LEC 23

CSE 373

Sorting I

BEFORE WE START

Instructor: Hunter Schafer

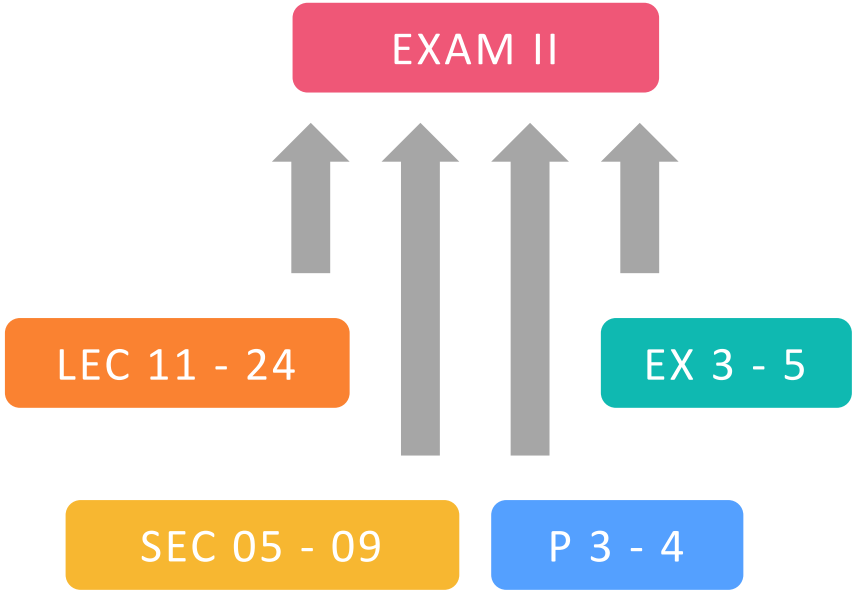
TAs:
Ken Aragon
Khushi Chaudhari
Joyce Elauria
Santino Iannone
Leona Kazi
Nathan Lipiarski
Sam Long
Amanda Park
Paul Pham
Mitchell Szeto
Batina Shikhalieva
Ryan Siu
Elena Spasova
Alex Teng
Blarry Wang
Aileen Zeng

Announcements

- EX4 due this Friday, 12/4 at 11:59 pm
- Last exercise, EX5 (on sorting), will be released that Friday and is due week after
- P4 due next Wednesday 12/09
  - Starting now: ☺! Starting this weekend: ☹!
- Exam 2, during finals week!

Exam II Logistics

- Same logistics as Exam I:
  - 48 hours to complete an exam written for 1-2 hours
  - Open notes & internet, groups up to 6
  - Submit via Gradescope, OH in lecture
- Released Wed 12/16 8:30 AM PST
- Due Fri 12/18 8:30 AM PST
  - **No late submissions!**
- Focuses on second half of the course, up through this Wednesday's lecture (Sorting)
  - But technically "cumulative" in that you *will* need to use skills from the first half (e.g. algorithmic analysis, use List/Stack/Queue/Map, etc.)
- Like Exam I, will emphasize conceptual and "why?" questions. Unlike Exam I, will require you to write short snippets of code!



STUDYING

- Topics list released tonight so you can start looking things over, practice materials published next Monday
- Remember to use the Learning Objectives!

Learning Objectives

After this lecture, you should be able to...

1. Define an ordering relation and stable sort and determine whether a given sorting algorithm is stable
2. Implement Selection Sort and Insertion Sort, compare runtimes and best/worst cases of the two algorithms, and decide when they are appropriate
3. Implement Heap Sort, describe its runtime, and implement the in-place variant

Lecture Outline

- Sorting Definitions
- Insertion & Selection Sort
- Heap Sort

Sorting

- Generally: given items, put them in order

- Why study sorting?
  - Sorting is incredibly common in programming
    - Often a component of other algorithms!
    - Very common in interviews
  - Interesting case study for approaching computational problems
    - We'll use some data structures we've already studied

Types of Sorts

1. Comparison Sorts

Compare two elements at a time.

Works whenever we could implement a compareTo method between elements.

We'll focus on comparison sorts: much more common, and very generalizable!

2. Niche Sorts

Leverage specific properties of data or problem to sort without directly comparing elements.

E.g. if you already know you'll only be sorting numbers < 5, make 5 buckets and add directly

Bonus topic beyond the scope of the class

Sorting: Definitions (Knuth's TAOCP)

- An **ordering relation** < for keys a, b, and c has the following properties:
  - Law of Trichotomy: Exactly one of a < b, a = b, b < a is true
  - Law of Transitivity: If a < b, and b < c, then a < c

- A **sort** is a permutation (re-arrangement) of a sequence of elements that puts the keys into non-decreasing order, relative to the ordering relation
  - $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_N$

```
int temperature
```

- Built-in, simple ordering relation

```
class Movie {
  String name;
  int year;
}
```

- More complex: Whenever we sort, *we also must decide* what ordering relation to use for that application
  - Sort by name?
  - Sort by year?
  - Some combination of both?

Sorting: Stability

- A sort is **stable** if the relative order of *equivalent* keys is maintained after sorting

INPUT

Anita *2010*	Basia *2018*	Caris *2019*	Duska *2020*	Duska *2015*	Anita *2016*

Stable sort using name as key

Anita *2010*	Anita *2016*	Basia *2018*	Caris *2019*	Duska *2020*	Duska *2015*

Unstable sort using name as key

Anita *2016*	Anita *2010*	Basia *2018*	Caris *2019*	Duska *2015*	Duska *2020*

- Stability and Equivalency only matter for complex types
  - i.e. when there is more data than just the key

Anita	Basia	Anita	Duska	Esteban	Duska	Caris

Anita	Anita	Basia	Caris	Duska	Duska	Esteban

Sorting: Performance Definitions

- Runtime performance is sometimes called the **time complexity**
  - Example: Dijkstra's has time complexity O(E log V).

- <u>Extra</u> memory usage is sometimes called the **space complexity**
  - Dijkstra's has space complexity Θ(V)
    - Priority Queue, distTo and edgeTo maps
  - The input graph takes up space Θ(V+E), but we don't count this as part of the space complexity since the graph itself already exists and is an input to Dijkstra's

Lecture Outline

- Sorting Definitions
- **Insertion & Selection Sort**
- Heap Sort

Sorting Strategy 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.

INVARIANT

Iterative Improvement

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

Selection Sort

Every iteration, **select** the smallest unsorted item to fill the next spot.

https://www.youtube.com/watch?v=Ns4TPTC8whw

0	1	2	3	4	5	6	7	8	9
2	3	6	7	18	10	14	9	11	15

Sorted Items — Current Item — Unsorted Items

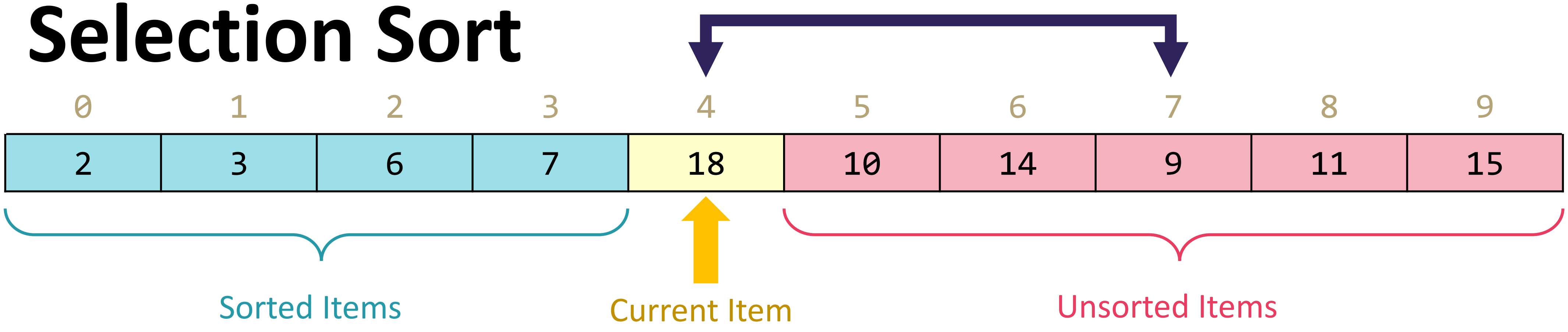
0	1	2	3	4	5	6	7	8	9
2	3	6	7	9	10	14	18	11	15

Sorted Items — Current Item — Unsorted Items

0	1	2	3	4	5	6	7	8	9
2	3	6	7	9	10	14	18	11	15

Sorted Items — Current Item — Unsorted Items

Selection Sort




```
void selectionSort(list) {
  for each current in list:
    target = findNextMin(current)
    swap(target, current)
}
int findNextMin(current) {
  min = current
  for each item in unsorted items:
    if (item < min):
      min = current
  return min
}
int swap(target, current) {
  temp = current
  current = target
  target = temp
}
```

Worst case runtime? $\Theta(n^2)$

Best case runtime? $\Theta(n^2)$

In-practice runtime? $\Theta(n^2)$

Stable? No

In-place? Yes

Selection Sort Stability

0	1	2	3	4	5	6
5a	3	4	5b	2	6	8

0	1	2	3	4	5	6
2	3	4	5b	5a	6	8

…

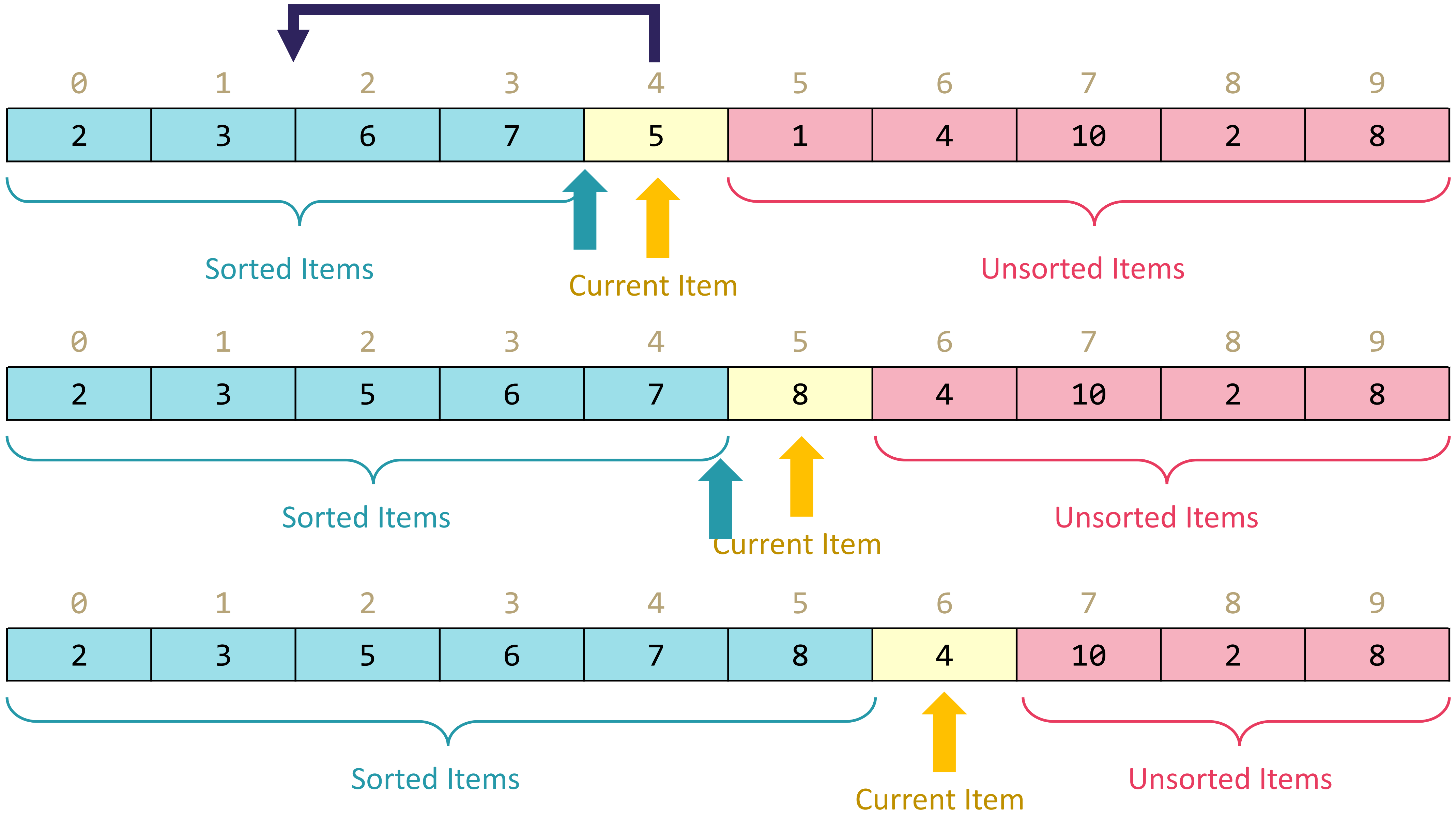
0	1	2	3	4	5	6
2	3	4	5b	5a	6	8

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

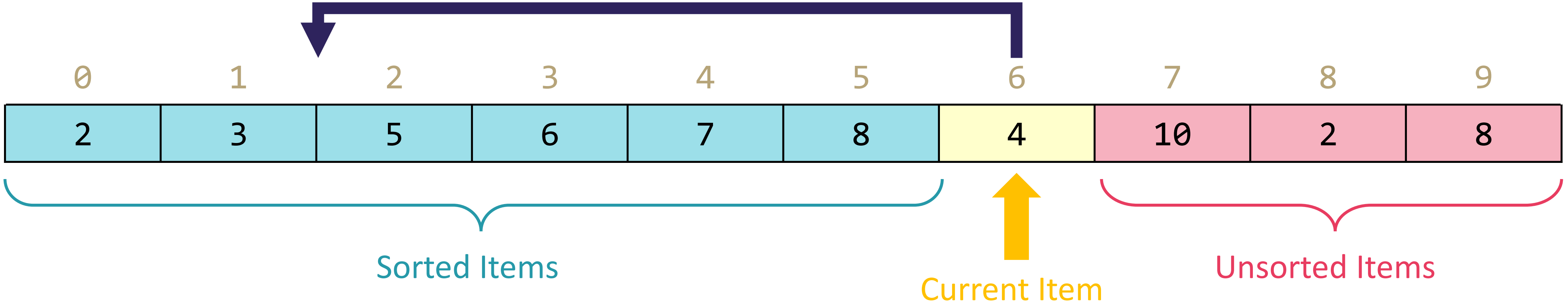
Insertion Sort

Every iteration, **insert** the next unsorted item into the sorted items

https://www.youtube.com/watch?v=ROalU379l3U



Insertion Sort




```
void insertionSort(list) {
  for each current in list:
    target = findSpot(current)
    shift(target, current)
}
int findSpot(current) {
  for each spot in sorted items going backwards:
    if (current goes in spot):
      return spot
}
void shift(target, current) {
  for (i = current; i > target; i--):
    item[i+1] = item[i]
  item[target] = current
}
```

Worst case runtime? $\Theta(n^2)$

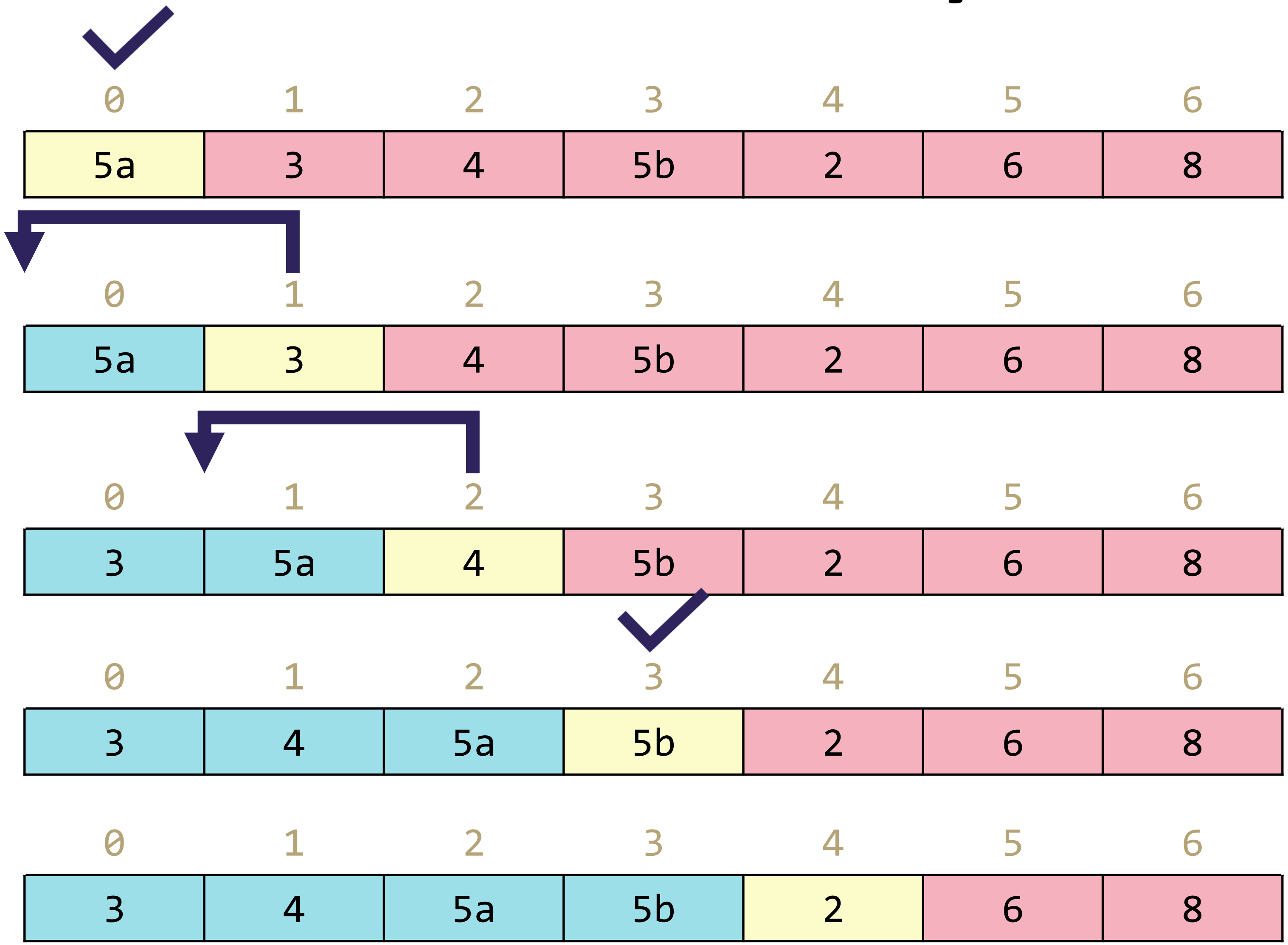
Best case runtime? $\Theta(n)$

In-practice runtime? $\Theta(n^2)$

Stable? Yes

In-place? Yes

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
- Duplicates will always be compared against one another **in their original orientation**, so can maintain stability with proper if logic

Selection vs. Insertion Sort

```
void selectionSort(list) {
   for each current in list:
      target = findNextMin(current)
      swap(target, current)
}
```

"Look through unsorted to **select** the smallest item to replace the current item"

- Then **swap** the two elements

Worst case runtime? $\Theta(n^2)$
Best case runtime? $\Theta(n^2)$
In-practice runtime? $\Theta(n^2)$
Stable? No
In-place? Yes

Minimizes writing to an array (doesn't have to shift everything)

```
void insertionSort(list) {
   for each current in list:
      target = findSpot(current)
      shift(target, current)
}
```

"Look through sorted to **insert** the current item in the spot where it belongs"

- Then **shift** everything over to make space

Worst case runtime? $\Theta(n^2)$
Best case runtime? $\Theta(n)$
In-practice runtime? $\Theta(n^2)$
Stable? Yes
In-place? Yes

Almost always preferred: Stable & can take advantage of an already-sorted list. (LinkedList means no shifting ☺, though doesn't change asymptotic runtime)

0	1	2	3	4	5	6	7	8	9
2	3	5	6	7	8	4	10	2	8

Sorted Items

Current Item

Unsorted Items

Lecture Outline

- Sorting Definitions
- Insertion & Selection Sort
- **Heap Sort**

Sorting Strategy 2: Impose Structure

- Consider what contributes to Selection sort runtime of $\Theta(n^2)$
  - Unavoidable n iterations to consider each element
  - Finding next minimum element to swap requires a $\Theta(n)$ linear scan! Could we do better?

$\Theta(n)$ iterations → for each current in list:

$\Theta(n)$ → target = findNextMin(current)

```
void selectionSort(list) {
   for each current in list:
      target = findNextMin(current)
      swap(target, current)
}
int findNextMin(current) {
   min = current
   for each item in unsorted items:
      if (item < min):
         min = current
   return min
}
```

- If only we knew a way to *structure* our *data* to make it fast to find the smallest item remaining in our dataset...

MIN PRIORITY QUEUE ADT

Heap Sort

https://www.youtube.com/watch?v=Xw2D9aJRBY4

1. run Floyd's buildHeap on your data
2. call removeMin n times to pull out every element!

```
void heapSort(list) {
   E[] heap = buildHeap(list)
   E[] output = new E[n]
   for (i = 0; i < n; i++):
      output[i] = removeMin(heap)
}
```

Worst case runtime?	$\Theta(n \log n)$
Best case runtime?	$\Theta(n)$
In-practice runtime?	$\Theta(n \log n)$
Stable?	No
In-place?	If we get clever…

In-Place Heap Sort

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

Heap

Sorted Items

Current Item

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

percolateDown(22)

Heap

Sorted Items

Current Item

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

Heap

Sorted Items

Current Item

In Place Heap Sort

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

Current Item

Heap

Sorted Items

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

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? $\Theta(n \log n)$

Best case runtime? $\Theta(n)$

In-practice runtime? $\Theta(n \log n)$

Stable? No

In-place? Yes