 011, 0 101 001 000)
  - Octal numbers: (0176, 0510, 1174, 0525, 0640, 0203, 0510)
  - Hexadecimal numbers: (07E, 148, 27C, 1A0, 083, 148)
- Radix sort of decimal numbers using ten buckets: 0 to 9

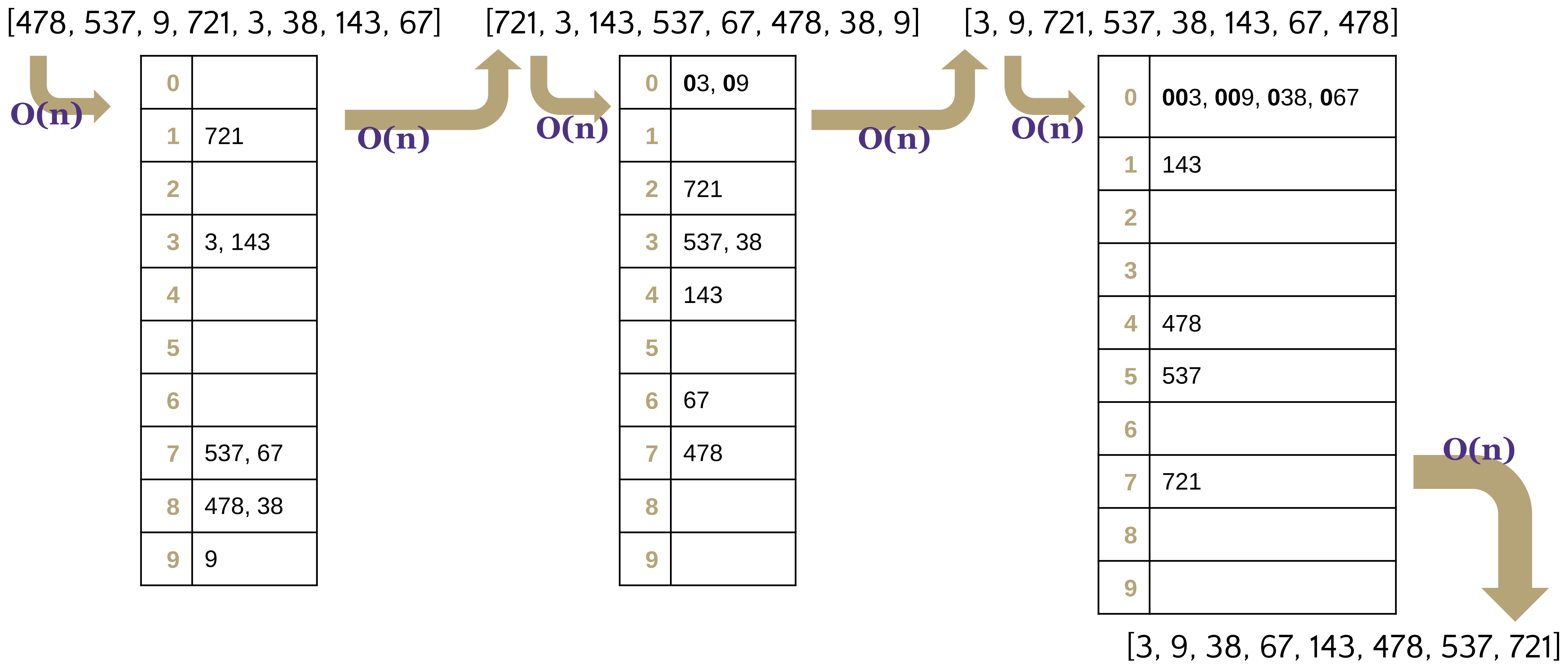


Example 1



Example 2

Radix Sort Example



[478, 537, 9, 721, 3, 38, 143, 67]

0	
1	721
2	
3	3, 143
4	
5	
6	
7	537, 67
8	478, 38
9	9

[721, 3, 143, 537, 67, 478, 38, 9]

0	**0**3, **0**9
1	
2	721
3	537, 38
4	143
5	
6	67
7	478
8	
9	

[3, 9, 721, 537, 38, 143, 67, 478]

0	**00**3, **00**9, **0**38, **0**67
1	143
2	
3	
4	478
5	537
6	
7	721
8	
9	

[3, 9, 38, 67, 143, 478, 537, 721]

Radix Sort Time Complexity

Worst case runtime?	O(n)
Best case runtime?	O(n)
Average runtime?	O(n)
Stable?	Yes
In-place?	No
Useful for:	Sorting ints

Sorting Summary

Sorting: Summary

	Best-Case	Worst-Case	Space	Stable
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$	Yes
Heap Sort	O(nlogn)	O(nlogn)	$O(n)$	No
In-Place Heap Sort	O(nlogn)	O(nlogn)	$O(1)$	No
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn) O(n)* optimized	Yes
Quick Sort	O(nlogn)	$O(n^2)$	$O(n)$	No
In-place Quick Sort	O(nlogn)	$O(n^2)$	$O(1)$	No
Bucket Sort	$O(n)$	$O(n^2)$	$O(K+n)$	Yes
Radix	$O(n)$	$O(n)$	$O(n)$	Yes

No single sorting algorithm is "the best"!

- Different algos have different properties in different situations
- The best one is one that is well-suited to your data

References I

- Insertion Sort | GeeksforGeeks
  - https://www.geeksforgeeks.org/insertion-sort-algorithm/
  - https://www.geeksforgeeks.org/time-and-space-complexity-of-insertion-sort-algorithm/
  - https://www.youtube.com/watch?v=OGzPmgsI-pQ
- Selection Sort | GeeksforGeeks
  - https://www.geeksforgeeks.org/selection-sort-algorithm-2/
  - https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-selection-sort/
  - https://www.youtube.com/watch?v=xWBP4lzkoyM
- Merge Sort Algorithm: A Step-by-Step Visualization, Quoc Dat Phung
  - https://www.youtube.com/watch?v=ho05egqcPl4
- Merge sort in 3 minutes
  - https://www.youtube.com/watch?v=4VqmGXwpLqc
- Merge Sort Algorithm: A Step-by-Step Visualization (recommended)
  - https://www.youtube.com/watch?v=ho05egqcPl4
- Merge Sort Animations | Data Structure | Visual How
  - https://www.youtube.com/watch?v=spVhtO_IcGg

References II

- Quicksort: Partitioning an array, KC Ang
  - https://www.youtube.com/watch?v=MZaf_9IZCrc
- QuickSort | geeksforgeeks
  - https://www.geeksforgeeks.org/quick-sort-algorithm/
  - https://www.youtube.com/watch?v=PgBzjlCcFvc
- Quick sort in 4 minutes (recommended)
  - https://www.youtube.com/watch?v=Hoixgm4-P4M
- Quicksort Algorithm: A Step-by-Step Visualization (recommended)
  - https://www.youtube.com/watch?v=pM-6r5xsNEY
- Visualization of Quick sort (HD)
  - https://www.youtube.com/watch?v=aXXWXz5rF64

References III

- Radix Sort Algorithm Introduction in 5 Minutes, CS Dojo
    - https://www.youtube.com/watch?v=XiuSW_mEn7g
- Radix Sort Animations | Data Structure | Visual How
    - https://www.youtube.com/watch?v=Om4BljCs_qE
- Radix Sort
    - https://www.geeksforgeeks.org/radix-sort/
    - https://www.geeksforgeeks.org/time-and-space-complexity-of-radix-sort-algorithm/
- Bucket Sort | GeeksforGeeks
    - https://www.geeksforgeeks.org/bucket-sort-2/
    - https://www.youtube.com/watch?v=VuXbEb5ywrU
- Time Complexities of all Sorting Algorithms
    - https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

References IV

- Sort Algos // Michael Sambol Michael Sambol
  - https://www.youtube.com/playlist?list=PL9xmBV_5YoZOZSbGAXAPIq1BeUf4j20pl
  - Merge Sort, Quick Sort, Bubble Sort, Insertion Sort, Selection Sort, Heap Sort
- 10 Sorting Algorithms Easily Explained
  - https://www.youtube.com/watch?v=rbbTd-gkajw
  - Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Counting Sort, Shell Sort, Tim Sort, Radix Sort

Full-Length Lectures

- [CSE 373 WI24] Lecture 19: Introduction to Sorting
  - https://www.youtube.com/watch?v=xCezN9A7yhQ&list=PLEcoVsAaONjd5n69K84sSmAuvTrTQT_Nl&index=18
- [CSE 373 WI24] Lecture 20: More Sorting Algorithms
  - https://www.youtube.com/watch?v=9wVXtKko5CM&list=PLEcoVsAaONjd5n69K84sSmAuvTrTQT_Nl&index=19