Lecture 30 (Sorting 2)

Insertion Sort and Quicksort

CS61B, Spring 2025 @ UC Berkeley

Slides credit: Josh Hug

Naive Insertion Sort

Lecture 30, CS61B, Spring 2025

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output: Demo (Link)

Input:

32	15	2	17	19	26	41	17	17

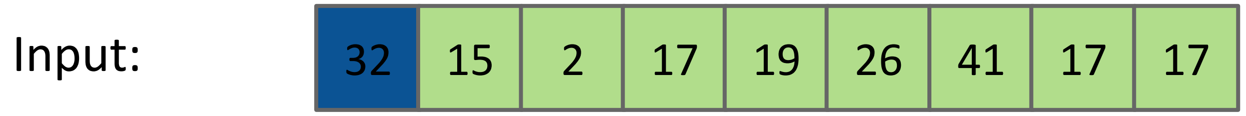
Output:

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



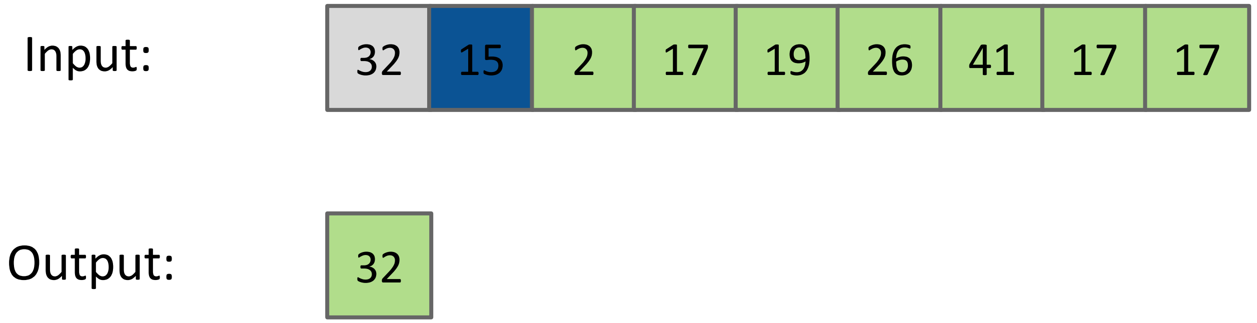
Output:

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:

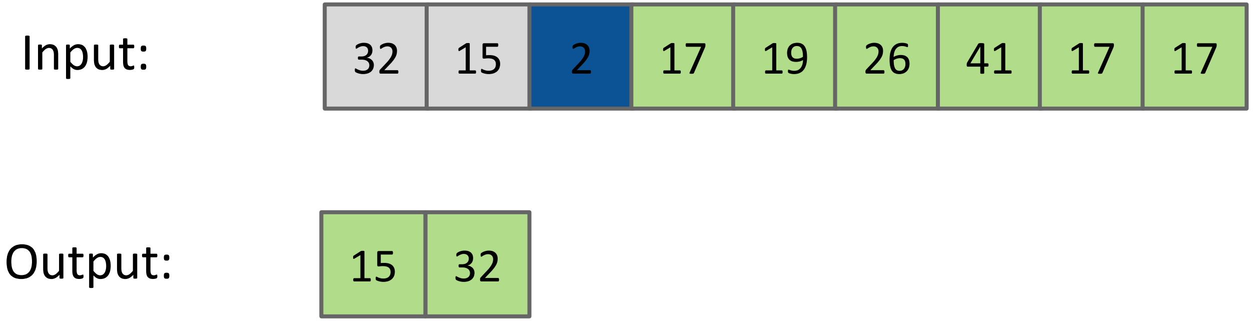


Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:

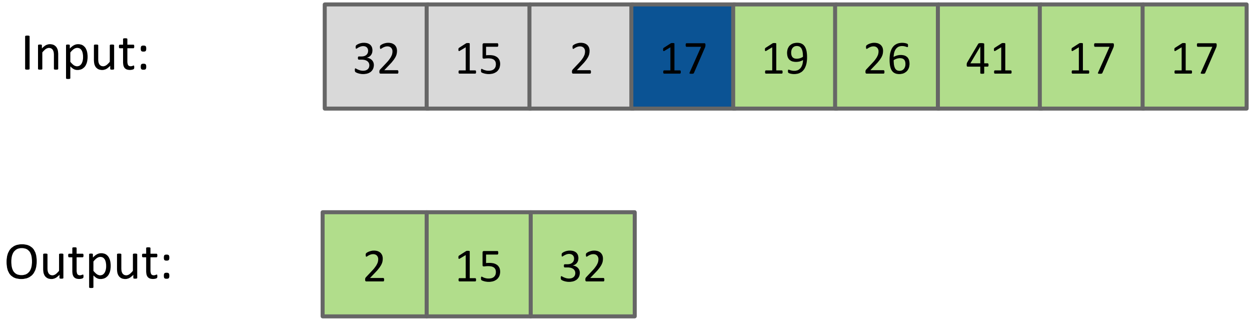


Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:

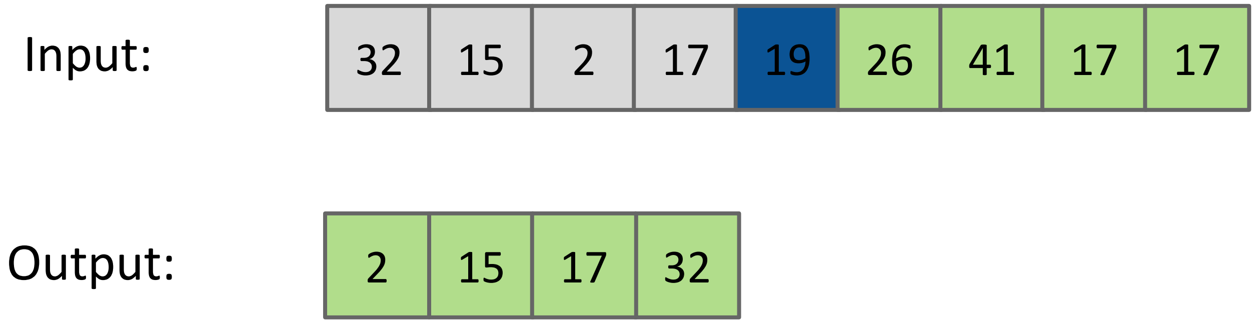


Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:

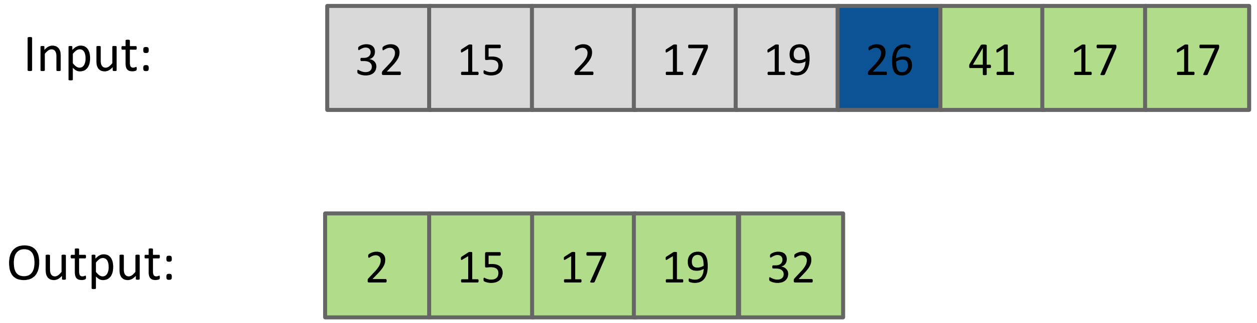


Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:

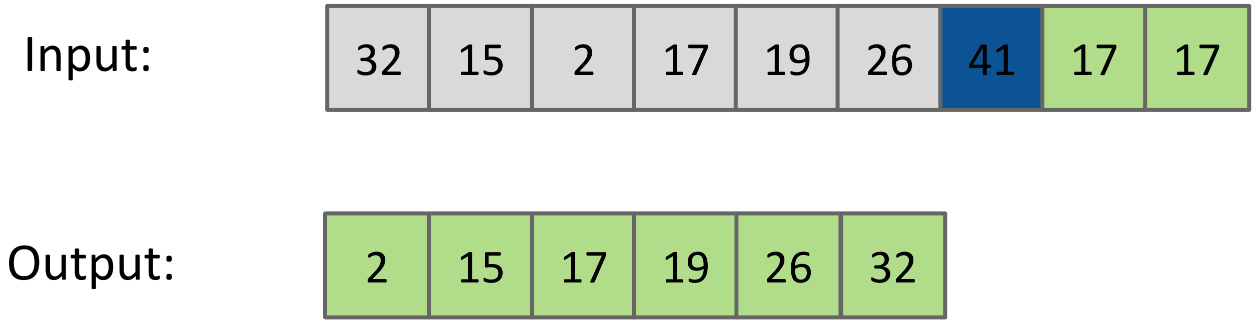


Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:

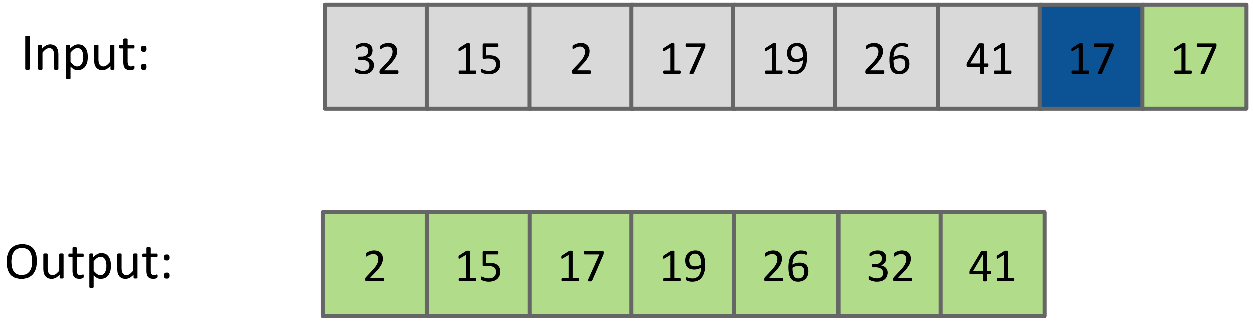


Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:

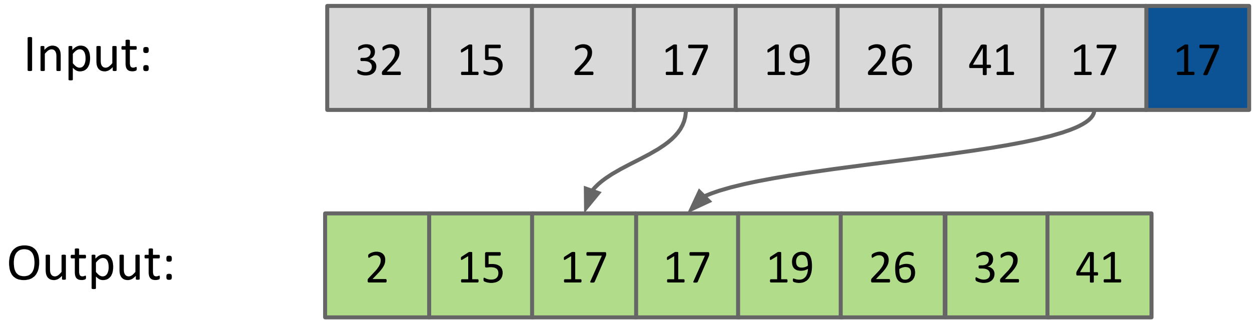


Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:

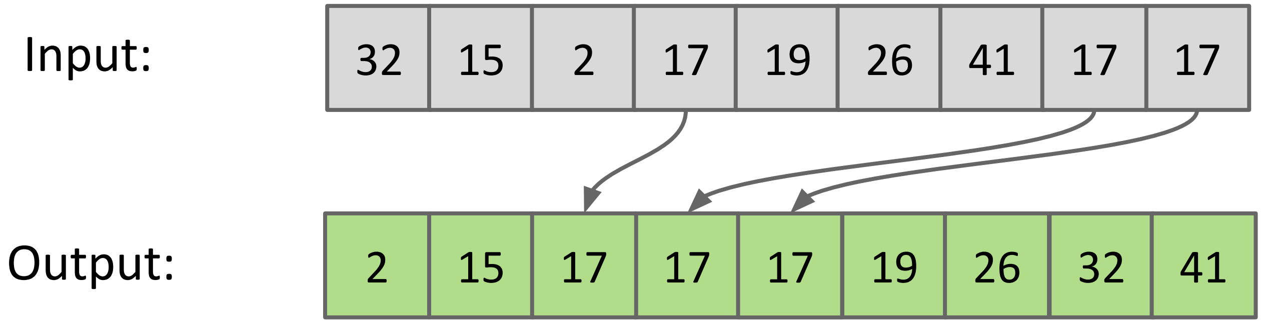


Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



In-Place Insertion Sort

Lecture 30, CS61B, Spring 2025

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

For naive approach, if output sequence contains k items, worst cost to insert a single item is k.

- Might need to move everything over.

More efficient method:

- Do everything in place using swapping.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - Swap item backwards until traveller is in the right place among all previously examined items.

Input:

32	15	2	17	19	26	41	17	17

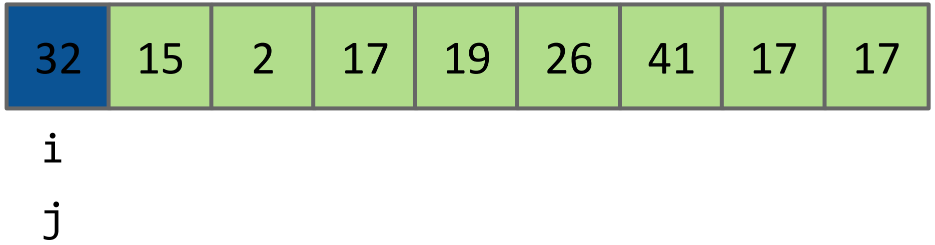
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - **Designate item i as the traveling item.**
    - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



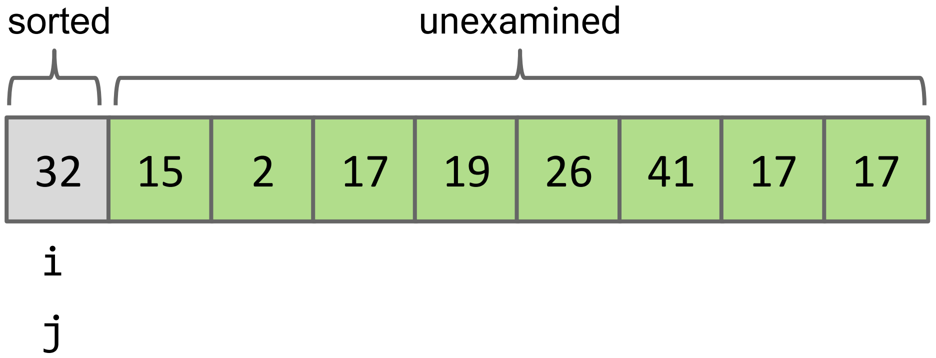
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



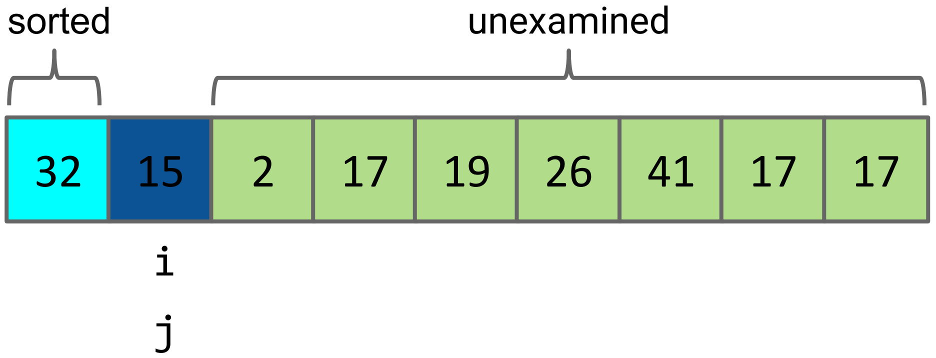
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



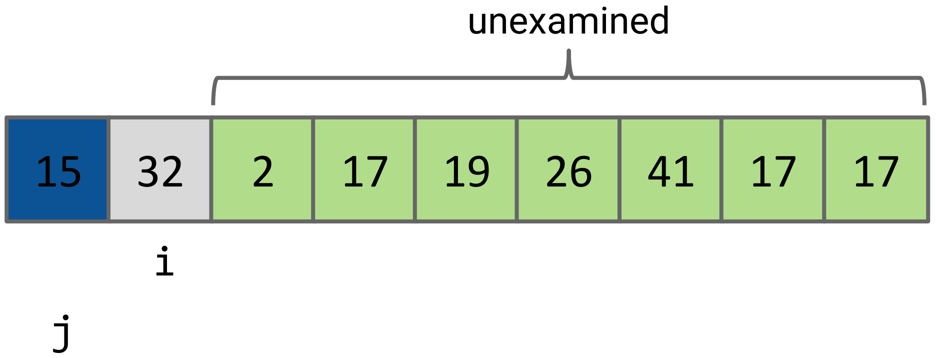
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:

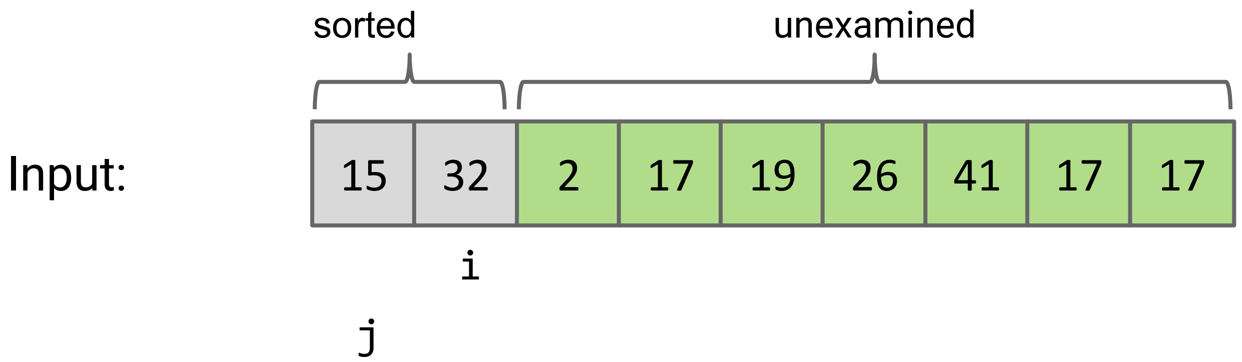


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**



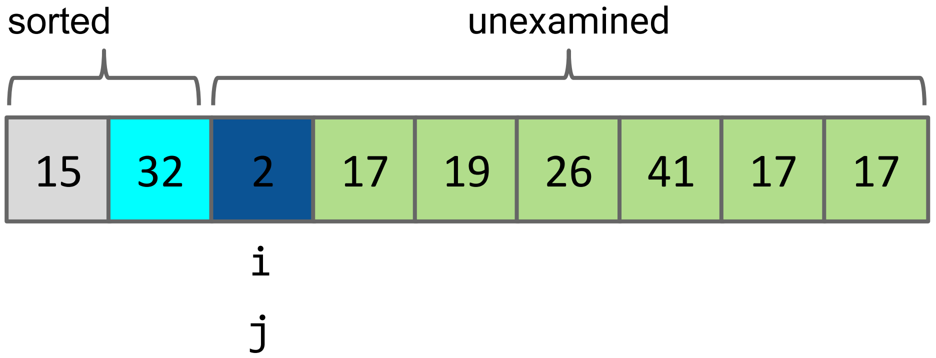
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - **Designate item i as the traveling item.**
    - Swap item backwards until traveller is in the right place among all previously examined items.

Input:

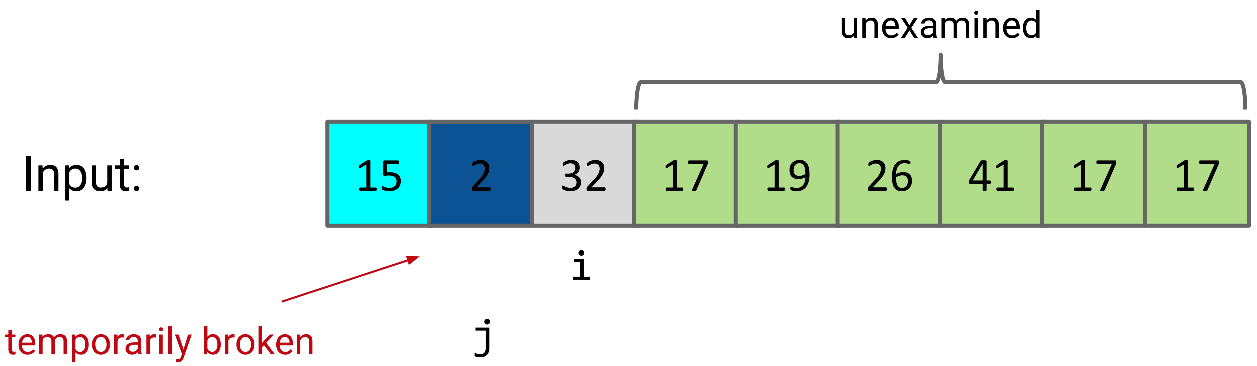


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**



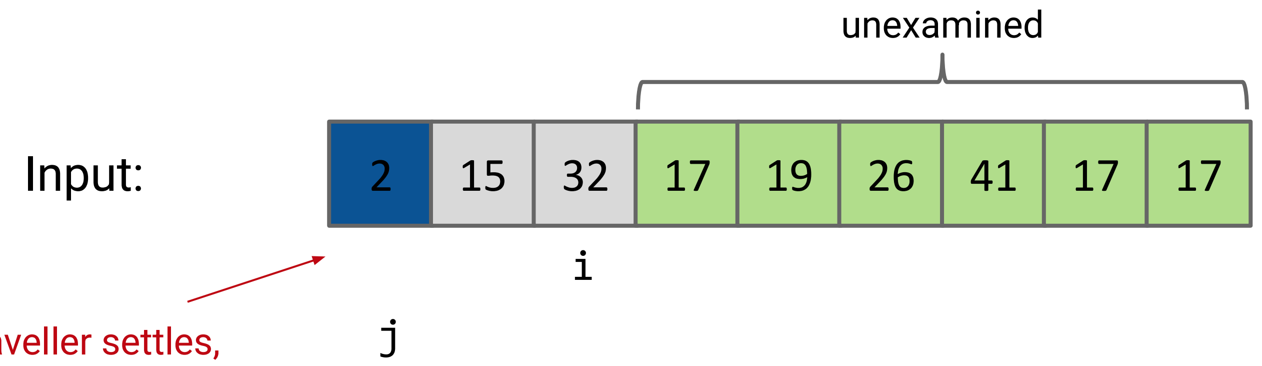
Note: We've temporarily broken our invariant that the items up through item i should be sorted!

In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**



Once the traveller settles, the invariant is restored.

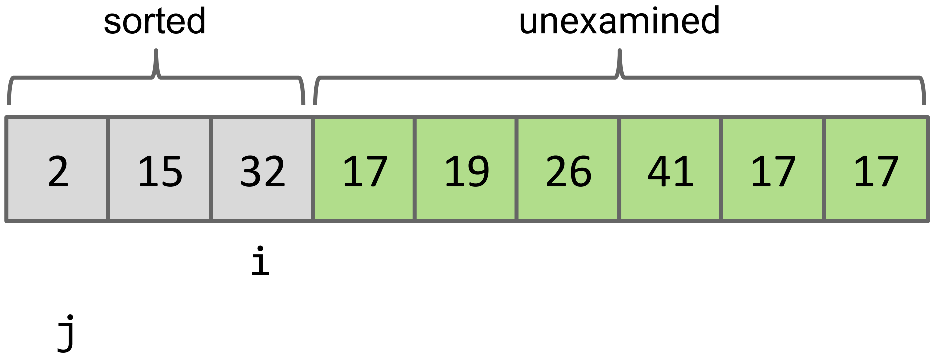
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



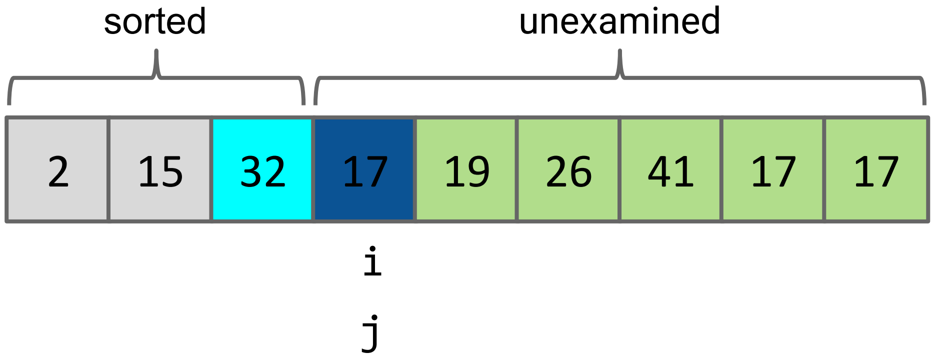
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.

Input:

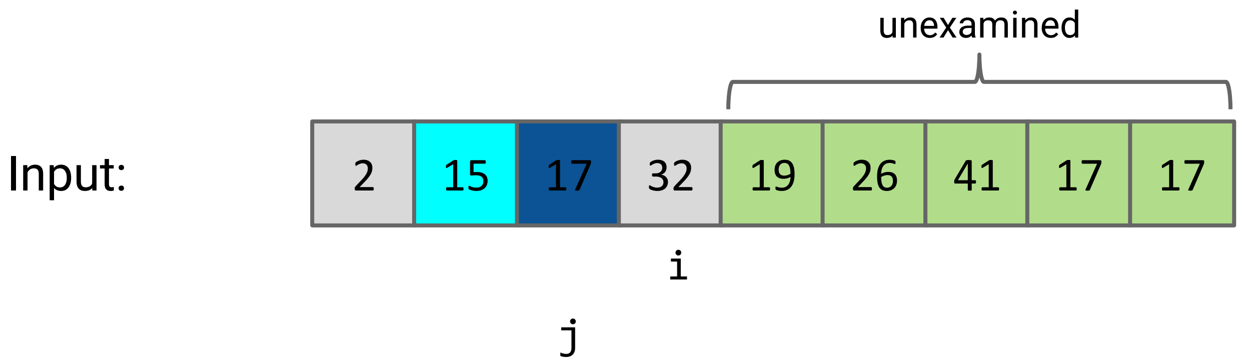


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**



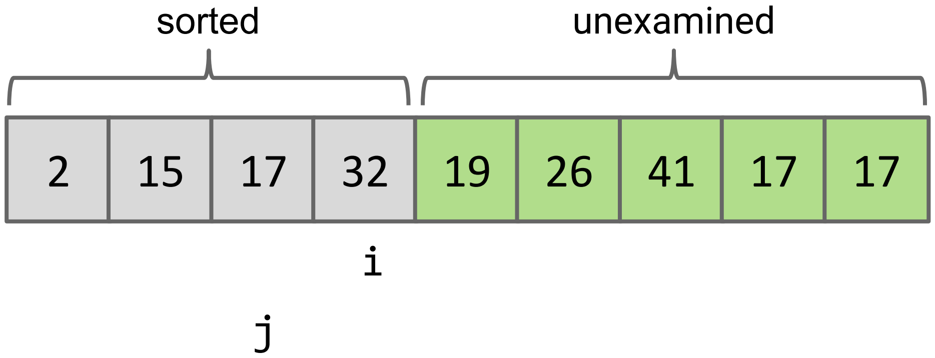
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:

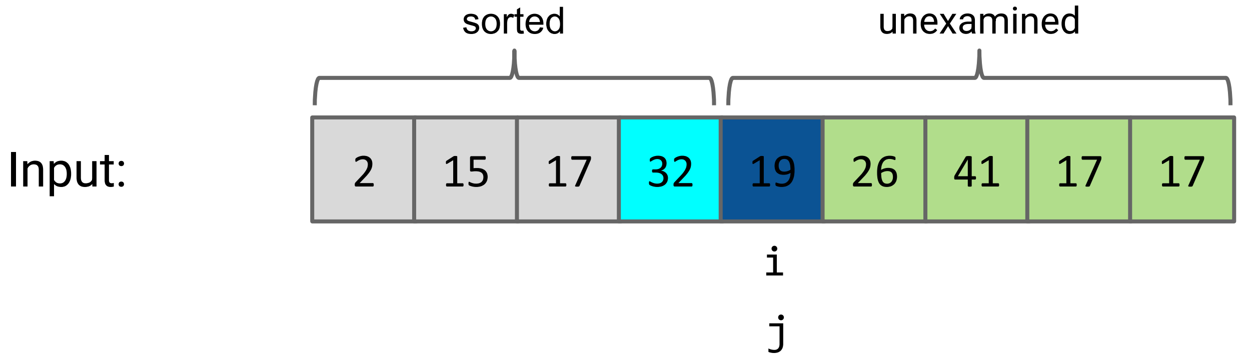


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.



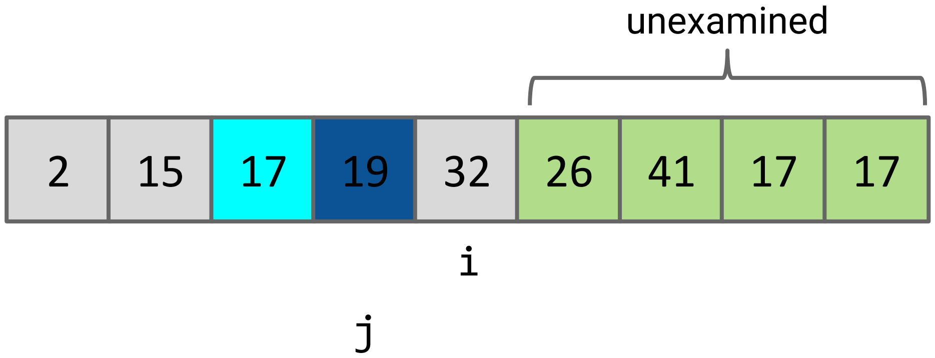
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



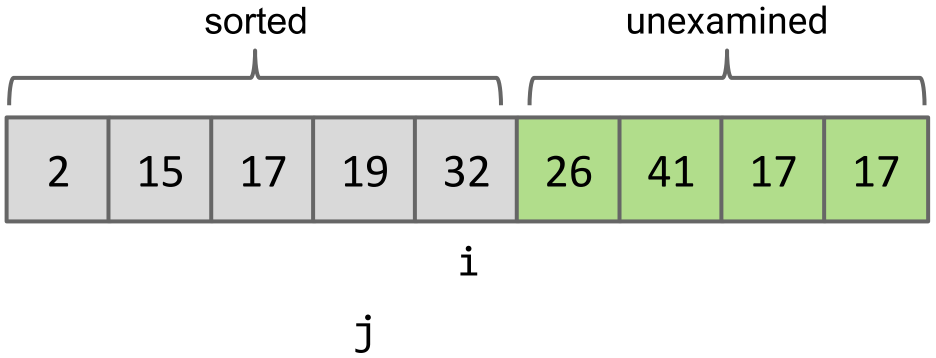
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



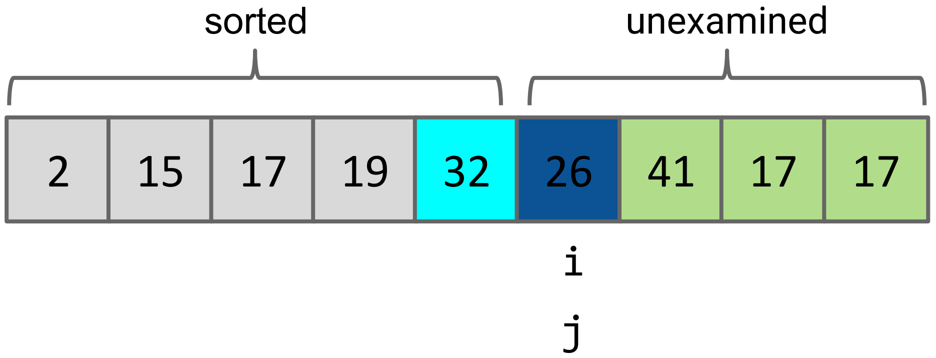
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - **Designate item i as the traveling item.**
    - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



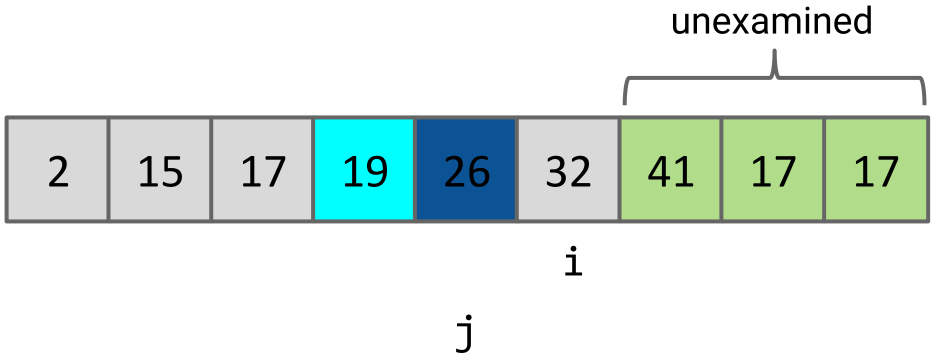
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



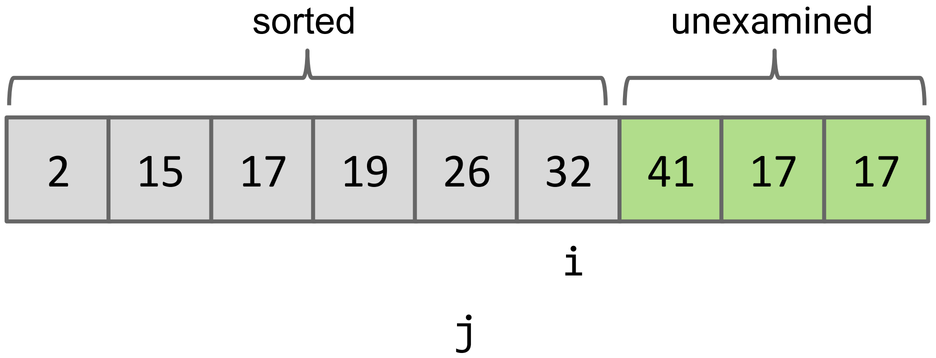
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



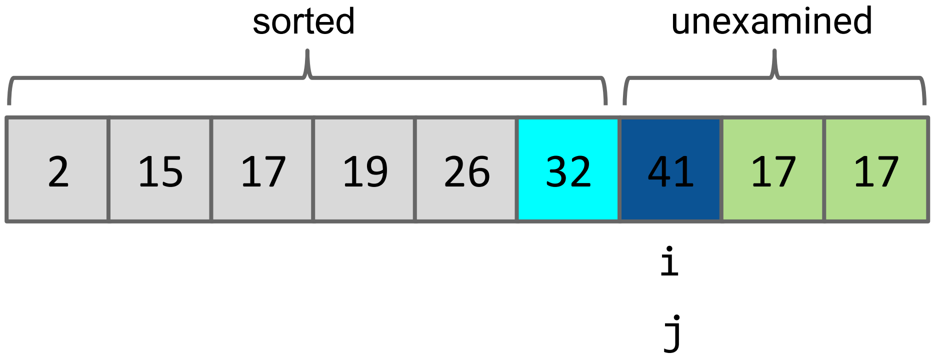
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.

Input:

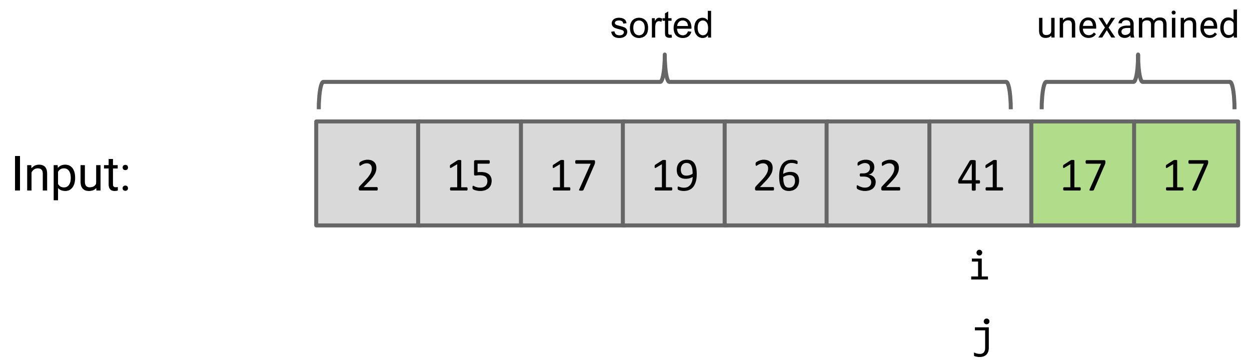


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**



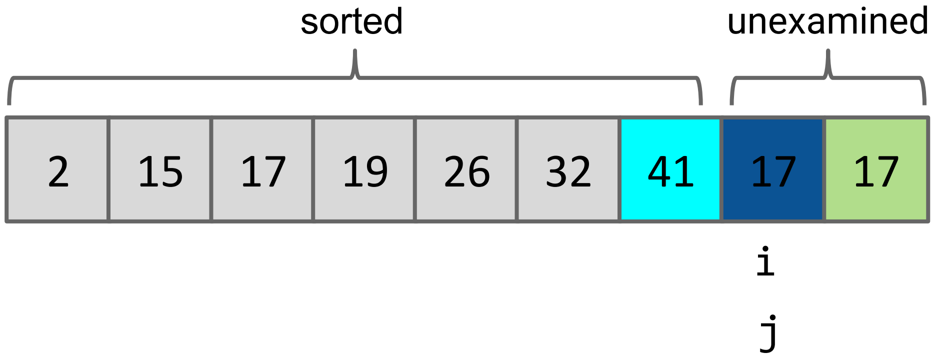
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - **Designate item i as the traveling item.**
    - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



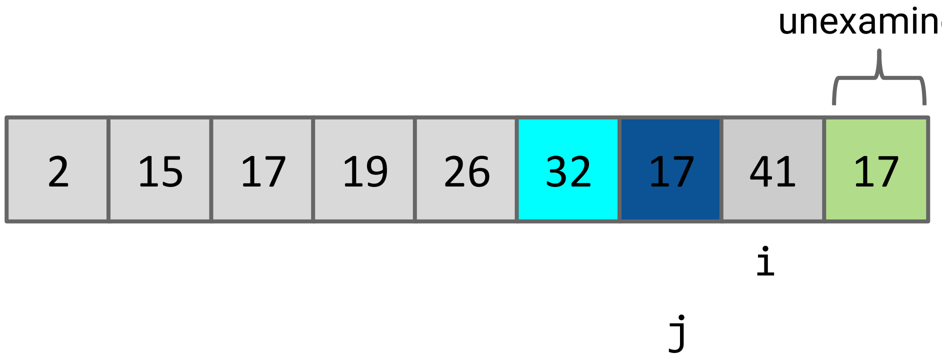
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:

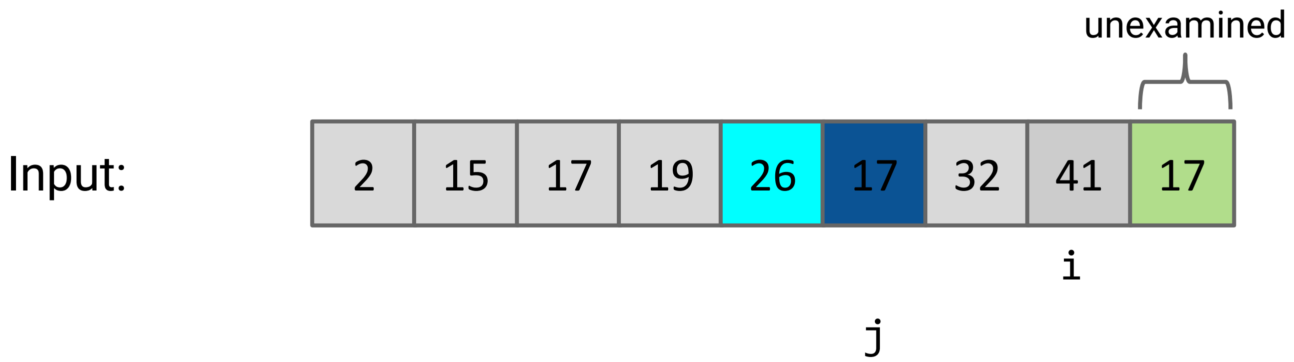


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

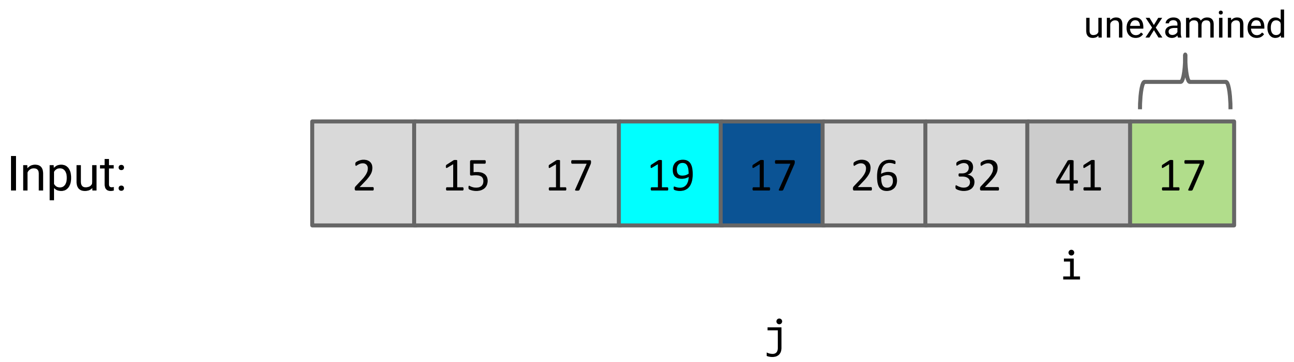


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**



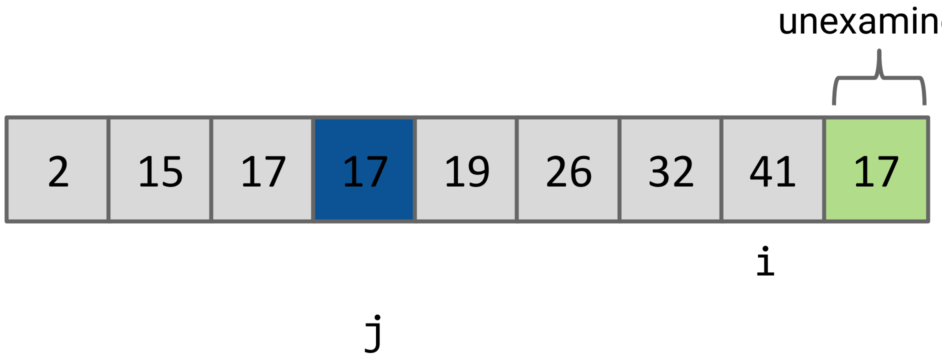
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



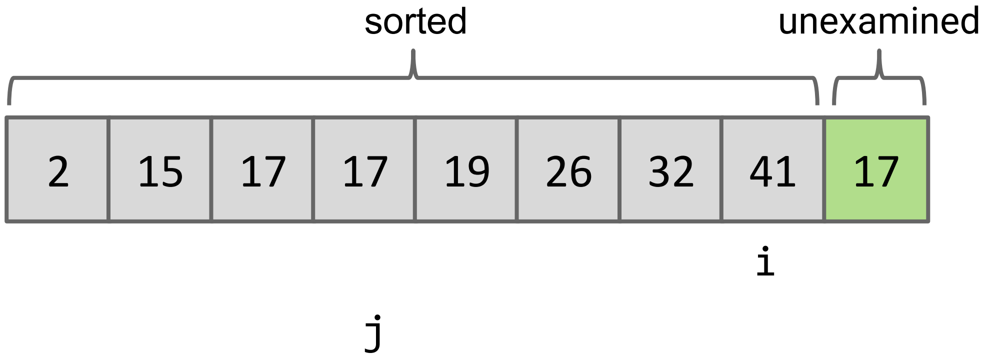
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



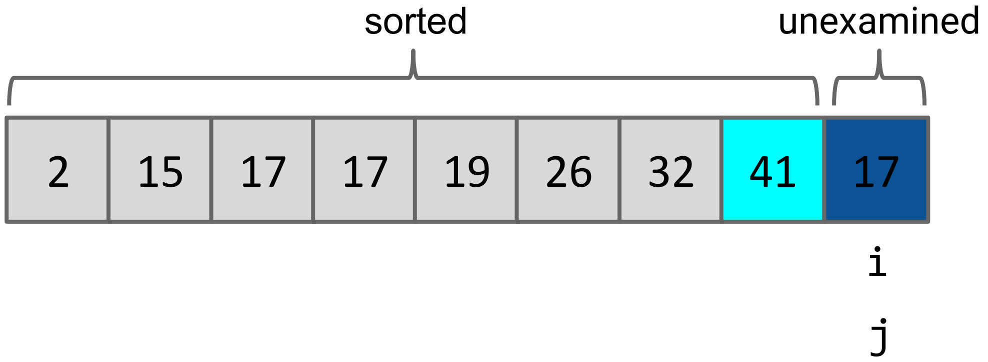
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - **Designate item i as the traveling item.**
    - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



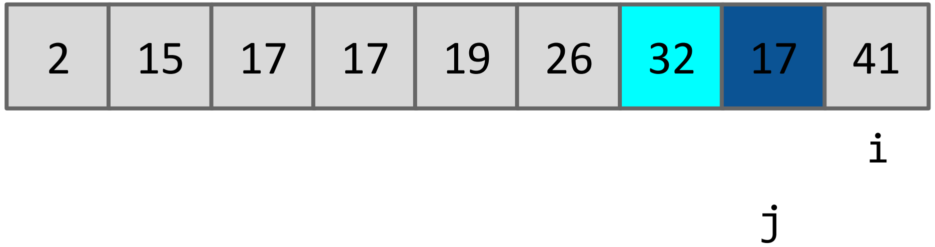
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



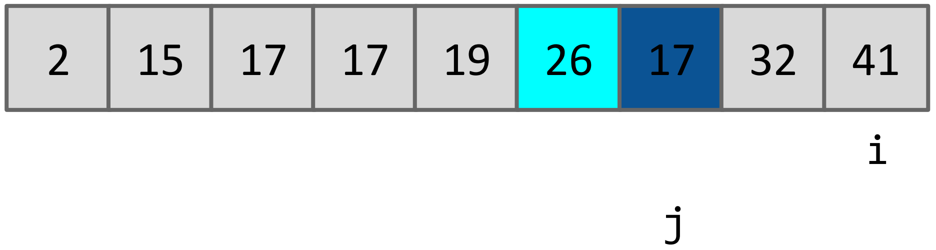
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



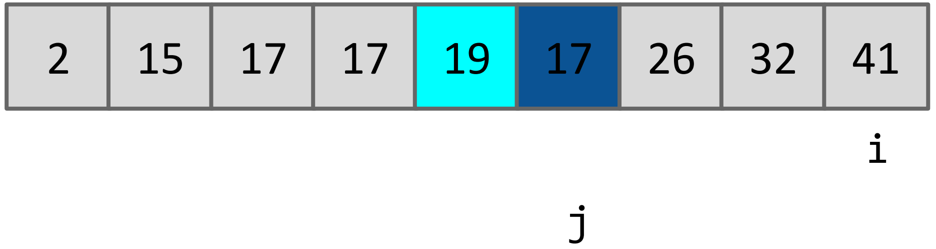
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



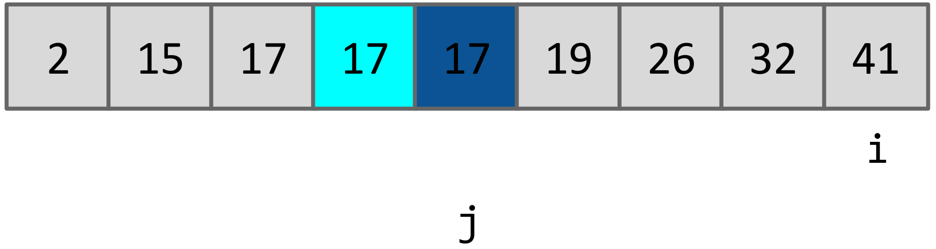
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



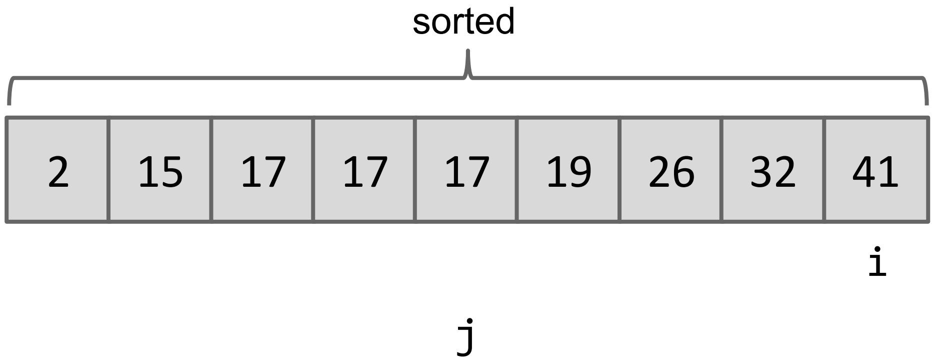
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



In example above: Use j pointer to track current spot of traveling item.

Insertion Sort Runtime

Lecture 30, CS61B, Spring 2025

In-place Insertion Sort

Two more examples.

7 swaps:

```
P O T A T O
P O T A T O   (0 swaps)
O P T A T O   (1 swap )
O P T A T O   (0 swaps)
A O P T T O   (3 swaps)
A O P T T O   (0 swaps)
A O O P T T   (3 swaps)
```

36 swaps:

```
S O R T E X A M P L E
S O R T E X A M P L E   (0 swaps)
O S R T E X A M P L E   (1 swap )
O R S T E X A M P L E   (1 swap )
O R S T E X A M P L E   (0 swaps)
E O R S T X A M P L E   (4 swaps)
E O R S T X A M P L E   (0 swaps)
A E O R S T X M P L E   (6 swaps)
A E M O R S T X P L E   (5 swaps)
A E M O P R S T X L E   (4 swaps)
A E L M O P R S T X E   (7 swaps)
A E E L M O P R S T X   (8 swaps)
```

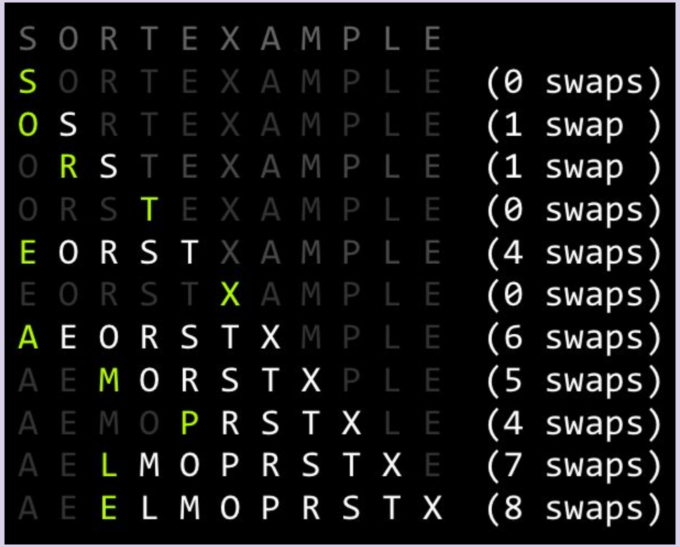
Purple: Element that we're moving left (with swaps).
Black: Elements that got swapped with purple.
Grey: Not considered this iteration.

Insertion Sort Runtime: yellkey.com/TODO

What is the runtime of insertion sort?

A. $\Omega(1)$, $O(N)$
B. $\Omega(N)$, $O(N)$
C. $\Omega(1)$, $O(N^2)$
D. $\Omega(N)$, $O(N^2)$
E. $\Omega(N^2)$, $O(N^2)$

36 swaps:



```
S O R T E X A M P L E
S O R T E X A M P L E   (0 swaps)
O S R T E X A M P L E   (1 swap )
O R S T E X A M P L E   (1 swap )
O R S T E X A M P L E   (0 swaps)
E O R S T X A M P L E   (4 swaps)
E O R S T X A M P L E   (0 swaps)
A E O R S T X M P L E   (6 swaps)
A E M O R S T X P L E   (5 swaps)
A E M O P R S T X L E   (4 swaps)
A E L M O P R S T X E   (7 swaps)
A E E L M O P R S T X   (8 swaps)
```

Insertion Sort Runtime

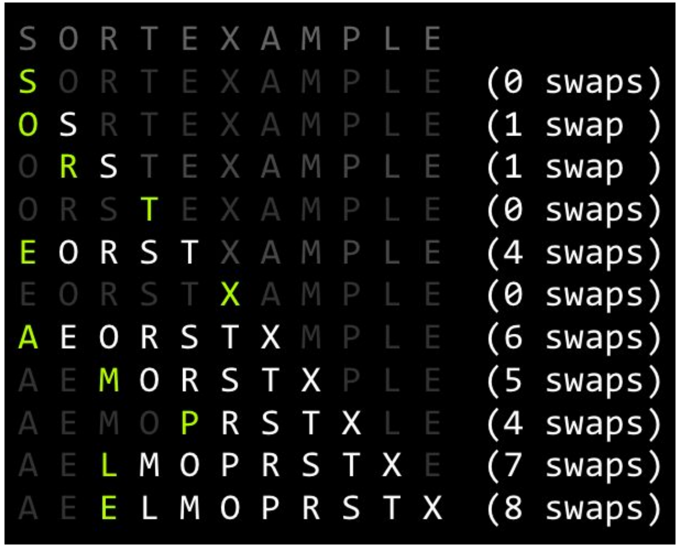
What is the runtime of insertion sort?

A. $\Omega(1)$, $O(N)$
B. $\Omega(N)$, $O(N)$
C. $\Omega(1)$, $O(N^2)$
D. $\Omega(N)$, $O(N^2)$
E. $\Omega(N^2)$, $O(N^2)$

You may recall Ω is not "best case".

So technnnnniically you could also say $\Omega(1)$

36 swaps:



```
S O R T E X A M P L E
S O R T E X A M P L E   (0 swaps)
O S R T E X A M P L E   (1 swap )
O R S T E X A M P L E   (1 swap )
O R S T E X A M P L E   (0 swaps)
E O R S T X A M P L E   (4 swaps)
E O R S T X A M P L E   (0 swaps)
A E O R S T X M P L E   (6 swaps)
A E M O R S T X P L E   (5 swaps)
A E M O P R S T X L E   (4 swaps)
A E L M O P R S T X E   (7 swaps)
A E E L M O P R S T X   (8 swaps)
```

Picking the Best Sort: yellkey.com/TODO

Suppose we do the following:

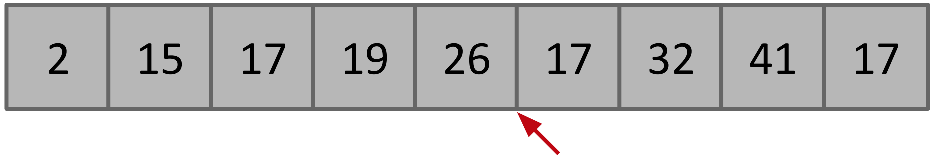
- Read 1,000,000 integers from a file into an array of length 1,000,000.
- Mergesort these integers.
- Select one integer randomly and change it.
- Sort using algorithm X of your choice.

Which sorting algorithm would be the fastest choice for X?

A. Selection Sort: $O(N^2)$
B. Heapsort: O(N Log N)
C. Mergesort: O(N Log N)
D. Insertion Sort: $O(N^2)$

Recall: Inversions

An ***inversion*** is a pair of elements that are out of order with respect to <.



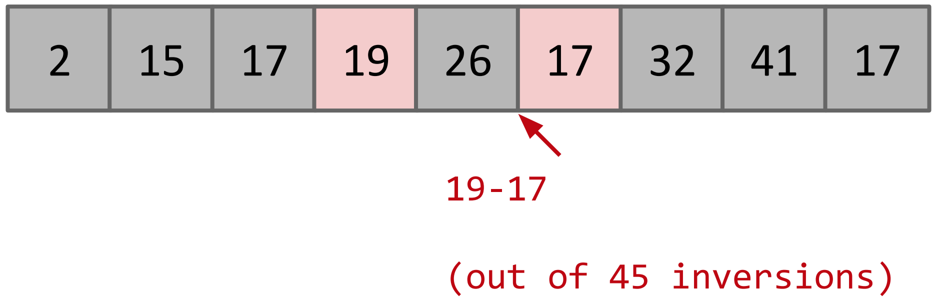
(9 choose 2) = 45
inversions at most

Another way to state the goal of sorting:

- Given a sequence of elements with Z inversions.
- Perform a sequence of operations that reduces inversions to 0.

Recall: Inversions

An ***inversion*** is a pair of elements that are out of order with respect to <.

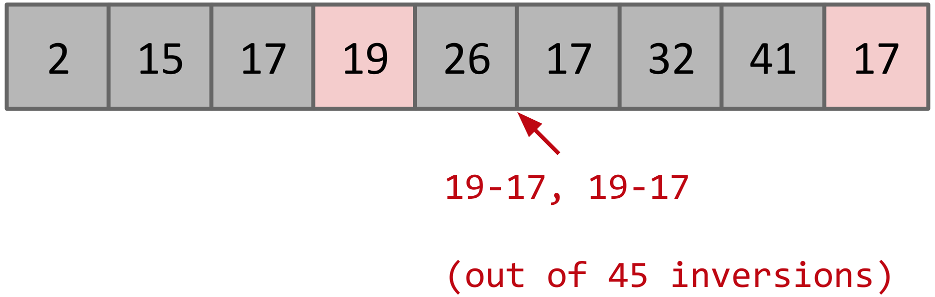


Another way to state the goal of sorting:

- Given a sequence of elements with Z inversions.
- Perform a sequence of operations that reduces inversions to 0.

Recall: Inversions

An ***inversion*** is a pair of elements that are out of order with respect to <.

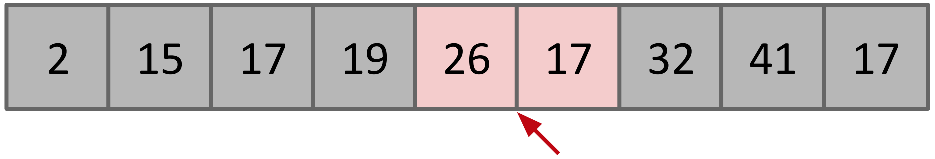


Another way to state the goal of sorting:

- Given a sequence of elements with Z inversions.
- Perform a sequence of operations that reduces inversions to 0.

Recall: Inversions

An ***inversion*** is a pair of elements that are out of order with respect to <.



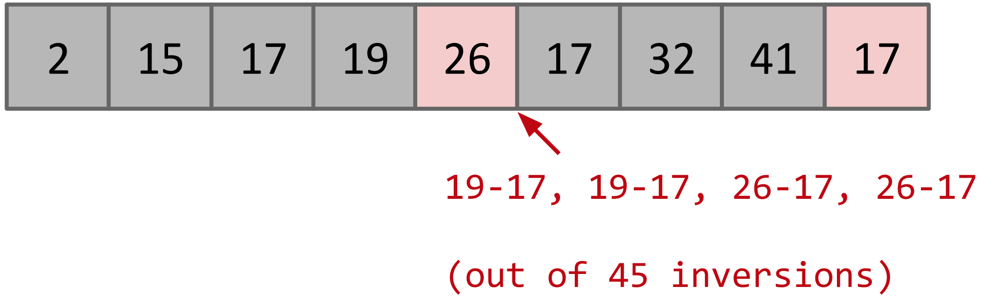
```
19-17, 19-17, 26-17

(out of 45 inversions)
```

Another way to state the goal of sorting:

- Given a sequence of elements with Z inversions.
- Perform a sequence of operations that reduces inversions to 0.

Recall: Inversions

An ***inversion*** is a pair of elements that are out of order with respect to <.

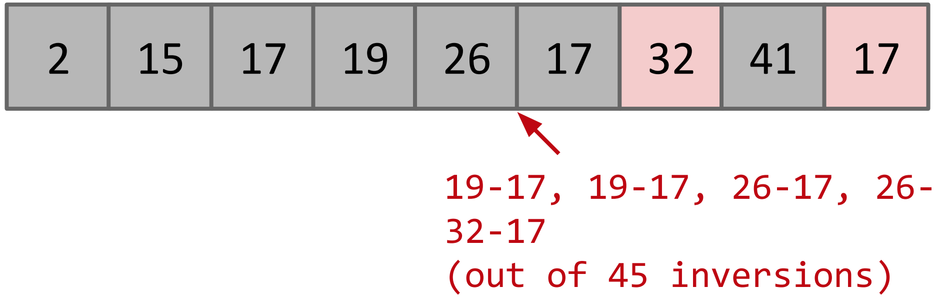


Another way to state the goal of sorting:

- Given a sequence of elements with Z inversions.
- Perform a sequence of operations that reduces inversions to 0.

Recall: Inversions

An ***inversion*** is a pair of elements that are out of order with respect to <.

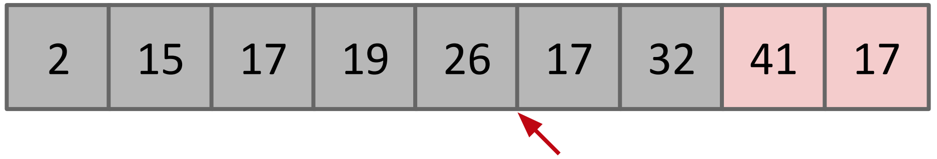


Another way to state the goal of sorting:

- Given a sequence of elements with Z inversions.
- Perform a sequence of operations that reduces inversions to 0.

Recall: Inversions

An ***inversion*** is a pair of elements that are out of order with respect to <.



19-17, 19-17, 26-17, 26-17
32-17, 41-17 = 6 total inversions
(out of 45 inversions)

Another way to state the goal of sorting:

- Given a sequence of elements with Z inversions.
- Perform a sequence of operations that reduces inversions to 0.

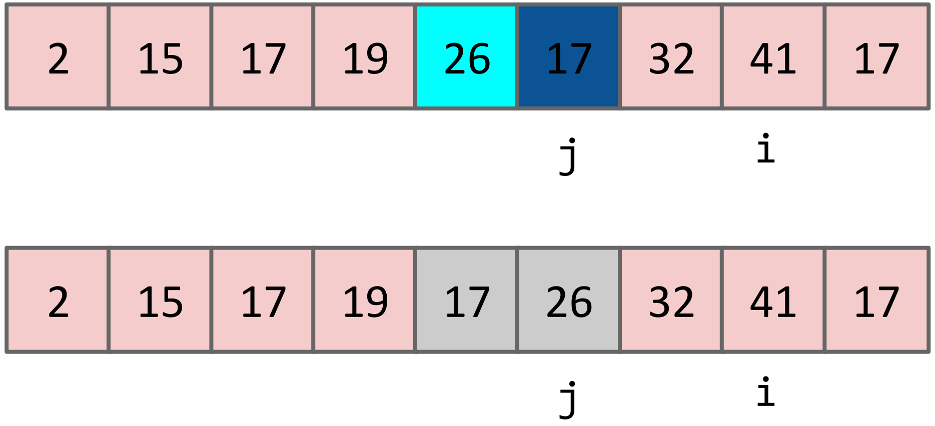
Observation: Each swap in Insertion Sort reduces the number of inversions by 1

Inversions between red items don't change (because they don't move)

Inversions between a red and blue item don't change (their relative position in the array stays the same)

We fix one inversion within the blue items

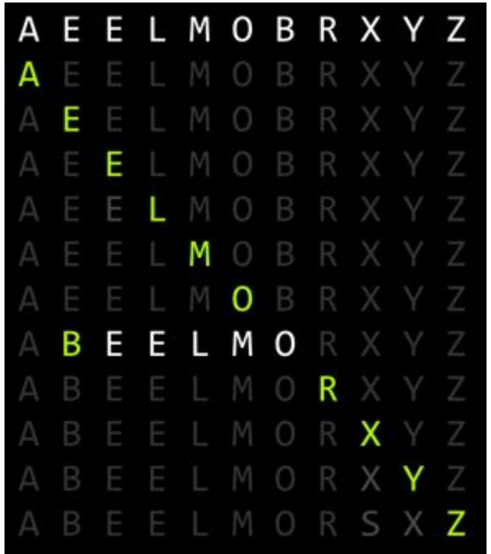
So total number of inversions decrease by 1 every swap



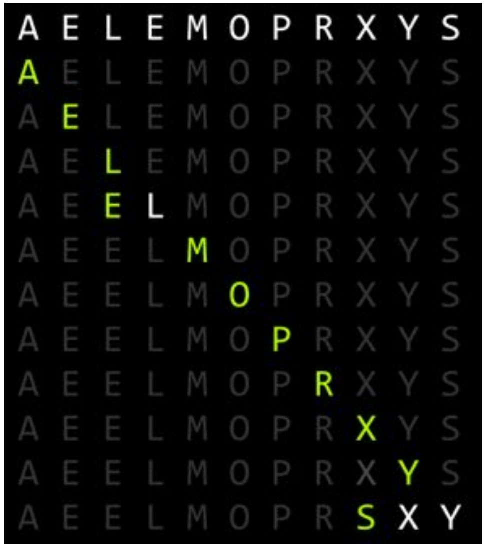
Observation: Insertion Sort on Almost Sorted Arrays

For arrays that are almost sorted, insertion sort does very little work.

- Left array: 5 inversions, so only 5 swaps.
- Right array: 3 inversion, so only 3 swaps.



```
A E E L M O B R X Y Z
A E E L M O B R X Y Z
A E E L M O B R X Y Z
A E E L M O B R X Y Z
A E E L M O B R X Y Z
A E E L M O B R X Y Z
A E E L M O B R X Y Z
A B E E L M O R X Y Z
A B E E L M O R X Y Z
A B E E L M O R X Y Z
A B E E L M O R X Y Z
A B E E L M O R S X Z
```



```
A E L E M O P R X Y S
A E L E M O P R X Y S
A E L E M O P R X Y S
A E L E M O P R X Y S
A E E L M O P R X Y S
A E E L M O P R X Y S
A E E L M O P R X Y S
A E E L M O P R X Y S
A E E L M O P R X Y S
A E E L M O P R X Y S
A E E L M O P R X Y S
A E E L M O P R S X Y
```

Picking the Best Sort (Poll Everywhere)

Suppose we do the following:

- Read 1,000,000 integers from a file into an array of length 1,000,000.
- Mergesort these integers.
- Select one integer randomly and change it.
- Sort using algorithm X of your choice.
- In the worst case, we have 999,999 inversions: Θ(N) inversions.

Which sorting algorithm would be the fastest choice for X? Worst case run-times:

A. Selection Sort: $\Theta(N^2)$
B. Heapsort: Θ(N log N)
C. Mergesort: Θ(N log N)
D. Insertion Sort: Θ(N)

Insertion Sort Sweet Spots

On arrays with a small number of inversions, insertion sort is extremely fast.

- One exchange per inversion (and number of comparisons is similar). Runtime is $\Theta(N + K)$ where K is number of inversions.
- K is on average $\Theta(N^2)$ in random lists, but…
- Define an ***almost sorted*** array as one in which number of inversions $\leq cN$ for some c. Insertion sort is excellent on these arrays.

Less obvious: For small arrays ($N < 15$ or so), insertion sort is fastest.

- More of an empirical fact than a theoretical one.
- Theoretical analysis beyond scope of the course.
- Rough idea: Divide and conquer algorithms like heapsort / mergesort spend too much time dividing, but insertion sort goes straight to the conquest.
- The Java implementation of Mergesort does this (Link).

Sorts So Far

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	Link	
Heapsort (in place)	$\Theta(N)$*	$\Theta(N \log N)$	$\Theta(1)$	Link	Bad cache (61C) performance.
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Link	Fastest of these.
Insertion Sort (in place)	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$	Link	Best for small N or almost sorted.

See this link for bonus slides on Shell's Sort, an optimization of insertion sort.

Miscellaneous Sorts

Lecture 30, CS61B, Spring 2025

A lightning round of other sorting algorithms

There are many sorting algorithms. Not all of them are actually useful, but they still get referenced every now and then.

We haven't yet covered the fastest sorting algorithms. But before we do that, let's go over a few other sorting algorithms.

Adaptive Sorting Algorithms

On arrays with a small number of inversions, insertion sort is extremely fast.

- One exchange per inversion (and number of comparisons is similar). Runtime is Θ(N + K) where K is number of inversions.
- This is known as an **adaptive sorting algorithm**, since it takes advantage of existing order

More generally, we can think of sorting algorithms that behave like this:

1. Find a pair of adjacent elements in the wrong order (if you can't find one, the list is sorted)
2. Swap the two elements
3. Repeat steps 1 and 2 until the list is sorted

Step 2 fixes one inversion each time, so we get Θ(K) time from that step

Step 1 in insertion sort looks at and solves the inversions of one item at a time, and takes Θ(N+K) in total (since we check N+K total pairs for inversions).

Bad sorting algorithm: Bubble sort

Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:

32	15	2	17	19	26	41	17	17

Bad sorting algorithm: Bubble sort

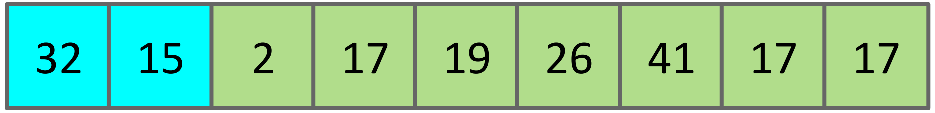
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

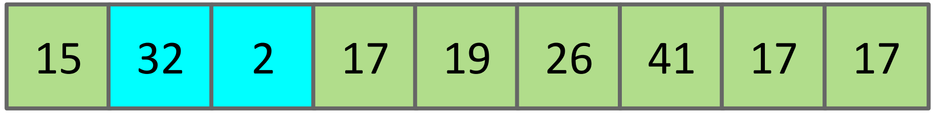
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

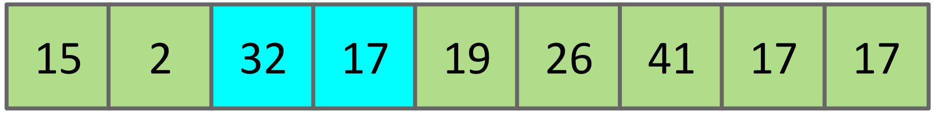
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

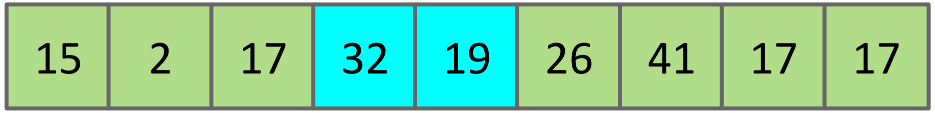
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

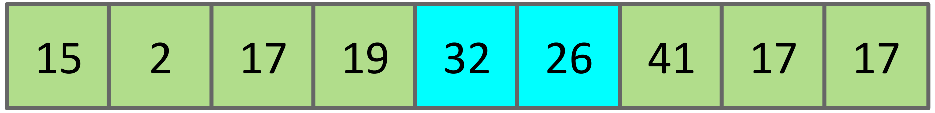
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

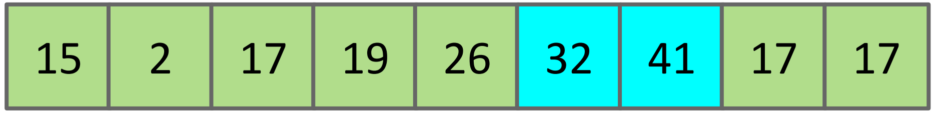
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

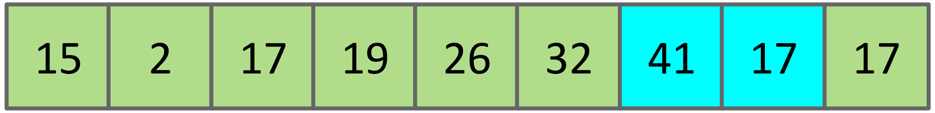
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

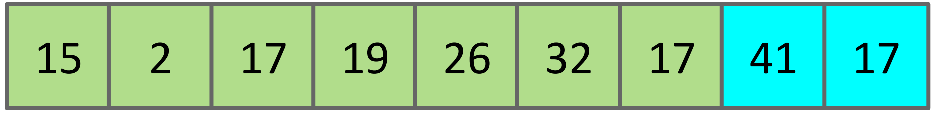
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

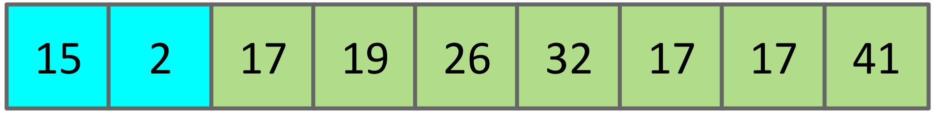
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

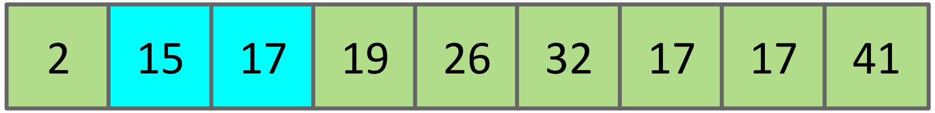
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

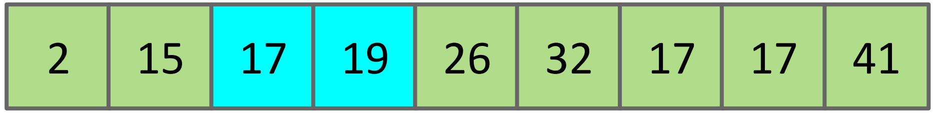
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

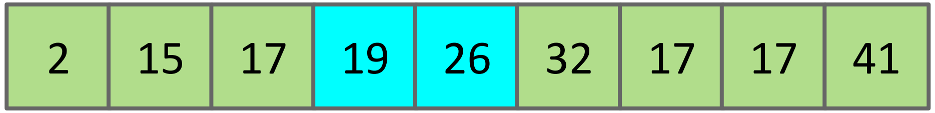
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

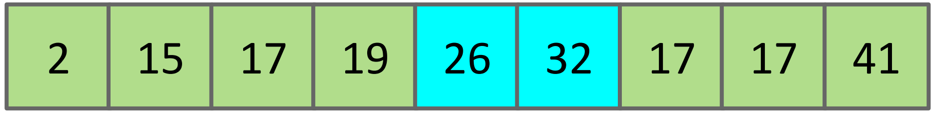
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

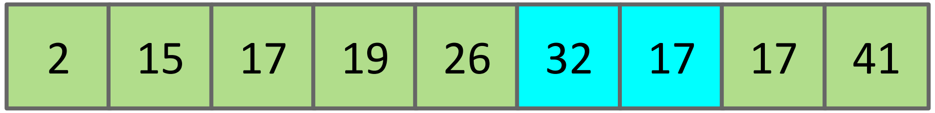
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

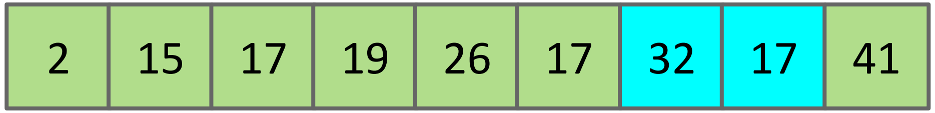
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



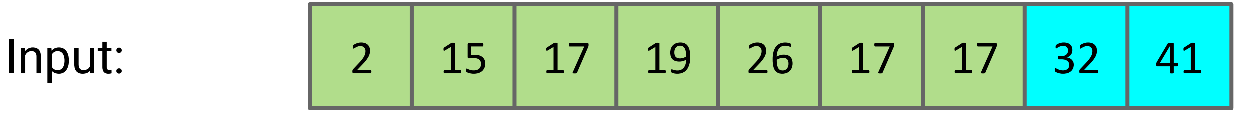
Bad sorting algorithm: Bubble sort

Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted



Bad sorting algorithm: Bubble sort

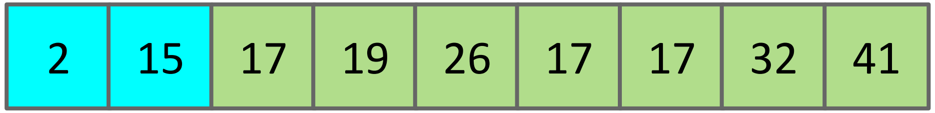
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

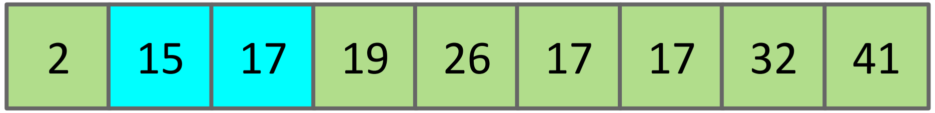
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

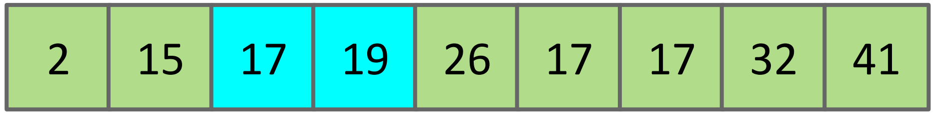
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

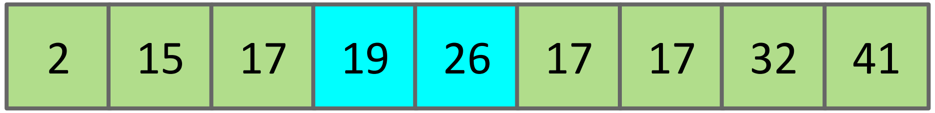
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

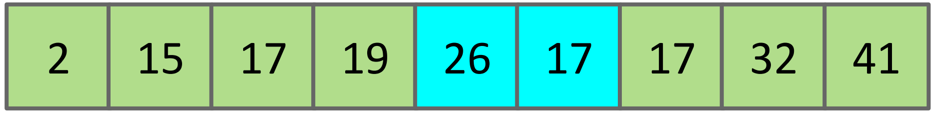
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

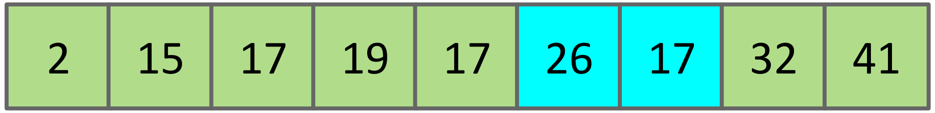
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

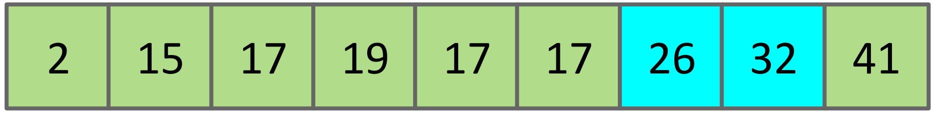
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

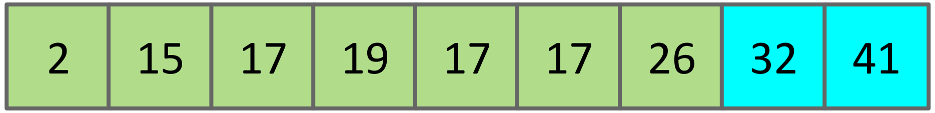
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

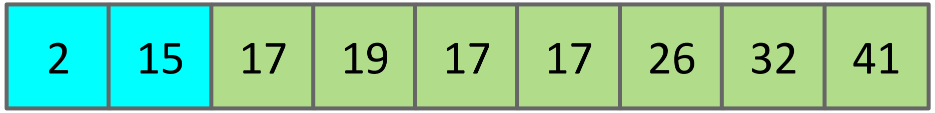
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

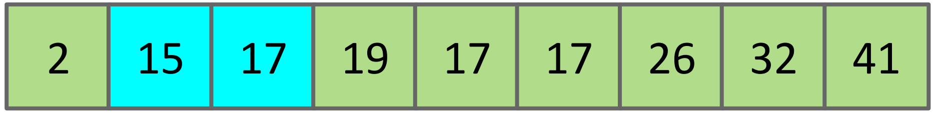
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

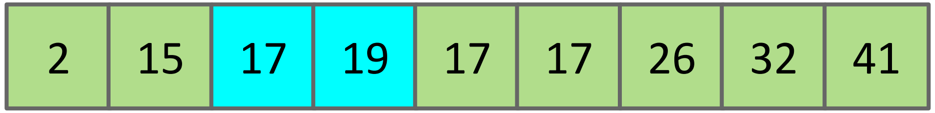
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

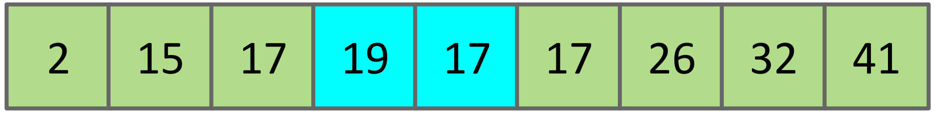
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

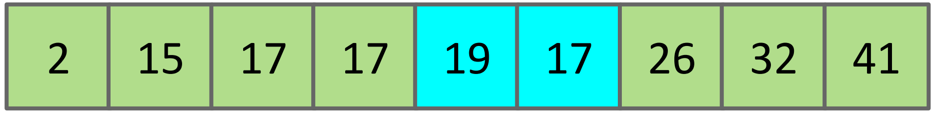
Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

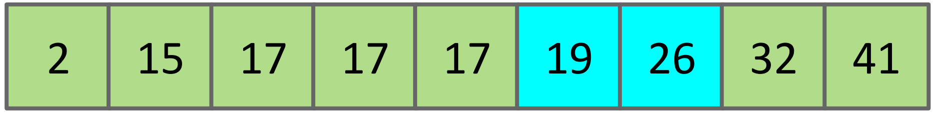
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

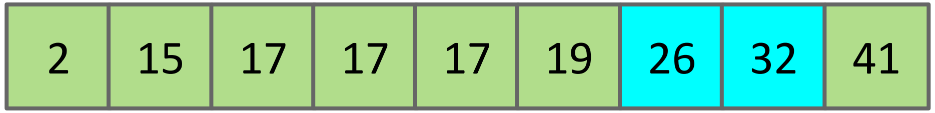
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

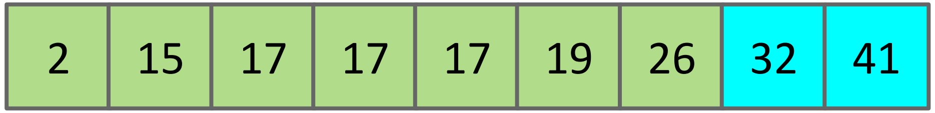
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

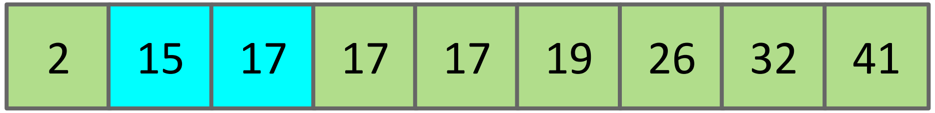
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

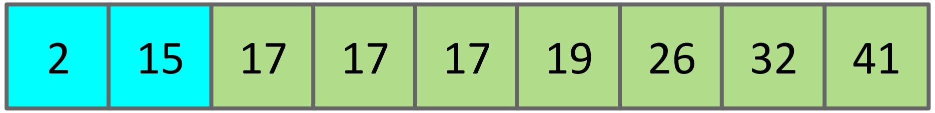
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

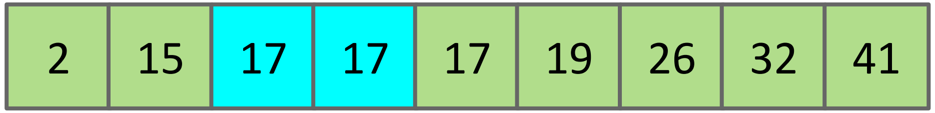
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

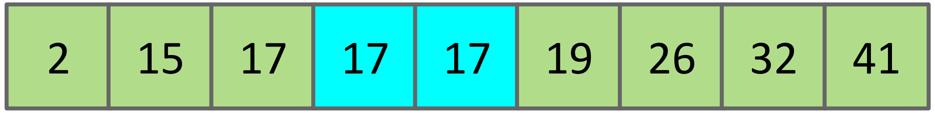
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

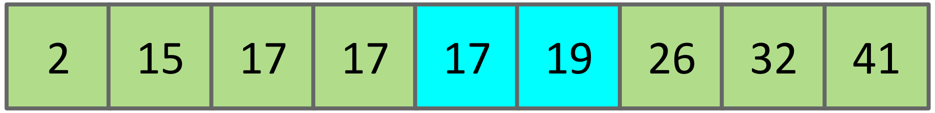
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

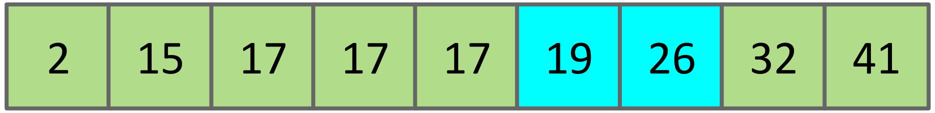
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

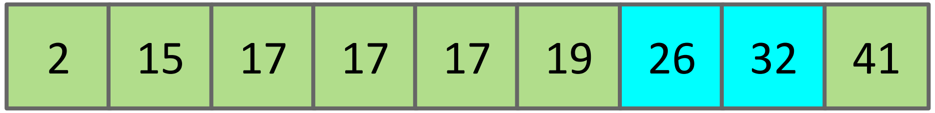
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

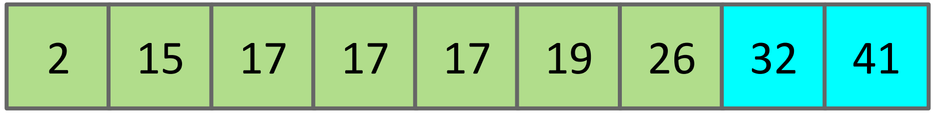
Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

(At this point, the list is sorted, but we run through the list one more time to confirm that)

Input:



Bad sorting algorithm: Bubble sort

Bubble sort:

Iterate through the list and look at each adjacent pair

If the two elements are in a wrong order, swap them

Repeat iterating through list until the array is sorted

Input:

2	15	17	17	17	19	26	32	41

Bad sorting algorithm: Bubble sort

Bubble sort is the same as insertion sort, except it doesn't find inversions as quickly.

Very slow in theory, and slow in practice too (~5x slower than insertion sort)

Items that need to move left are "turtles" that take a long time to move to the right spot.

Other variations of insertion sort:

- Cocktail shaker sort
  - Bubble sort but iteration order swaps forwards and backwards
  - Fixes the issue of turtles, but still slow
- Shell sort
  - Instead of comparing adjacent items, compare at longer distances first to fix multiple inversions at once.
  - Depending on the jump lengths, can yield better asymptotic runtime than insertion sort
  - See this link for more info

Shuffling

Let's try solving the "reverse" problem to sorting: Shuffling

Given a list of n elements and a random number generator, return a *random* permutation (reordering) of the list.

The random number generator receives as input two integers i and j, and returns a random integer between i and j.

Goals:

- Even distribution (after shuffling, each of the n! permutations are equally likely)
- Fast runtime (Θ(n) runtime)

Input:

1	2	3	4	5	6	7	8	9

Shuffling

For each number i from 0 to n-1:

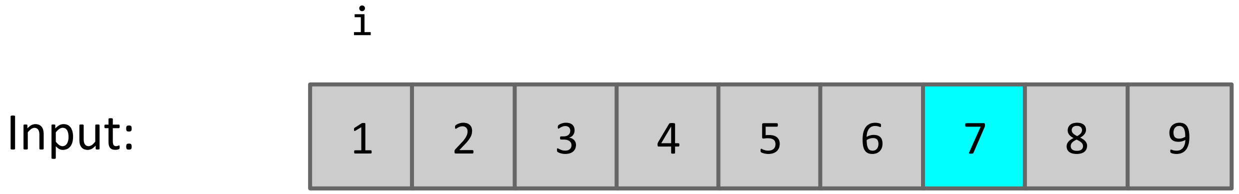
- Pick a random item by selecting an index from i to n-1
- Swap that item with the item in index i

Input:	1	2	3	4	5	6	7	8	9

Shuffling

For each number i from 0 to n-1:

- **Pick a random item by selecting an index from i to n-1**
- Swap that item with the item in index i

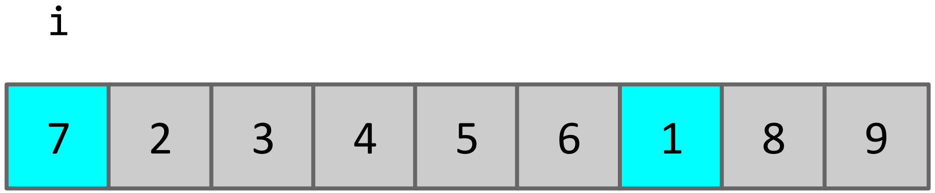


Shuffling

For each number i from 0 to n-1:

- Pick a random item by selecting an index from i to n-1
- **Swap that item with the item in index i**

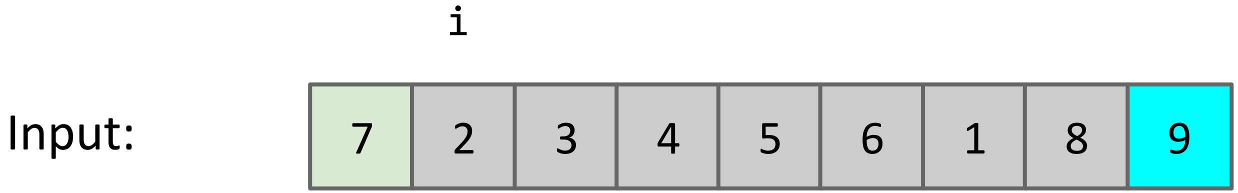
Input:



Shuffling

For each number i from 0 to n-1:

- **Pick a random item by selecting an index from i to n-1**
- Swap that item with the item in index i

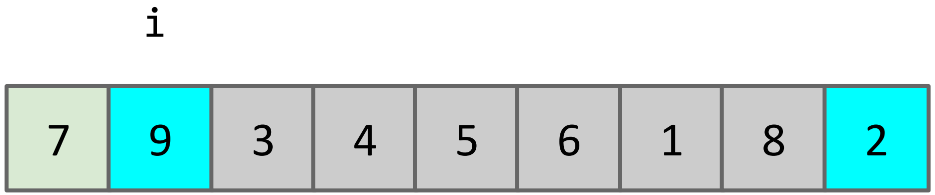


Shuffling

For each number i from 0 to n-1:

- Pick a random item by selecting an index from i to n-1
- **Swap that item with the item in index i**

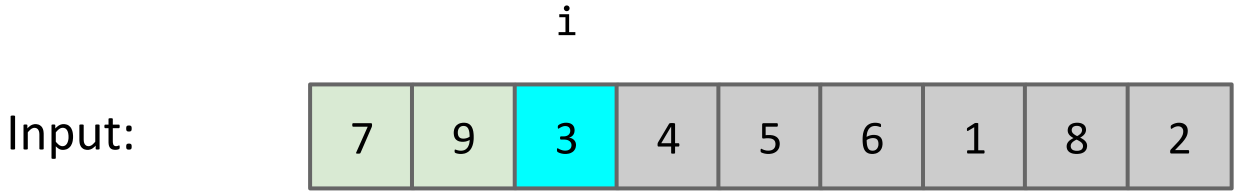
Input:



Shuffling

For each number i from 0 to n-1:

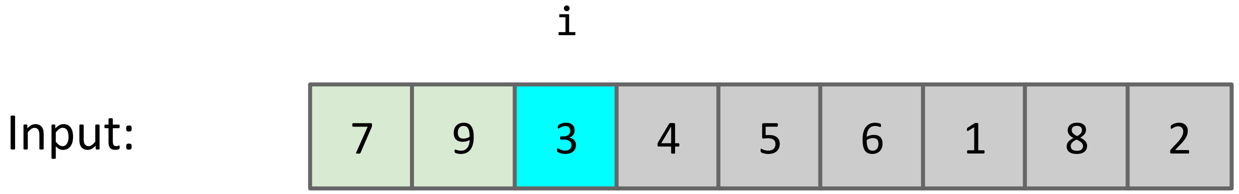
- **Pick a random item by selecting an index from i to n-1**
- Swap that item with the item in index i



Shuffling

For each number i from 0 to n-1:

- Pick a random item by selecting an index from i to n-1
- **Swap that item with the item in index i**

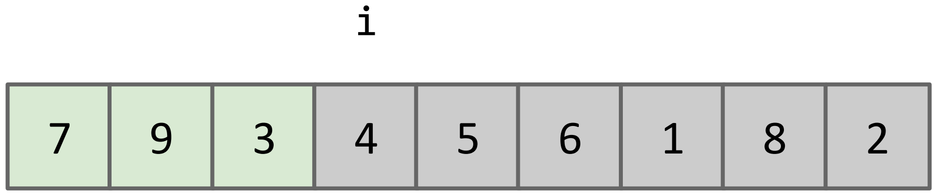


Shuffling

For each number i from 0 to n-1:

- **Pick a random item by selecting an index from i to n-1**
- Swap that item with the item in index i

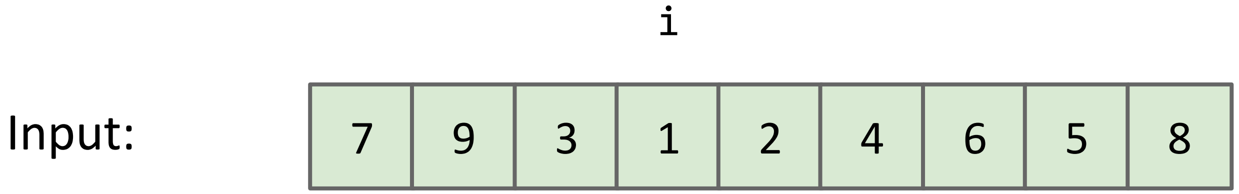
(Remaining Steps omitted)

Input: 

Shuffling

For each number i from 0 to n-1:

- Pick a random item by selecting an index from i to n-1
- Swap that item with the item in index i

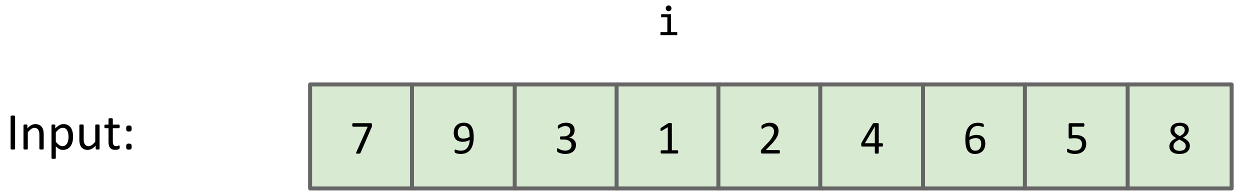


Shuffling

For each number i from 0 to n-1:

- Pick a random item by selecting an index from i to n-1
- Swap that item with the item in index i

Runtime?



Shuffling

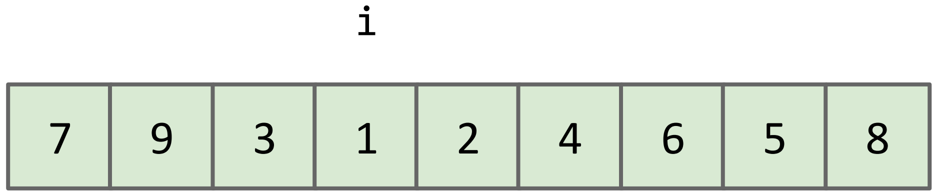
For each number i from 0 to n-1:

- Pick a random item by selecting an index from i to n-1
- Swap that item with the item in index i

Runtime? **Θ(n)** (We do n iterations of constant time each)

Even distribution?

Input:



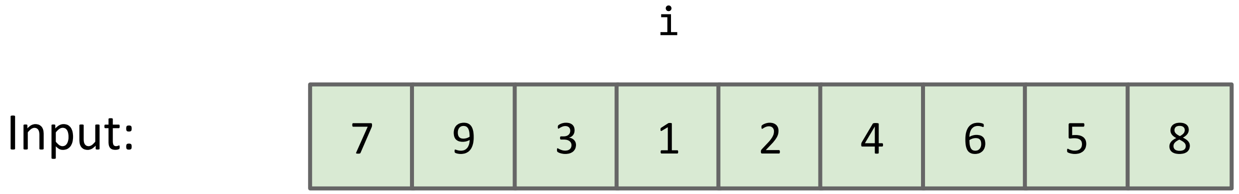
Shuffling

For each number i from 0 to n-1:

- Pick a random item by selecting an index from i to n-1
- Swap that item with the item in index i

Runtime? **Θ(n)**

Even distribution? **Perfectly even** (Each permutation has exactly one way of being created, with probability 1/n * 1/(n-1) * … * (1/1) = 1/(n!)



Bad sorting algorithm: BOGO sort

BOGO sort:

While list is not sorted: (O(n) time to check if list is sorted)

- Shuffle the list (Θ(n) time to shuffle)

Let's analyze this quickly:

Best-case runtime:

Worst-case runtime:

Average runtime:

Bad sorting algorithm: BOGO sort

Here's a bad sorting algorithm:

While list is not sorted: (O(n) time to check if list is sorted)

- Shuffle the list (Θ(n) time to shuffle)

Let's analyze this quickly:

Best-case runtime: Θ(n) (if the list is sorted initially)

Worst-case runtime: Infinite runtime (if we never shuffle it the right way)

Average runtime: Θ(n*n!) (while loop has on average n! iterations before we get the sorted order)

Quicksort Backstory, Partitioning

Lecture 30, CS61B, Spring 2025

Sorts So Far

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Selection Sort	Θ(N^2)	Θ(N^2)	Θ(1)	Link	
Heapsort (in place)	Θ(N)*	Θ(N log N)	Θ(1)	Link	Bad cache (61C) performance.
Mergesort	Θ(N log N)	Θ(N log N)	Θ(N)	Link	Fastest of these.
Insertion Sort (in place)	Θ(N)	Θ(N^2)	Θ(1)	Link	Best for small N or almost sorted.

Sorting So Far

Core ideas:

- Selection sort: Find the smallest item and put it at the front.
  - Heapsort variant: Use MaxPQ to find max element and put at the back.
- Merge sort: Merge two sorted halves into one sorted whole.
- Insertion sort: Figure out where to insert the current item.

Quicksort:

- Core idea: Partitioning.
- Invented by Sir Tony Hoare in 1960, at the time a novice programmer.

Context for Quicksort's Invention (Source)

1960: Tony Hoare was working on a crude automated translation program for Russian and English.

"The cat wore a beautiful hat."

N words

...	...
beautiful	красивая
...	...
cat	кошка
...	...

Dictionary of D english words

"Кошка носил красивая шапка."

How would you do this?

- Binary search for each word.
  - Find "the" in log D time.
  - Find "cat" in log D time...
- Total time: N log D

Context for Quicksort's Invention (Source)

1960: Tony Hoare was working on a crude automated translation program for Russian and English.

Algorithm: N binary searches of D length dictionary.

- Total runtime: N log D
- ASSUMES log time binary search!

…	…
beautiful	красивая
…	…
cat	кошка
…	…

Limitation at the time:

- Dictionary stored on long piece of tape, sentence is an array in RAM.
    - Binary search of tape is not log time (requires physical movement!).
- Better: **Sort the sentence** and scan dictionary tape once. Takes N log N + D time.
    - But Tony had to figure out how to sort an array (without Google!)...

Partitioning

Main idea: I have a list of small and large items. To sort this:

- Separate the small items and large items into two separate piles
- Sort the small items
- Sort the large items
- Put the sorted lists together

How to decide what's big and what's small?

- Pick an item from the list (we'll call this the pivot). Smaller items are small, and bigger items are big

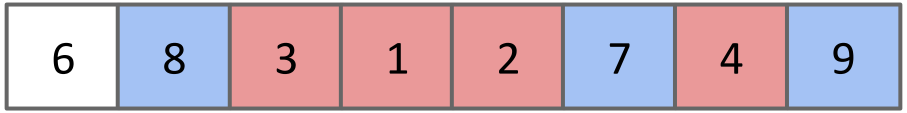
We'll call this a **partition**; we separate the big items and the small items, and put remaining items in the middle.

Partitioning

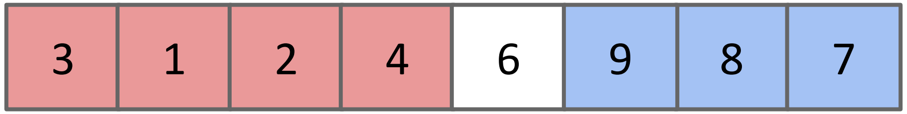
A **partition** of an array, given a **pivot** x, is a rearrangement of the items so that:

- All entries to the left of x are <= x.
- All entries to the right of x are >= x.
- x moves between the smaller and larger items

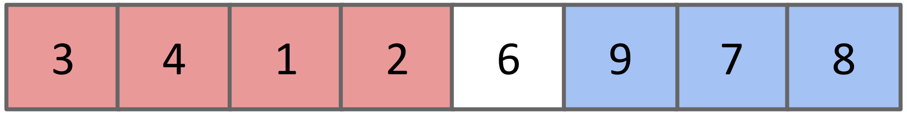
Input



Example of a valid output



Another example of a valid output



Core Idea of Tony's Sort: Partitioning http://yellkey.com/TODO

A **partition** of an array, given a **pivot** x, is a rearrangement of the items so that:

- All entries to the left of x are <= x.
- All entries to the right of x are >= x.

5	550	10	4	10	9	330

A.

			j			
4	5	9	10	10	330	550

B.

			j			
5	4	9	10	10	550	330

C.

				j		
5	9	10	4	10	550	330

D.

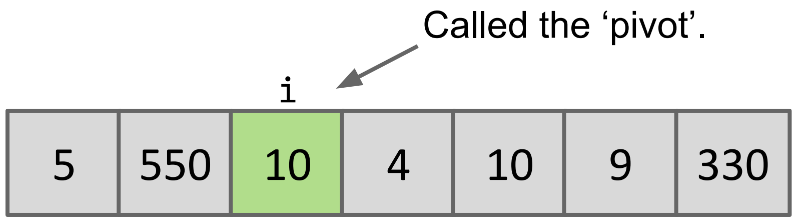
		j				
5	9	10	4	10	550	330

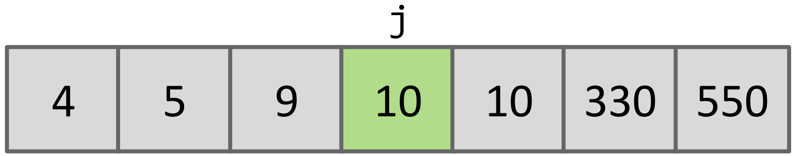
Which partitions are valid?

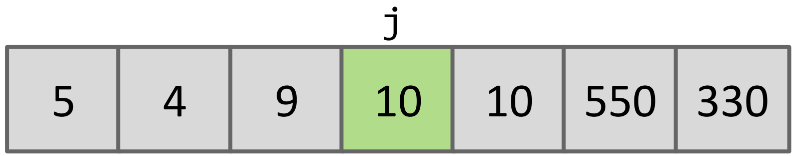
The Core Idea of Tony's Sort: Partitioning

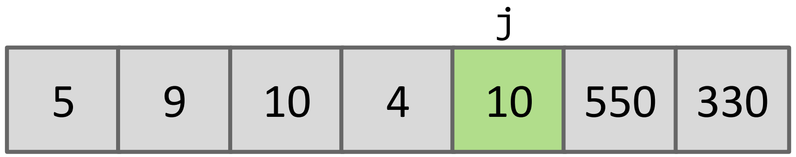
A **partition** of an array, given a **pivot** x, is a rearrangement of the items so that:

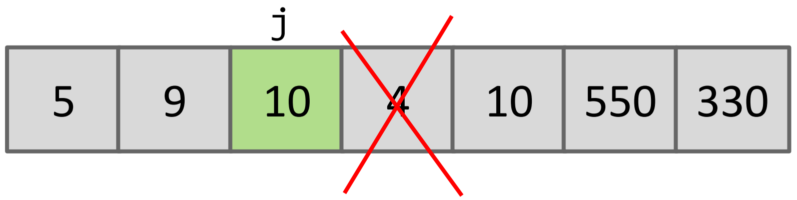
- All entries to the left of x are <= x.
- All entries to the right of x are >= x.



A.


B.


C.


D.


Which partitions are valid?

Job Interview Style Question (Partitioning)

Given an array of colors where the 0th element is white (and maybe a few more), and the remaining elements are red (less) or blue (greater), rearrange the array so that all red squares are to the left of the white square, the white squares end up together, and all blue squares are to the right. Your algorithm must complete in Θ (N) time (no space restriction).

- Relative order of red and blues does NOT need to stay the same.

Input

6	8	3	1	2	7	4	6

Example of a valid output

3	1	2	4	6	6	8	7

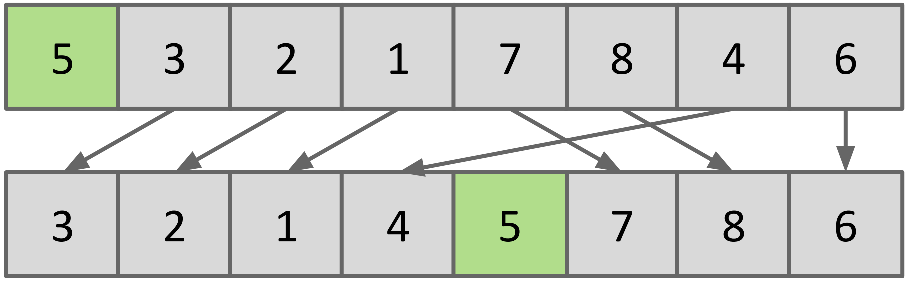
Another example of a valid output

3	4	1	2	6	6	7	8

Quicksort

Lecture 30, CS61B, Spring 2025

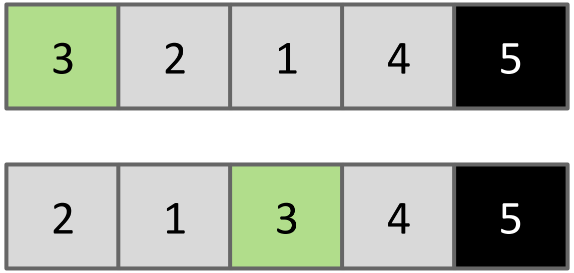
Partition Sort, a.k.a. Quicksort

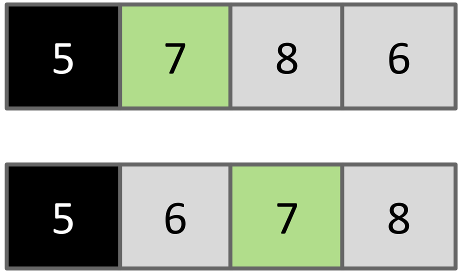


Q: How would we use this operation for sorting?

Observations:

- 5 is “in its place.” Exactly where it’d be if the array were sorted.
- Can sort two halves separately, e.g. through recursive use of partitioning.



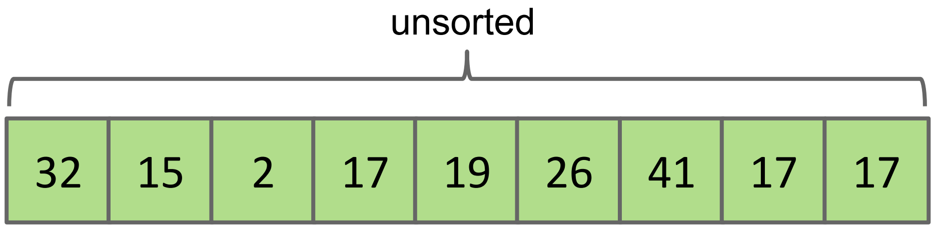


Quick Sort

Quick sorting N items:

- Partition on leftmost item.
- Quicksort left half.
- Quicksort right half.

Input:

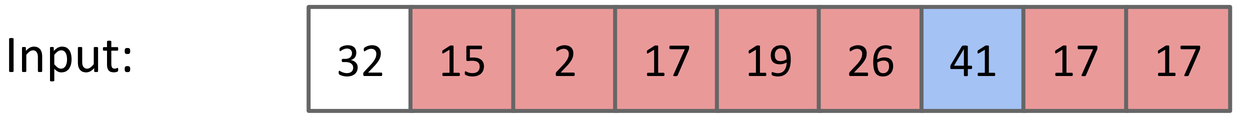


Quick Sort

Quick sorting N items:

- **Partition on leftmost item (32).**
- Quicksort left half.
- Quicksort right half.

partition(32)



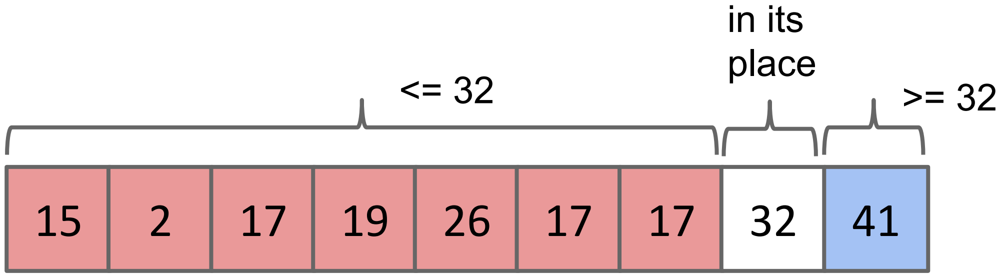
Quick Sort

Quick sorting N items:

- **Partition on leftmost item (32).**
- Quicksort left half.
- Quicksort right half.

partition(32)

Input:



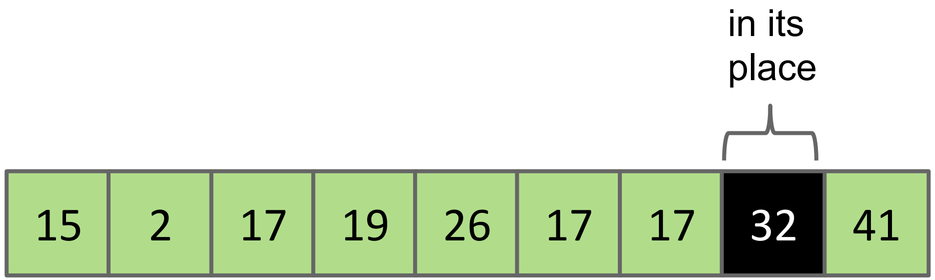
Quick Sort

Quick sorting N items:

- **Partition on leftmost item (32) (done).**
- Quicksort left half.
- Quicksort right half.

partition(32)

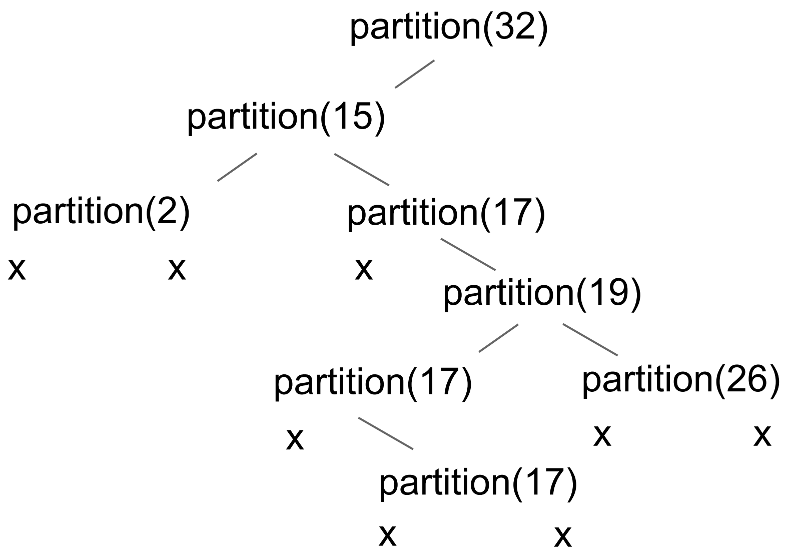
Input:

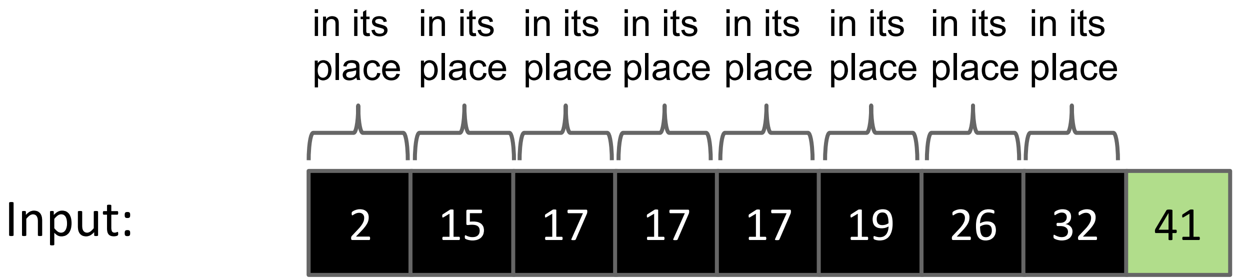


Quick Sort

Quick sorting N items:

- Partition on leftmost item (32) (done).
- **Quicksort left half (details not shown).**
- Quicksort right half.

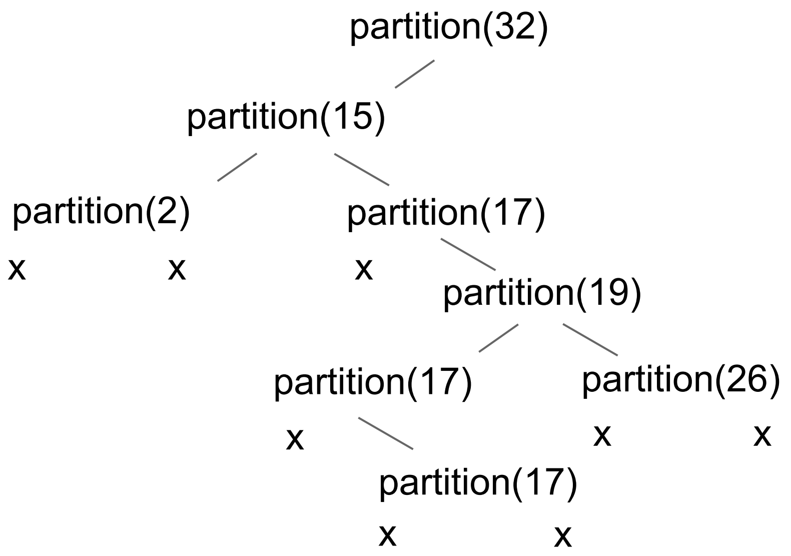




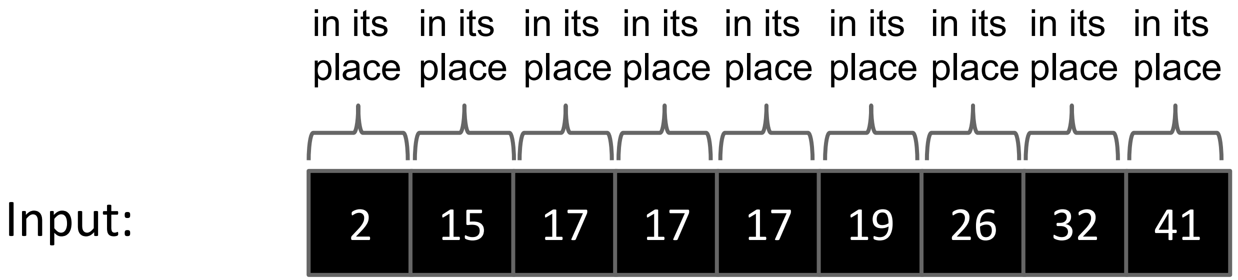
Quick Sort

Quick sorting N items:

- Partition on leftmost item (32) (done).
- Quicksort left half (details not shown).
- **Quicksort right half (details not shown).**



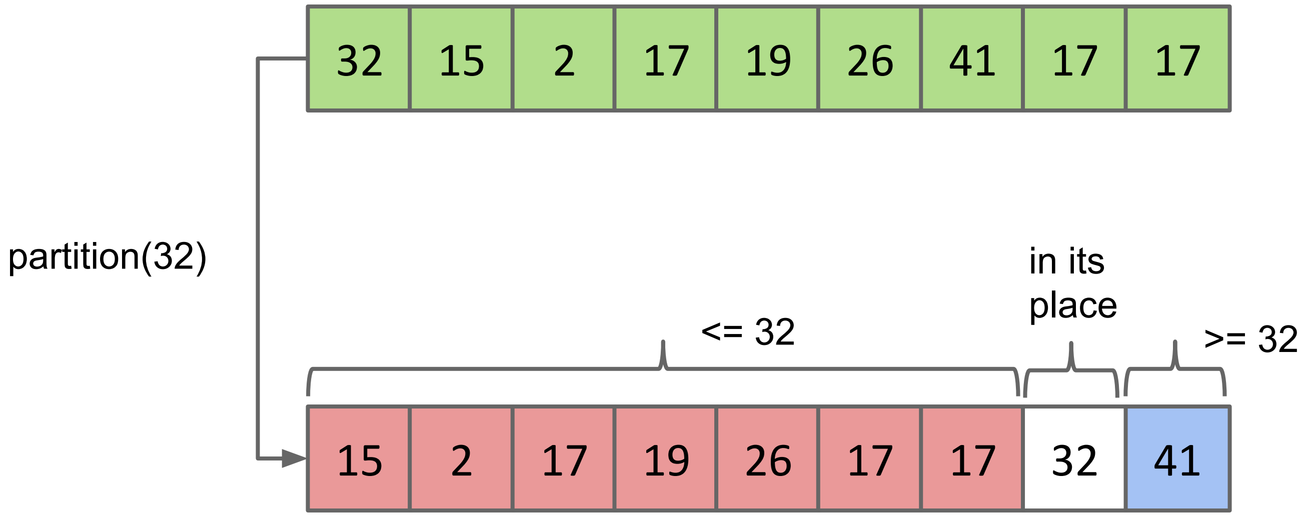
If you don't fully trust the recursion, see these extra slides for a complete demo.



Partition Sort, a.k.a. Quicksort

Quick sorting N items:

- Partition on leftmost item.
- Quicksort left half.
- Quicksort right half.



Quicksort

Quicksort was the name chosen by Tony Hoare for partition sort.

- For most common situations, it is empirically the fastest sort.
    - Tony was lucky that the name was correct.

How fast is Quicksort? Need to count number and difficulty of partition operations.

Theoretical analysis:

- Partitioning costs Θ(K) time, where Θ(K) is the number of elements being partitioned (as we saw in our earlier “interview question”).
- The interesting twist: Overall runtime will depend crucially on where pivot ends up.