Lecture 30 (Sorting 2)

Mergesort, Insertion Sort, and Quicksort

CS61B, Spring 2024 @ UC Berkeley

Slides credit: Josh Hug

Mergesort

Lecture 30, CS61B, Spring 2024

Mergesort

Insertion Sort

- Naive Insertion Sort
- In-Place Insertion Sort
- Insertion Sort Runtime

Quicksort

- Quicksort Backstory, Partitioning
- Quicksort

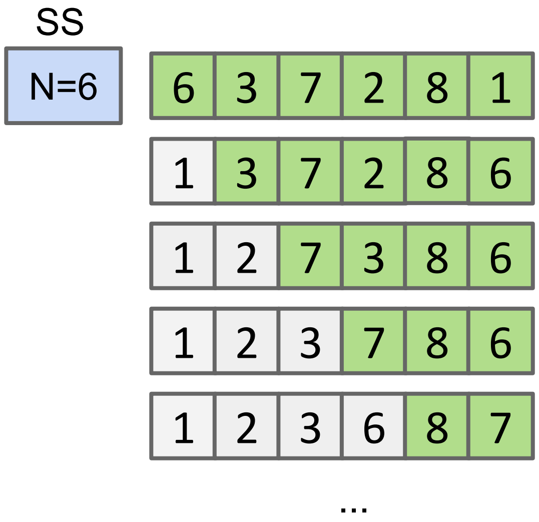
Selection Sort: A Prelude to Mergesort

Earlier we discussed a sort called selection sort:

- Find the smallest unfixed item, move it to the front, and ‘fix’ it.
- Sort the remaining unfixed items using selection sort.

Runtime of selection sort is Θ(N^2):

- Look at all N unfixed items to find smallest.
- Then look at N-1 remaining unfixed.
- …
- Look at last two unfixed items.
- Done, sum is 2+3+4+5+…+N = Θ(N^2)



6 3 7 2 8 1

1 3 7 2 8 6

1 2 7 3 8 6

1 2 3 7 8 6

1 2 3 6 8 7

…

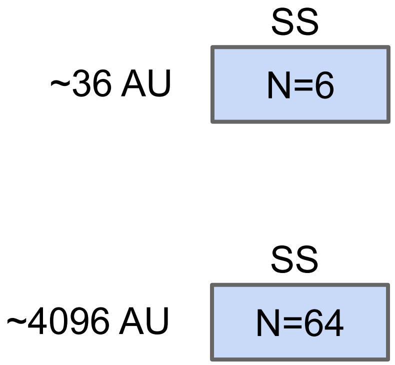
Selection Sort: A Prelude to Mergesort/Example 5

Earlier in class we discussed a sort called selection sort:

- Find the smallest unfixed item, move it to the front, and ‘fix’ it.
- Sort the remaining unfixed items using selection sort.

Runtime of selection sort is $\Theta(N^2)$:

- Look at all N unfixed items to find smallest.
- Then look at N-1 remaining unfixed.
- …
- Look at last two unfixed items.
- Done, sum is $2+3+4+5+\ldots+N = \Theta(N^2)$



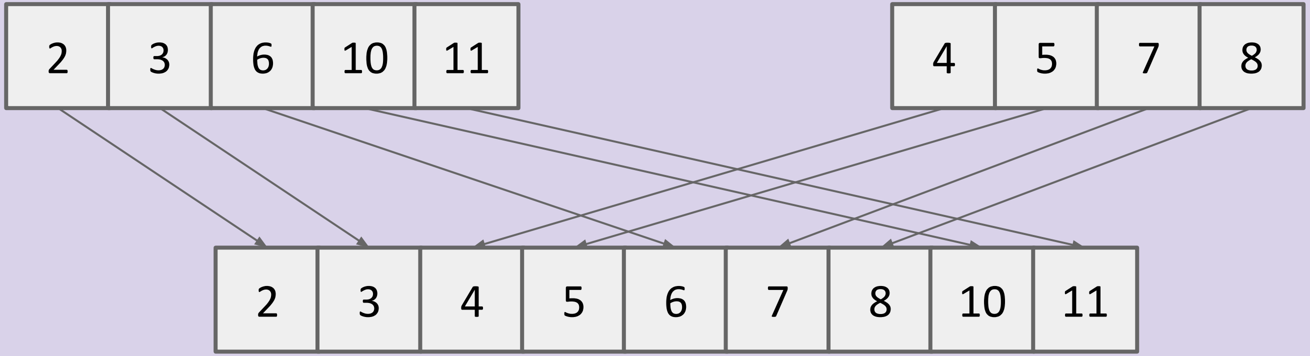
Given that runtime is quadratic, for N = 64, we might say the runtime for selection sort is 4,096 arbitrary units of time (AU).

The Merge Operation: Another Prelude to Mergesort/Example 5

Given two sorted arrays, the merge operation combines them into a single sorted array by successively copying the smallest item from the two arrays into a target array.

Merging Demo (Link)

Merge Runtime: http://yellkey.com/far



2	3	6	10	11

4	5	7	8

2	3	4	5	6	7	8	10	11

How does the runtime of merge grow with N, the total number of items?

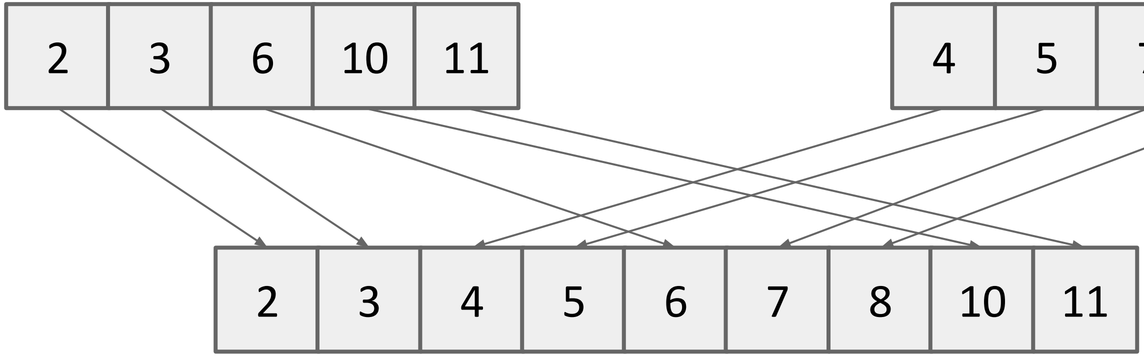
A. Θ(1)

B. Θ(log N)

C. Θ(N)

D. $\Theta(N^2)$

Merge Runtime



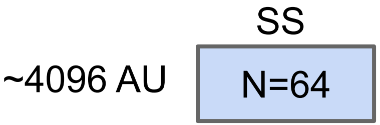
How does the runtime of merge grow with N, the total number of items?

C. Θ(N). Why? Use array writes as cost model, merge does exactly N writes.

Using Merge to Speed Up the Sorting Process

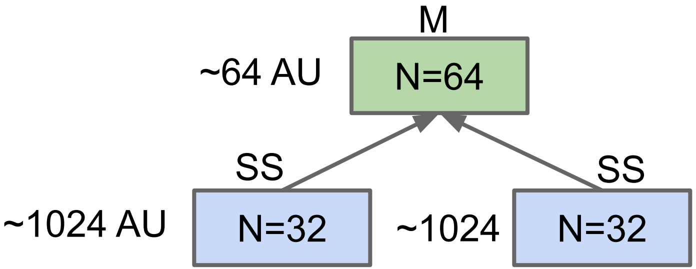
Merging can give us an improvement over vanilla selection sort:

- Selection sort the left half: Θ(N^2).
- Selection sort the right half: Θ(N^2).
- Merge the results: Θ(N).



N=64: ~2112 AU.

- Merge: ~64 AU.
- Selection sort: ~2*1024 = ~2048 AU.



Still Θ(N^2), but faster since $N+2*(N/2)^2 < N^2$

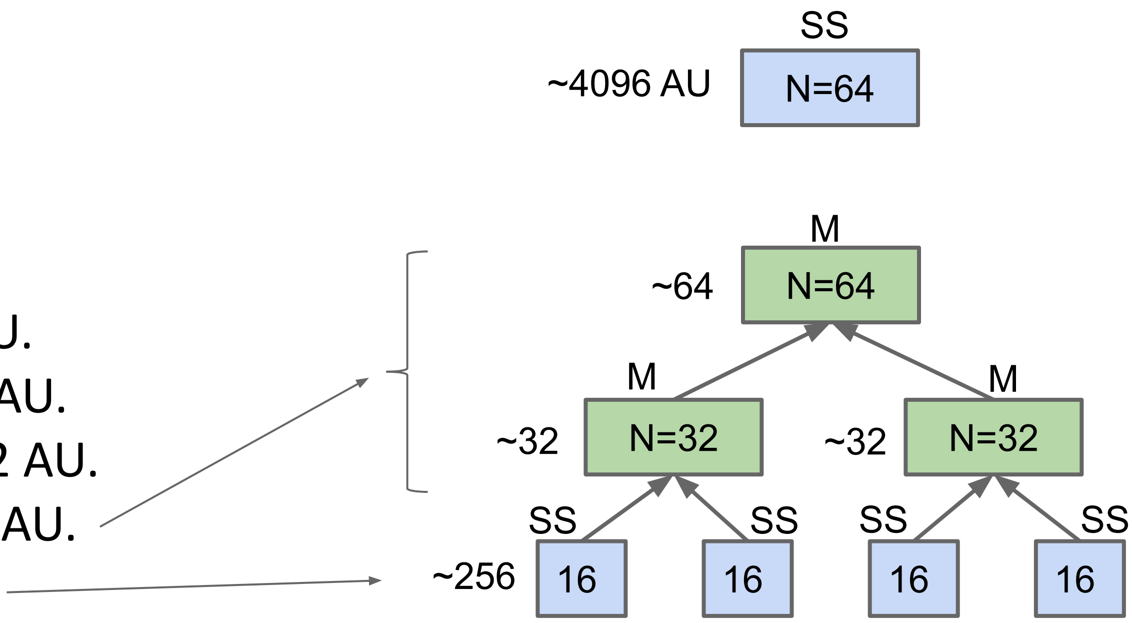
- ~2112 vs. ~4096 AU for N=64.

Two Merge Layers

Can do even better by adding a second layer of merges.

Runtime for each sort:

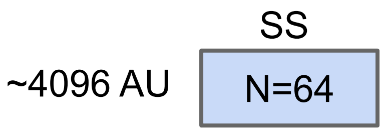
- Selection sort only: ~4096 AU.
- One layer of merges: ~2112 AU.
- Two layers of merges: ~1152 AU.
  - Merge: ~64 AU + 2*~32 AU.
  - Selection sort: 4*~256.



Example 5: Mergesort

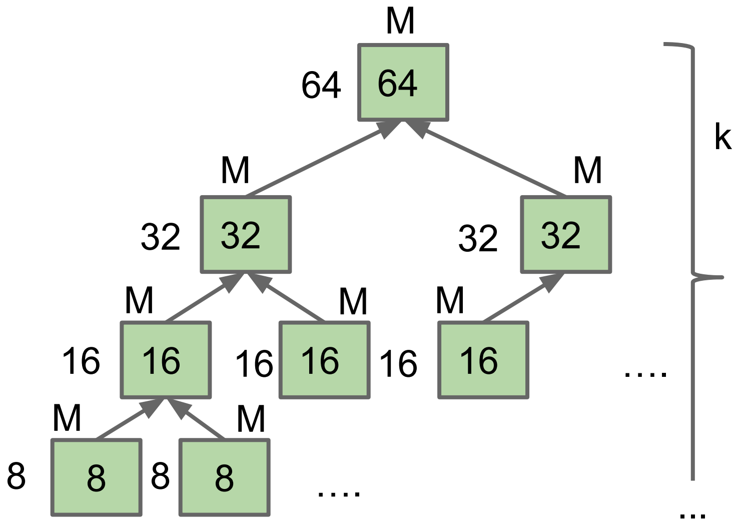
Mergesort does merges all the way down (no selection sort):

- If array is of size 1, return.
- Mergesort the left half: Θ(??).
- Mergesort the right half: Θ(??).
- Merge the results: Θ(N).



Total runtime to merge all the way down: ~384 AU

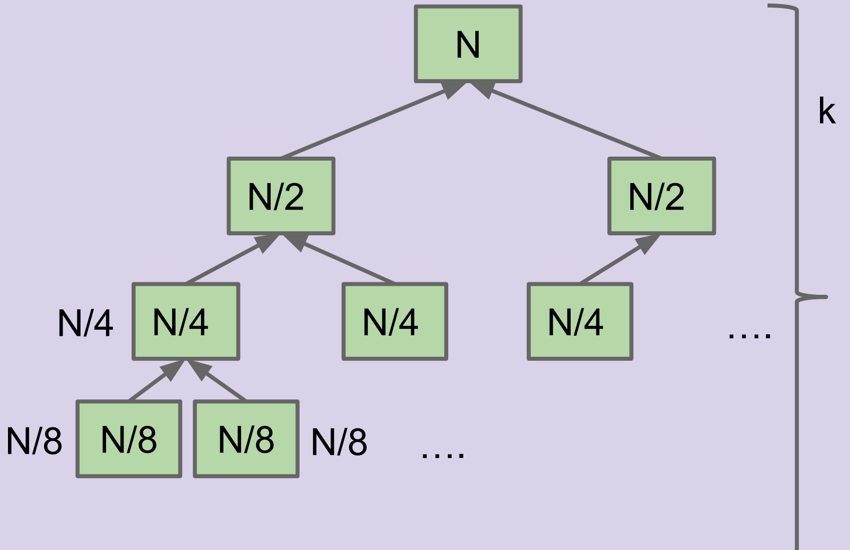
- Top layer: ~64 = 64 AU
- Second layer: ~32*2 = 64 AU
- Third layer: ~16*4 = 64 AU
- Overall runtime in AU is ~64k, where k is the number of layers.
- k = $\log_2(64)$ = 6, so ~384 total AU.



Example 5: Mergesort Order of Growth, yellkey.com/consider

For an array of size N, what is the worst case runtime of Mergesort?

A. Θ(1)
B. Θ(log N)
C. Θ(N)
D. Θ(N log N)
E. Θ(N^2)



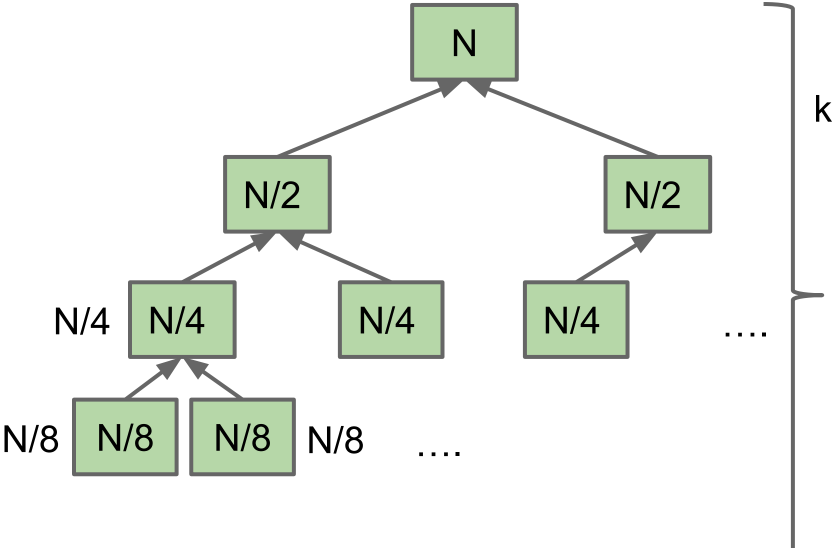
Example 5: Mergesort Order of Growth

Mergesort has worst case runtime = Θ(N log N).

- Every level takes ~N AU.
  - Top level takes ~N AU.
  - Next level takes ~N/2 + ~N/2 = ~N.
  - One more level down: ~N/4 + ~N/4 + ~N/4 + ~N/4 = ~N.
- Thus, total runtime is ~Nk, where k is the number of levels.
  - How many levels? Goes until we get to size 1.
  - k = $\log_2(N)$.
- Overall runtime is Θ(N log N).

Exact count explanation is tedious.

- Omitted here. See textbook exercises.



Mergesort

We’ve seen this one before as well.

Mergesort:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.

Time complexity, analysis from asymptotics lecture: Θ(N log N runtime)

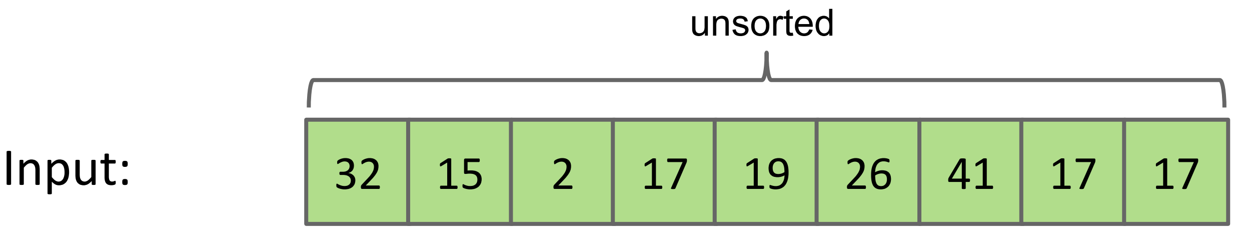
- Space complexity with aux array: Costs Θ(N) memory.

Also possible to do in-place merge sort, but algorithm is very complicated, and runtime performance suffers by a significant constant factor.

Top-Down Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half.
- Merge the two sorted halves to form the final result.



unsorted

Input:

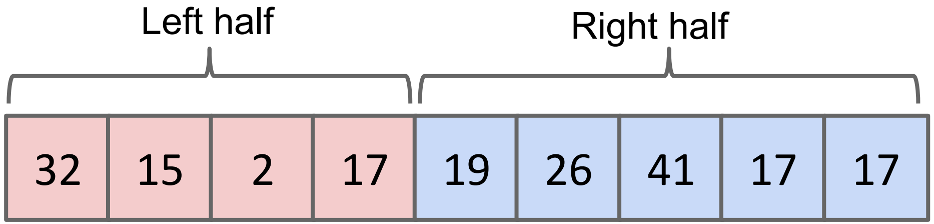
32	15	2	17	19	26	41	17	17

Top-Down Merge Sort

Top-Down merge sorting N items:

- **Split items into 2 roughly even pieces.**
- Mergesort each half.
- Merge the two sorted halves to form the final result.

Input:

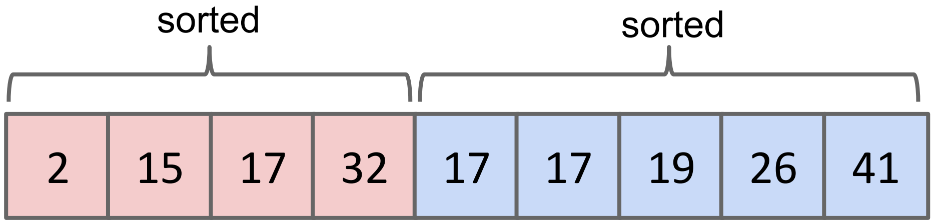


32	15	2	17	19	26	41	17	17

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- **Mergesort each half (steps not shown, this is a recursive algorithm!)**
- Merge the two sorted halves to form the final result.



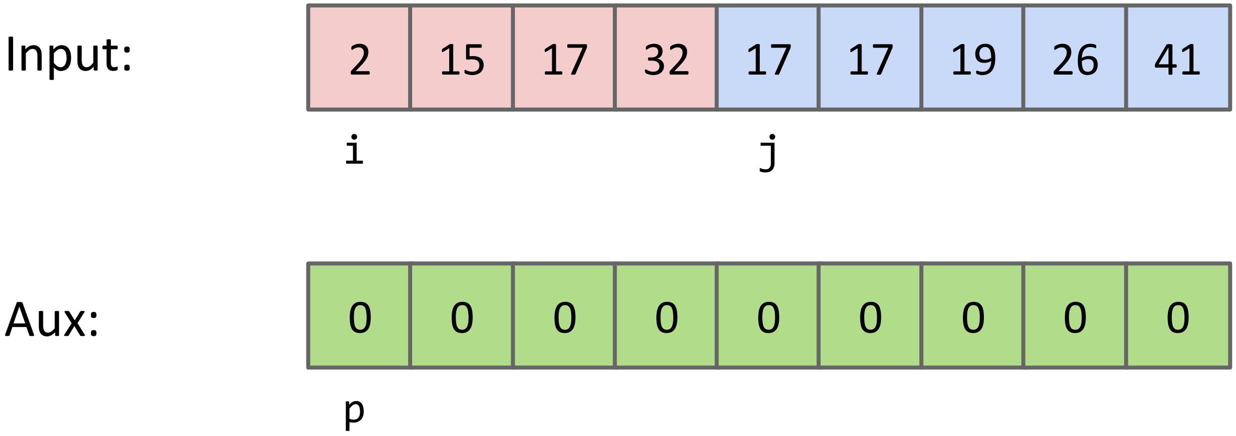
Input:

2	15	17	32	17	17	19	26	41

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- **Merge the two sorted halves to form the final result.**
  - Compare input[i] < input[j].
  - Copy smaller item and increment p and i or j.

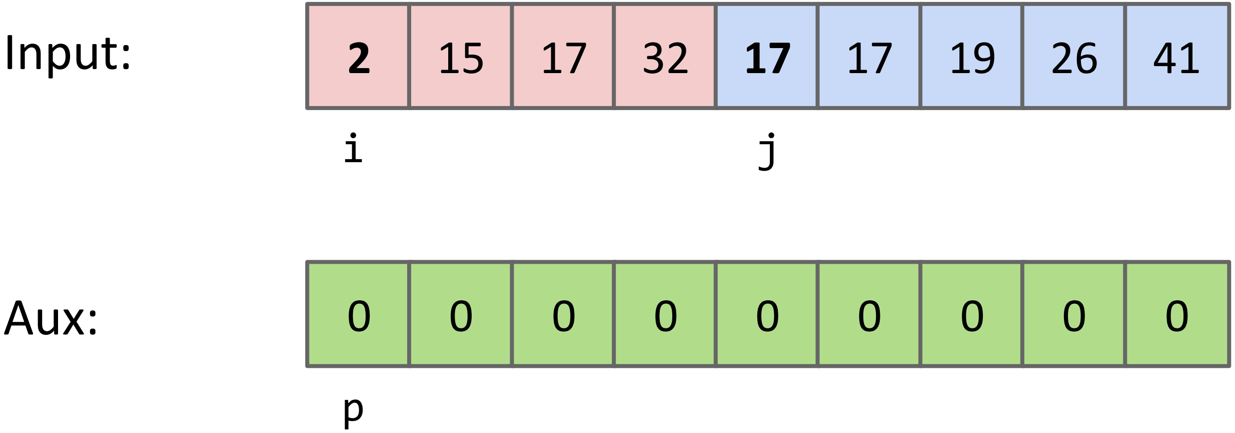


Input:	2	15	17	32	17	17	19	26	41
	i				j				
Aux:	0	0	0	0	0	0	0	0	0
	p								

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - **Compare input[i] < input[j] (if necessary).**
  - Copy smaller item and increment p and i or j.

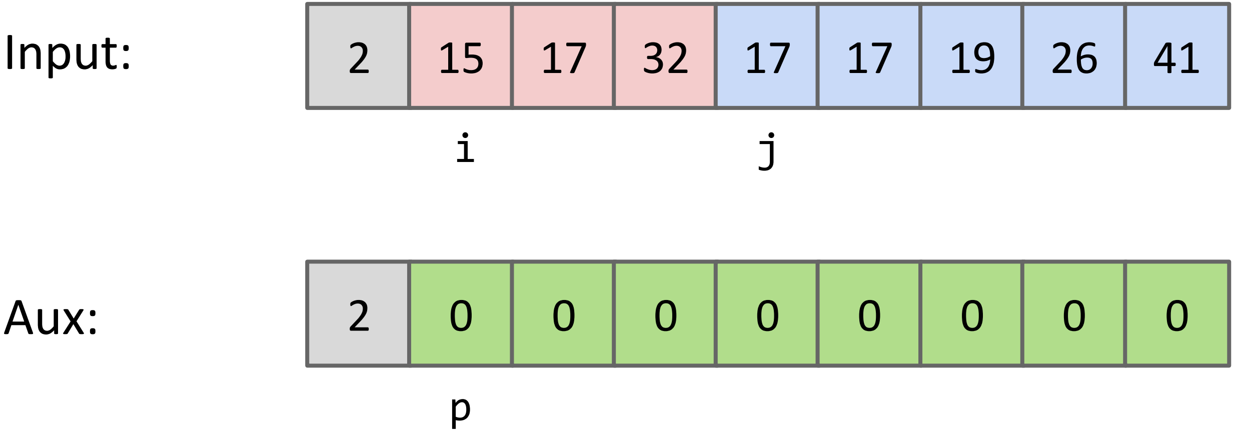


Input:	**2**	15	17	32	**17**	17	19	26	41
	i				j				
Aux:	0	0	0	0	0	0	0	0	0
	p								

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - Compare input[i] < input[j] (if necessary).
    - **Copy smaller item and increment p and i or j.**

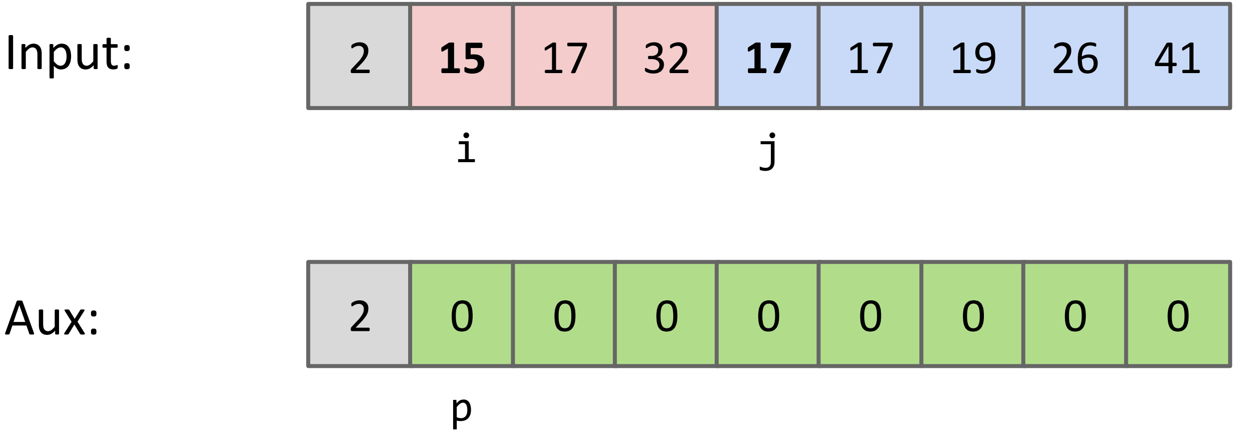


Input:	2	15	17	32	17	17	19	26	41
		i			j				
Aux:	2	0	0	0	0	0	0	0	0
		p							

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - **Compare input[i] < input[j] (if necessary).**
    - Copy smaller item and increment p and i or j.

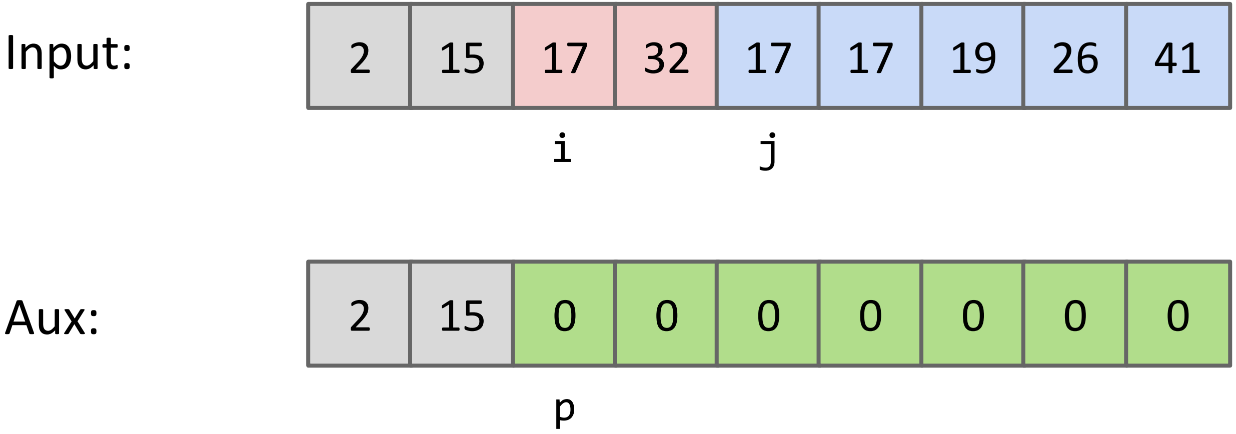


Input:	2	**15**	17	32	**17**	17	19	26	41
		i			j				
Aux:	2	0	0	0	0	0	0	0	0
		p							

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**

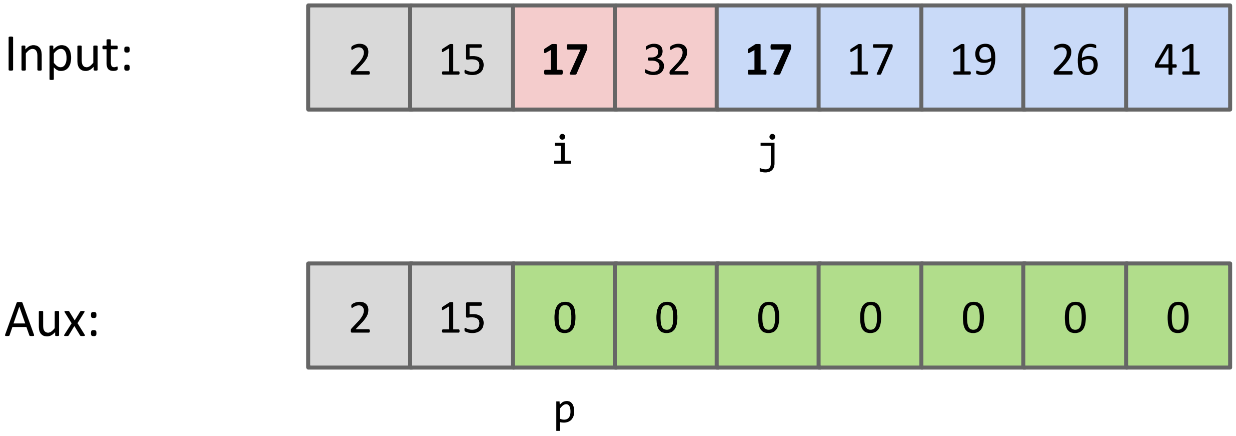


Input:	2	15	17	32	17	17	19	26	41
			i		j				
Aux:	2	15	0	0	0	0	0	0	0
			p						

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - **Compare input[i] < input[j] (if necessary).**
    - Copy smaller item and increment p and i or j.



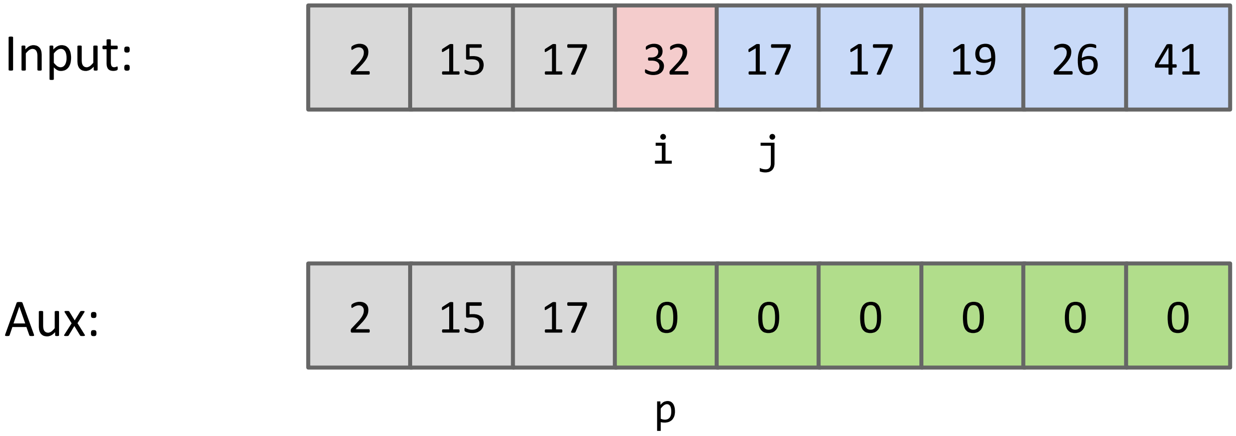
Input:	2	15	**17**	32	**17**	17	19	26	41
			i		j				

Aux:	2	15	0	0	0	0	0	0	0
			p						

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - Compare input[i] < input[j] (if necessary).
    - **Copy smaller item and increment p and i or j.**



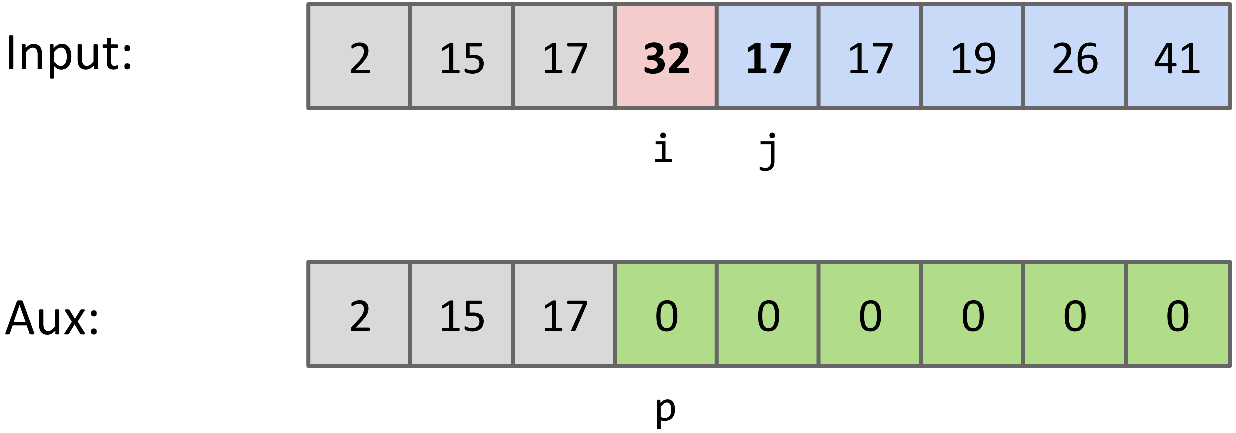
Input:	2	15	17	32	17	17	19	26	41
				i	j				

Aux:	2	15	17	0	0	0	0	0	0
				p					

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - **Compare input[i] < input[j] (if necessary).**
    - Copy smaller item and increment p and i or j.

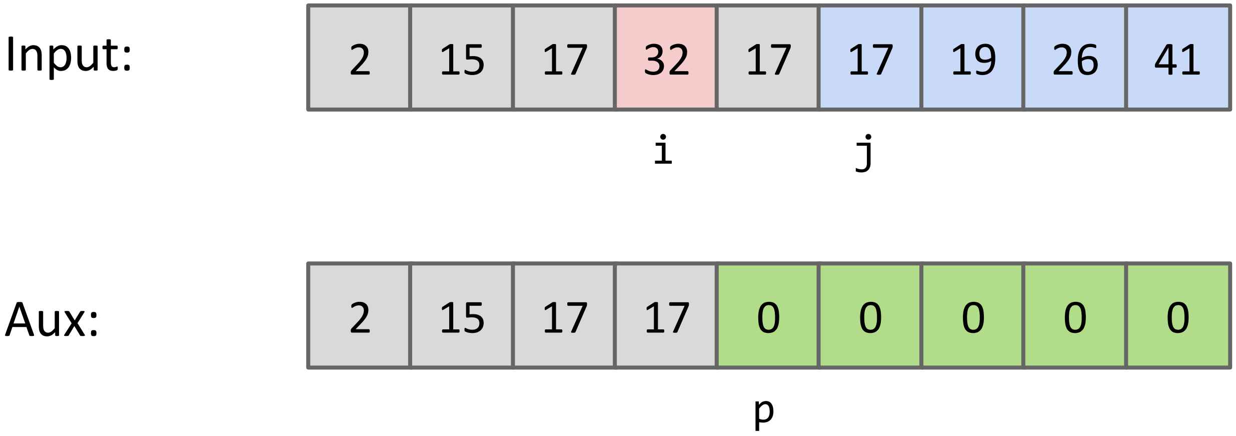


Input:	2	15	17	**32**	**17**	17	19	26	41
				i	j				
Aux:	2	15	17	0	0	0	0	0	0
				p					

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**



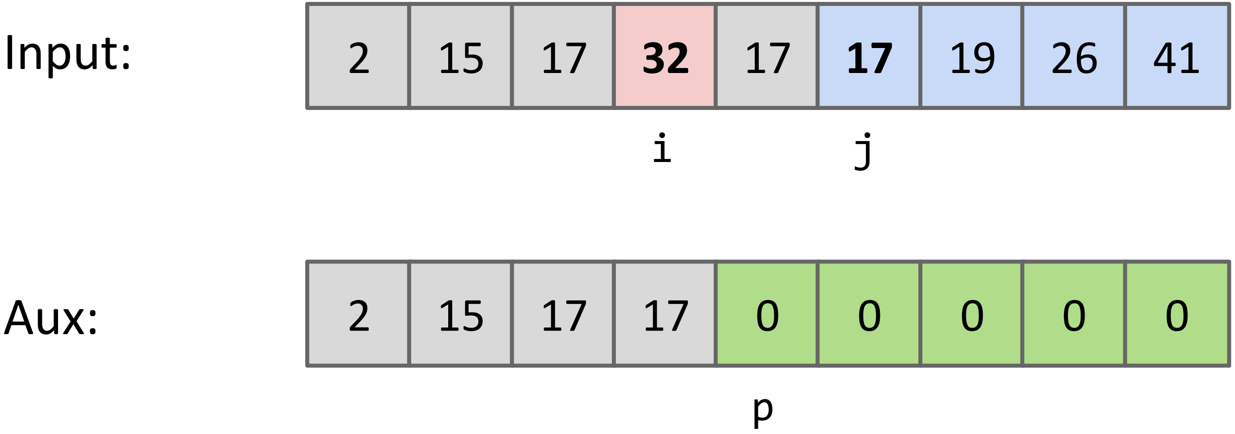
Input:	2	15	17	32	17	17	19	26	41
				i		j			

Aux:	2	15	17	17	0	0	0	0	0
					p				

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - **Compare input[i] < input[j] (if necessary).**
  - Copy smaller item and increment p and i or j.



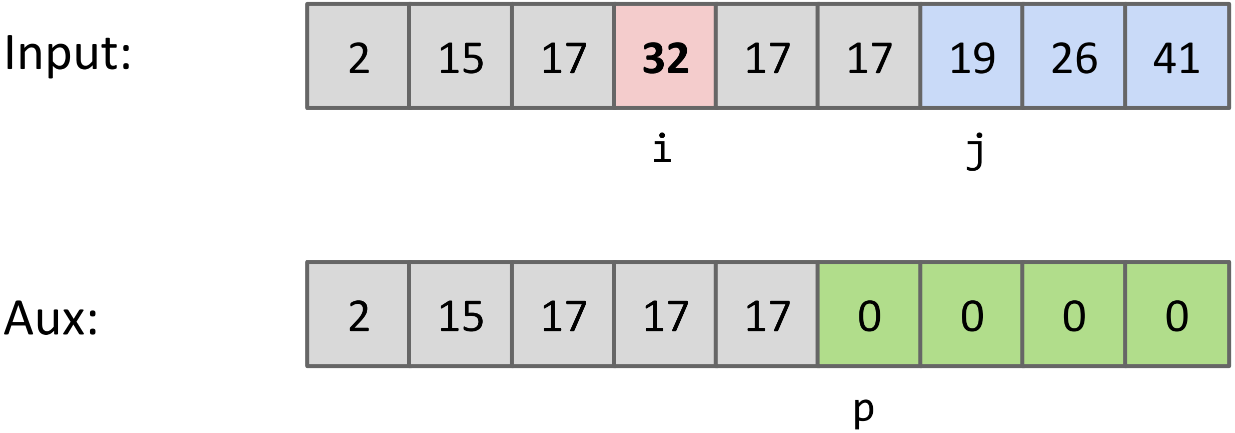
Input:	2	15	17	**32**	17	**17**	19	26	41
				i		j			

Aux:	2	15	17	17	0	0	0	0	0
					p				

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**



Input:

2	15	17	**32**	17	17	19	26	41
			i			j		

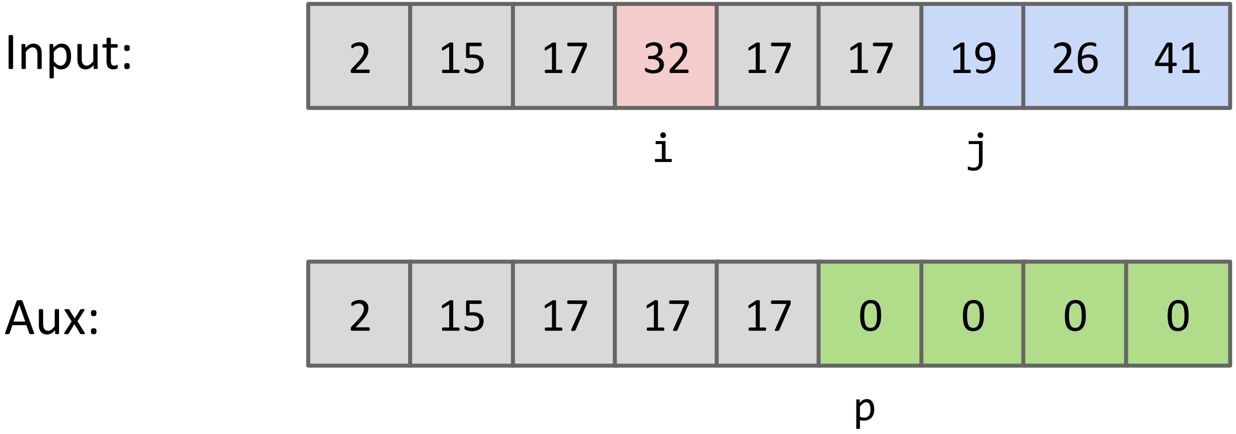
Aux:

2	15	17	17	17	0	0	0	0
					p			

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**

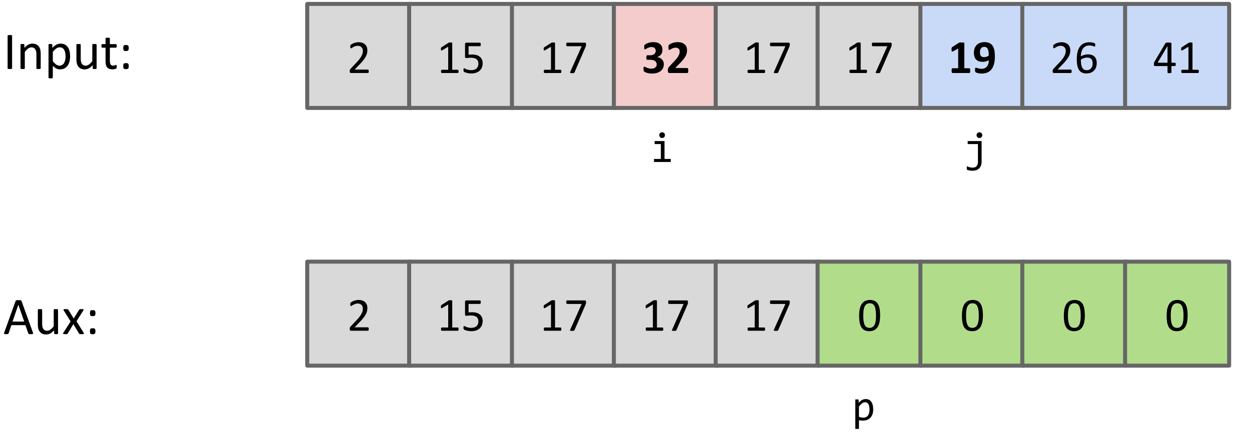


Input:	2	15	17	32	17	17	19	26	41
				i			j		
Aux:	2	15	17	17	17	0	0	0	0
						p			

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - **Compare input[i] < input[j] (if necessary).**
  - Copy smaller item and increment p and i or j.

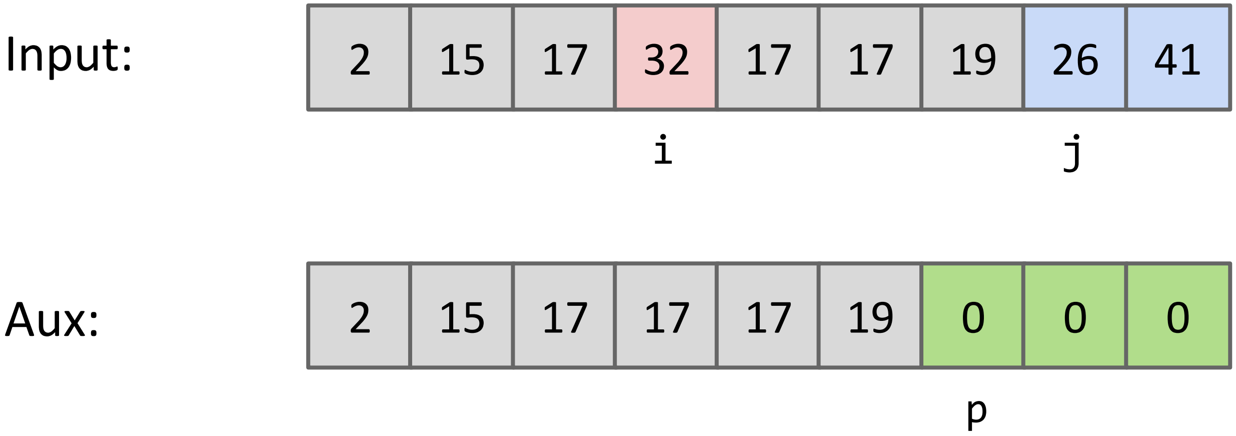


Input:	2	15	17	**32**	17	17	**19**	26	41
				i			j		
Aux:	2	15	17	17	17	0	0	0	0
						p			

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**

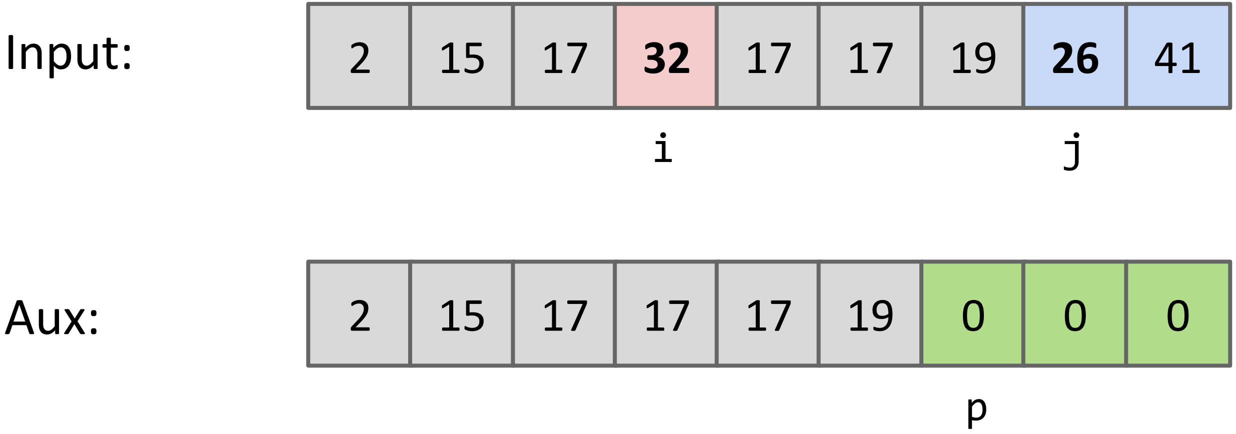


Input:	2	15	17	32	17	17	19	26	41
				i				j	
Aux:	2	15	17	17	17	19	0	0	0
							p		

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - **Compare input[i] < input[j] (if necessary).**
    - Copy smaller item and increment p and i or j.

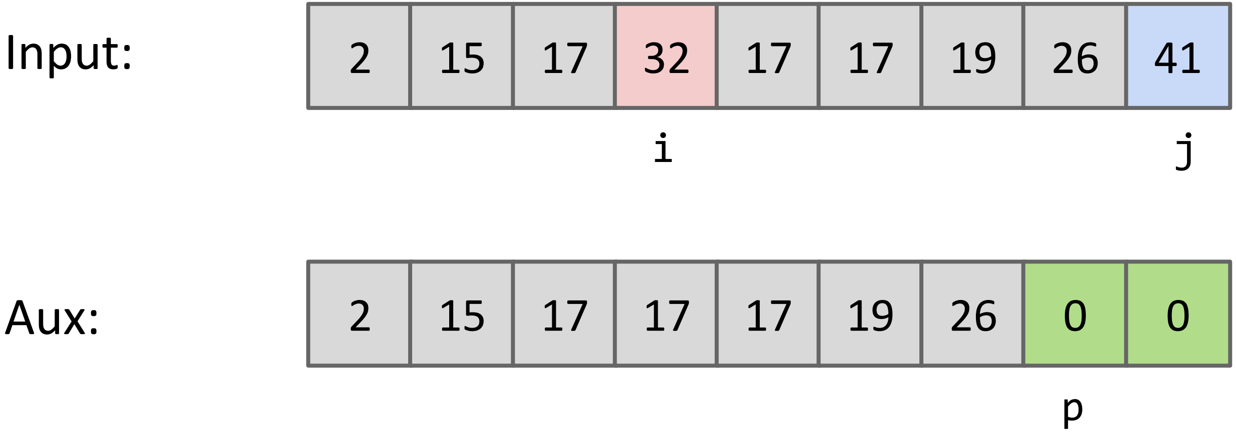


Input:	2	15	17	**32**	17	17	19	**26**	41
				i				j	
Aux:	2	15	17	17	17	19	0	0	0
							p		

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**

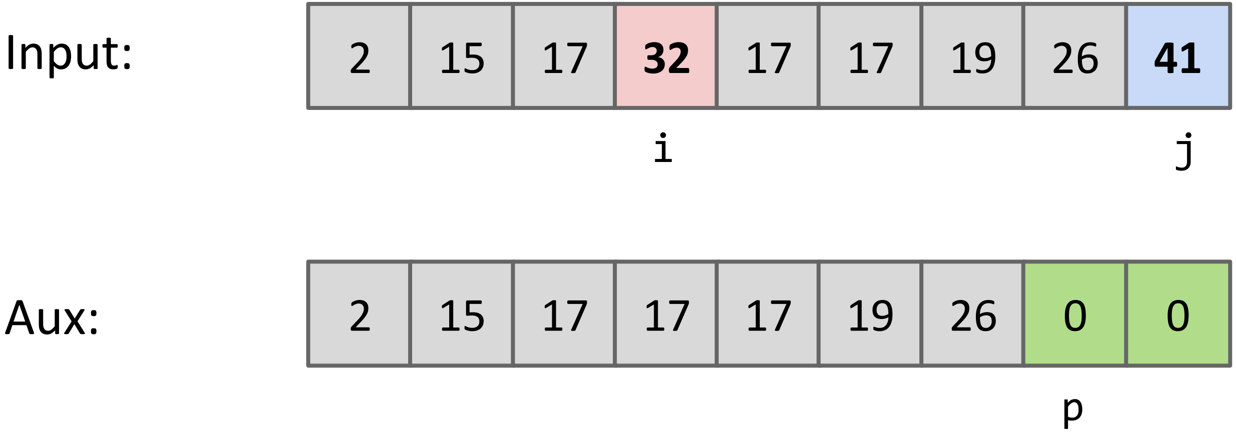


Input:	2	15	17	32	17	17	19	26	41
Aux:	2	15	17	17	17	19	26	0	0

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - **Compare input[i] < input[j] (if necessary).**
  - Copy smaller item and increment p and i or j.



Input:	2	15	17	**32**	17	17	19	26	**41**
				i					j
Aux:	2	15	17	17	17	19	26	0	0
								p	

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - **Compare input[i] < input[j] (if necessary).**
    - Copy smaller item and increment p and i or j.

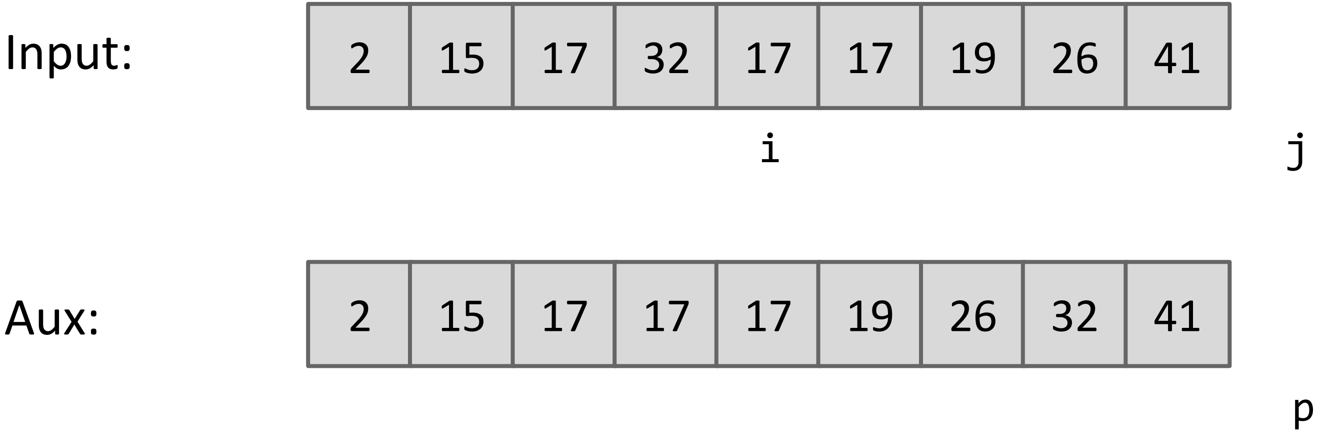
No comparison is made this time, since the left side has run out of items!

Input:	2	15	17	32	17	17	19	26	**41**
					i				j
Aux:	2	15	17	17	17	19	26	32	0
									p

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**



Input:	2	15	17	32	17	17	19	26	41
Aux:	2	15	17	17	17	19	26	32	41

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	Faster than heap sort.

*: An array of all duplicates yields linear runtime for heapsort.

**: Assumes heap operations implemented iteratively, not recursively.

Naive Insertion Sort

Lecture 30, CS61B, Spring 2024

Mergesort

Insertion Sort

- **Naive Insertion Sort**
- In-Place Insertion Sort
- Insertion Sort Runtime

Quicksort

- Quicksort Backstory, Partitioning
- Quicksort

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:

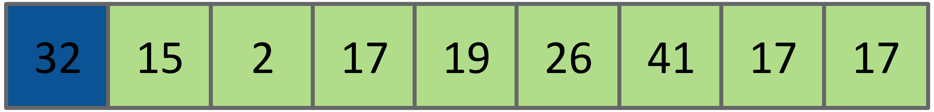
Insertion Sort

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:

Input:



32	15	2	17	19	26	41	17	17

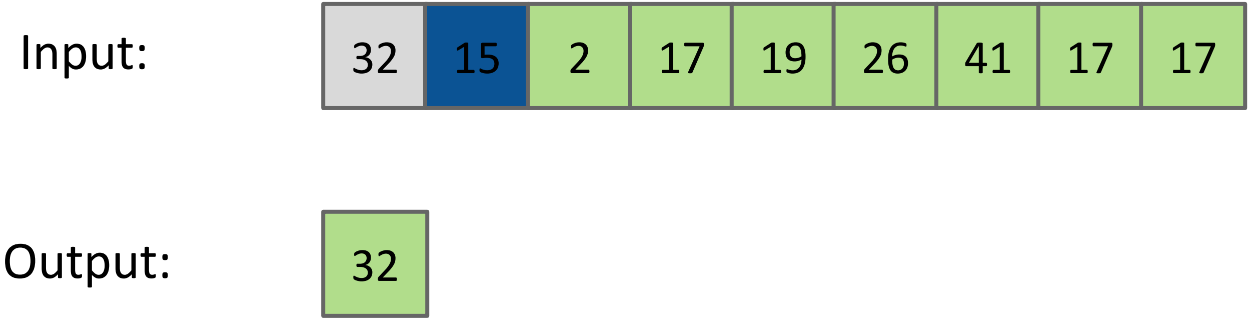
Output:

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:



Input:

32	15	2	17	19	26	41	17	17

Output:

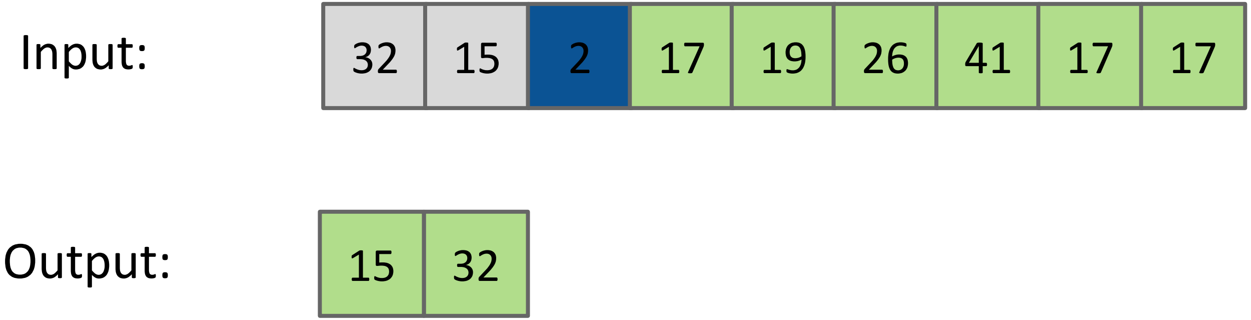
32

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:



Input: 32 15 2 17 19 26 41 17 17

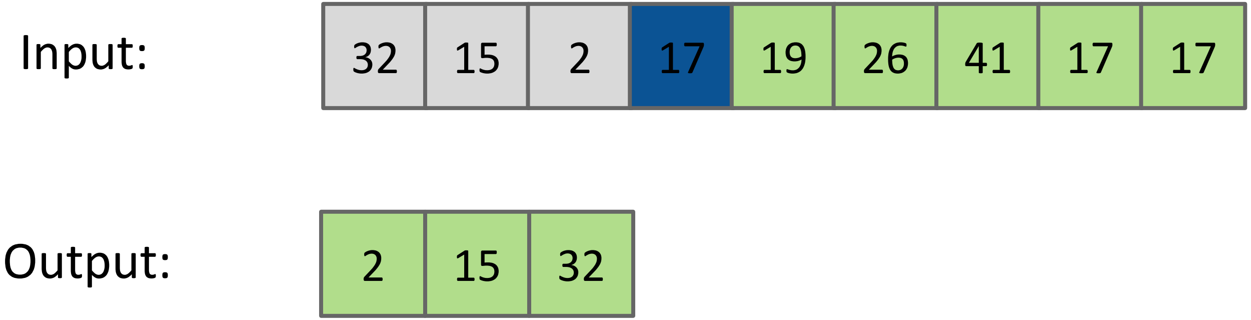
Output: 15 32

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:



Input: 32 15 2 17 19 26 41 17 17

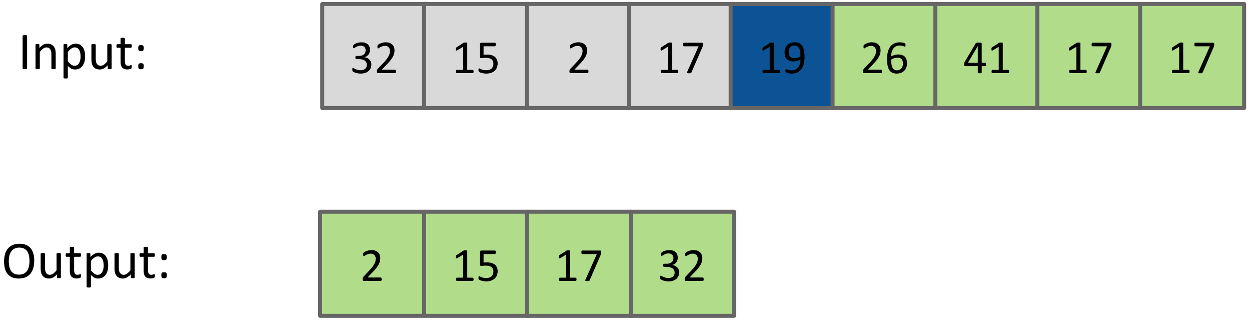
Output: 2 15 32

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:



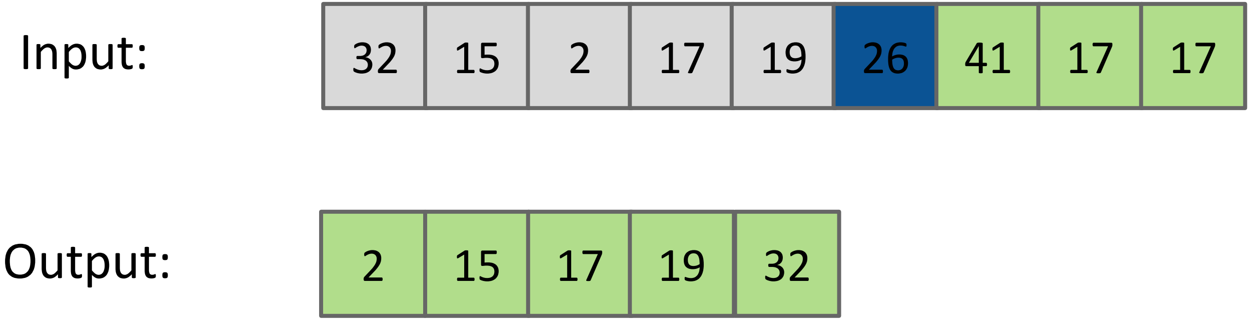
Input:	32	15	2	17	19	26	41	17	17
Output:	2	15	17	32					

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:



Input: 32 15 2 17 19 26 41 17 17

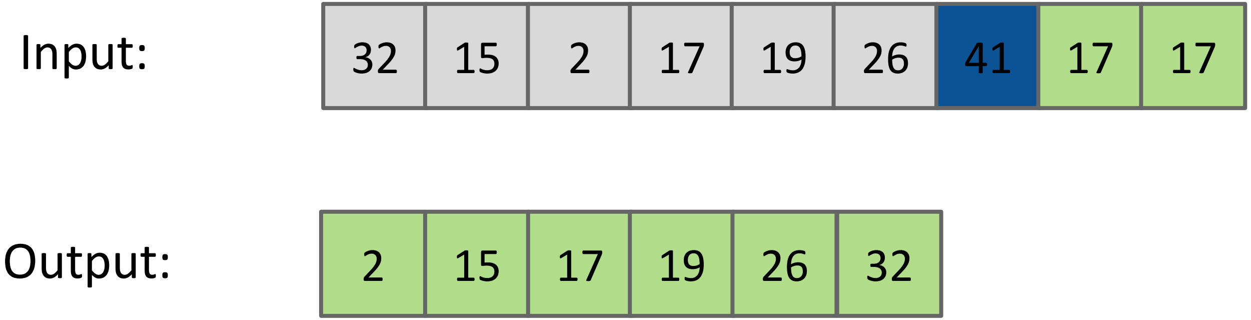
Output: 2 15 17 19 32

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:



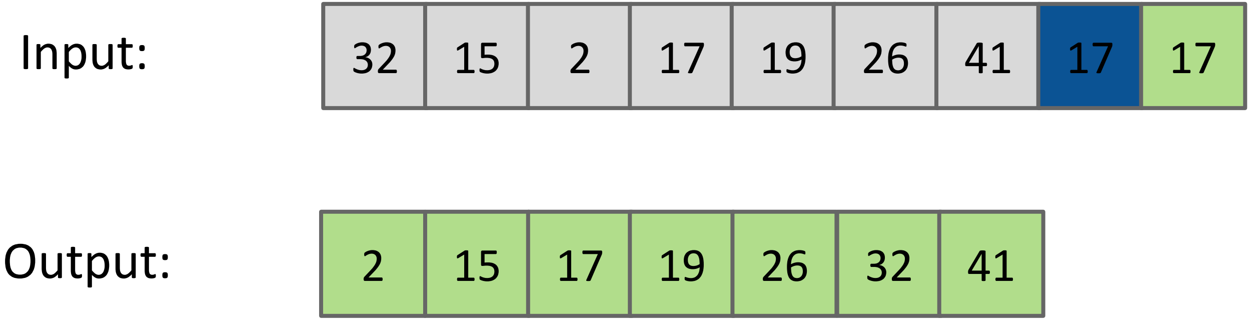
Input:	32	15	2	17	19	26	41	17	17
Output:	2	15	17	19	26	32			

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:



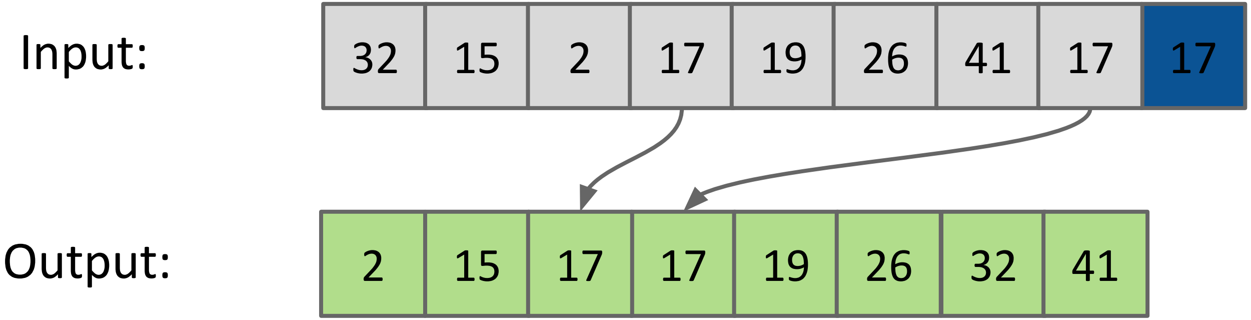
Input:	32	15	2	17	19	26	41	17	17
Output:	2	15	17	19	26	32	41		

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:



Input: 32 15 2 17 19 26 41 17 17

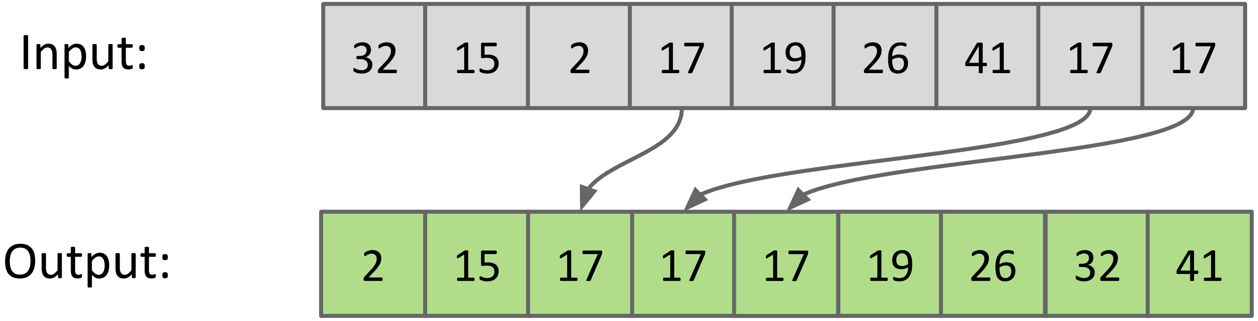
Output: 2 15 17 17 19 26 32 41

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:



Input: 32 15 2 17 19 26 41 17 17

Output: 2 15 17 17 17 19 26 32 41

In-Place Insertion Sort

Lecture 30, CS61B, Spring 2024

Mergesort

Insertion Sort

- Naive Insertion Sort
- **In-Place Insertion Sort**
- Insertion Sort Runtime

Quicksort

- Quicksort Backstory, Partitioning
- Quicksort

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

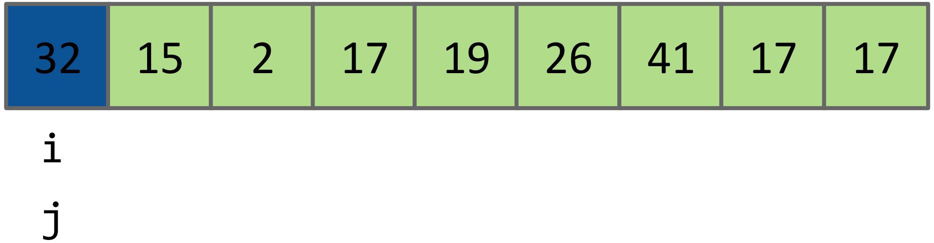
In example above: Use j pointer to track current spot of traveling item.

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:



32	15	2	17	19	26	41	17	17
i								
j								

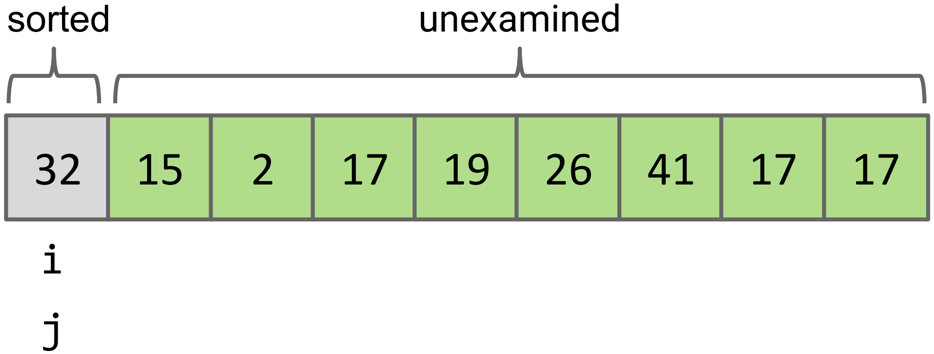
In example above: Use j pointer to track current spot of traveling item.

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:



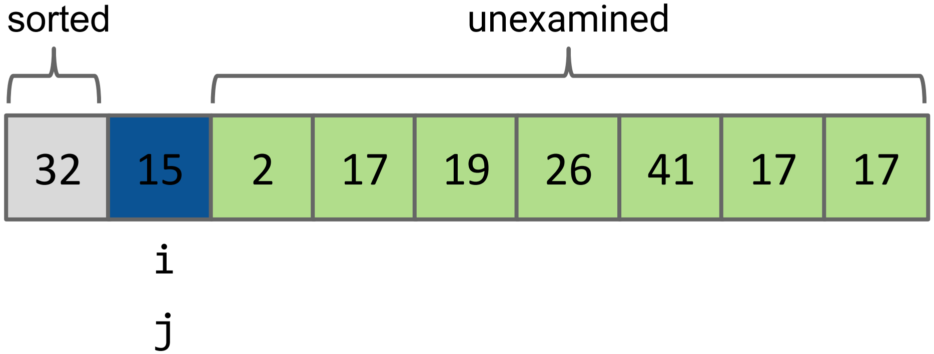
In example above: Use j pointer to track current spot of traveling item.

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:



sorted

unexamined

32	15	2	17	19	26	41	17	17

i

j

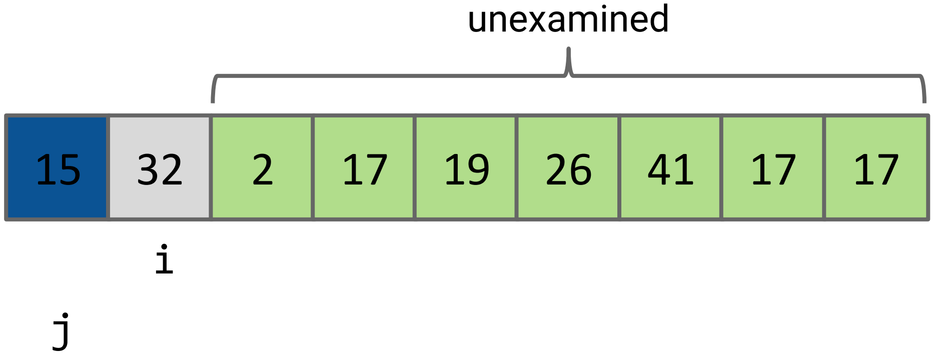
In example above: Use j pointer to track current spot of traveling item.

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:



15	32	2	17	19	26	41	17	17

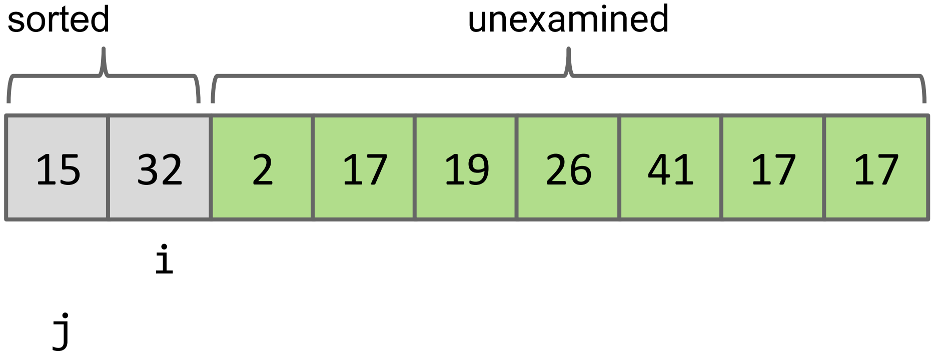
In example above: Use j pointer to track current spot of traveling item.

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:



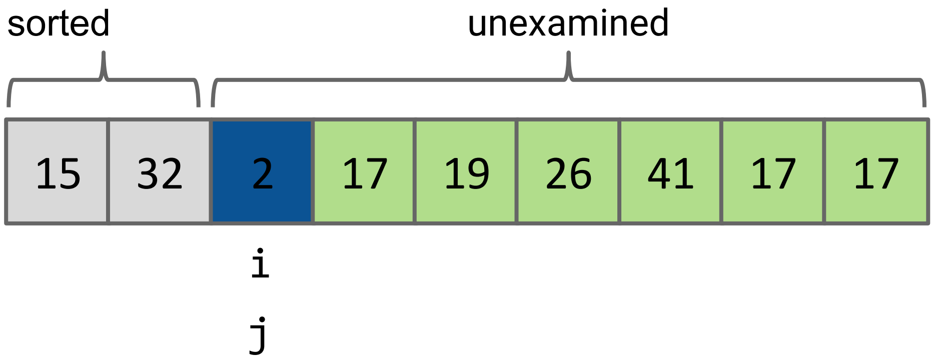
In example above: Use j pointer to track current spot of traveling item.

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:



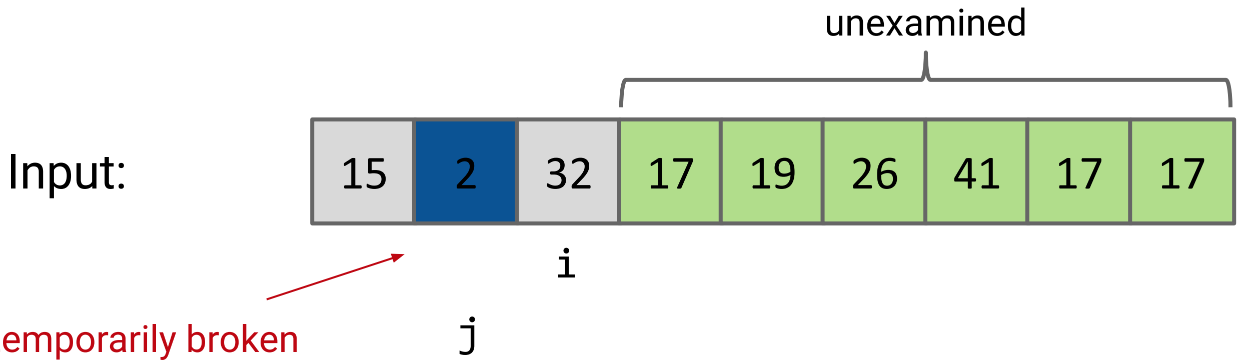
15	32	2	17	19	26	41	17	17

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.**



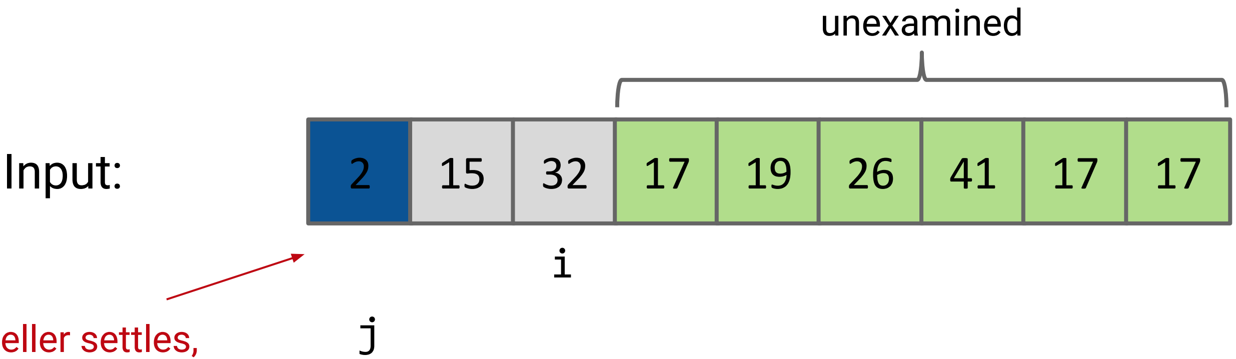
Note: We've temporarily broken our invariant that the items up through item i should be sorted!

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.

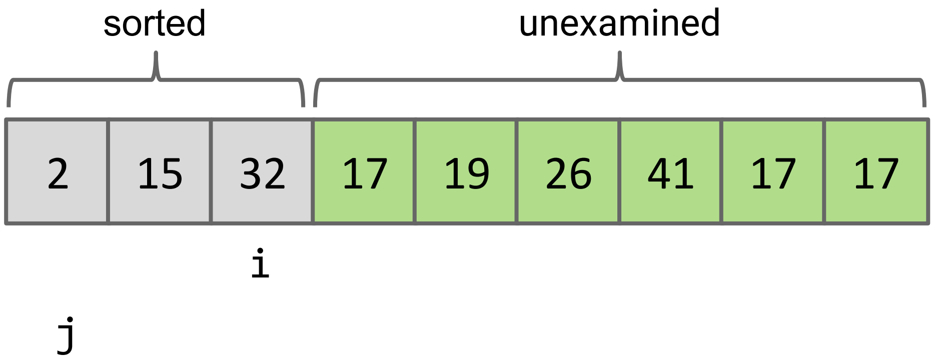
In example above: Use j pointer to track current spot of traveling item.

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:



sorted

unexamined

2	15	32	17	19	26	41	17	17

i

j

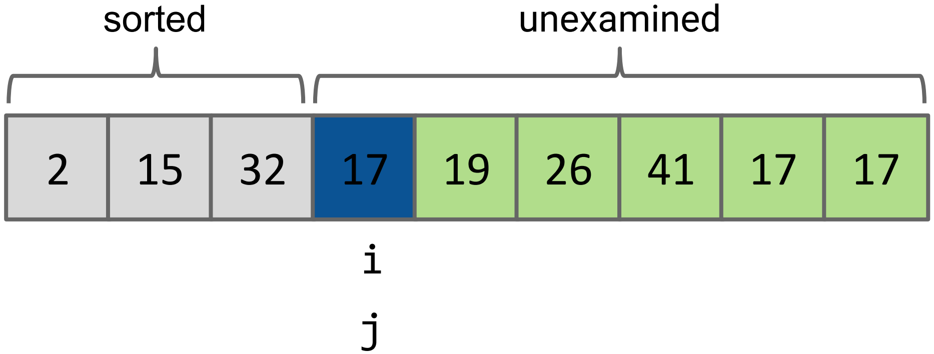
In example above: Use j pointer to track current spot of traveling item.

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:



2	15	32	17	19	26	41	17	17

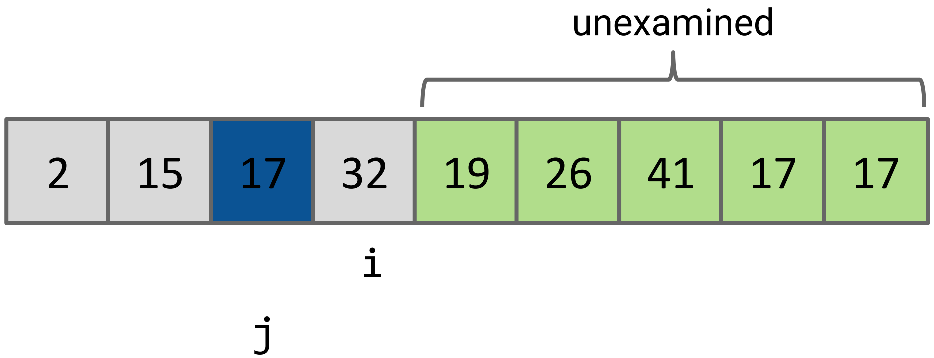
In example above: Use j pointer to track current spot of traveling item.

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:

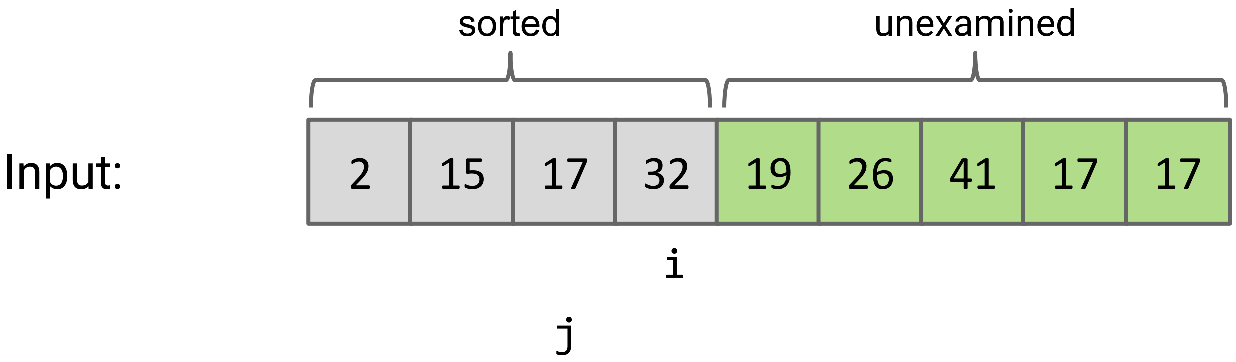


In example above: Use j pointer to track current spot of traveling item.

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.**



sorted

unexamined

Input:

2	15	17	32	19	26	41	17	17

i

j

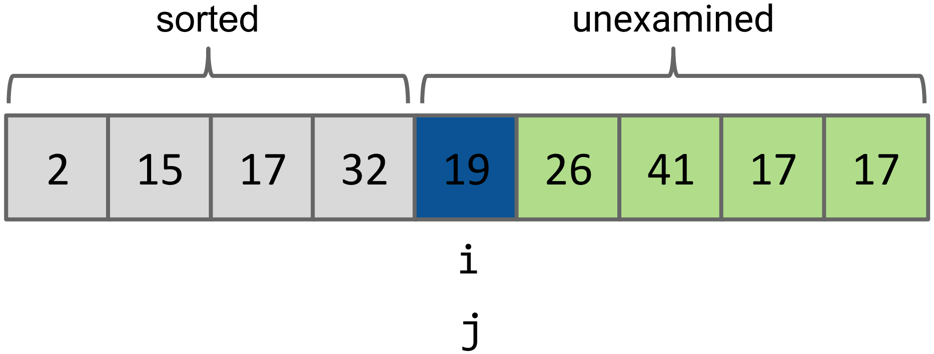
In example above: Use j pointer to track current spot of traveling item.

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:



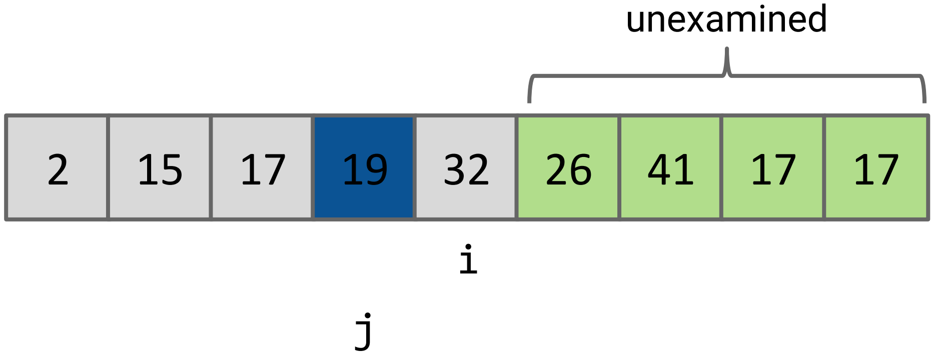
In example above: Use j pointer to track current spot of traveling item.

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:

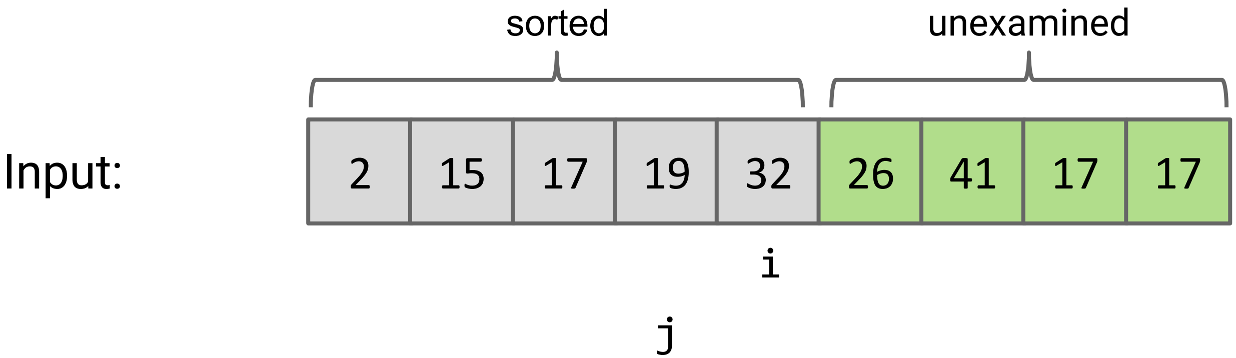


In example above: Use j pointer to track current spot of traveling item.

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.**



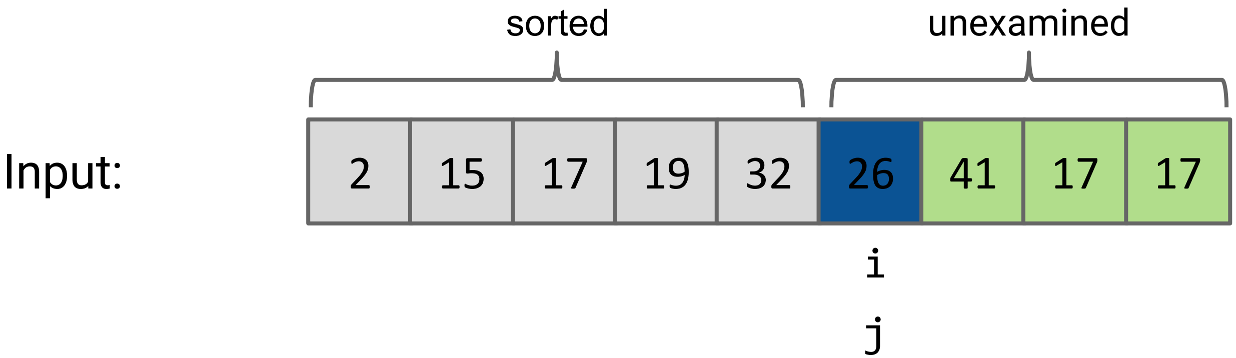
Input:	2	15	17	19	32	26	41	17	17

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.



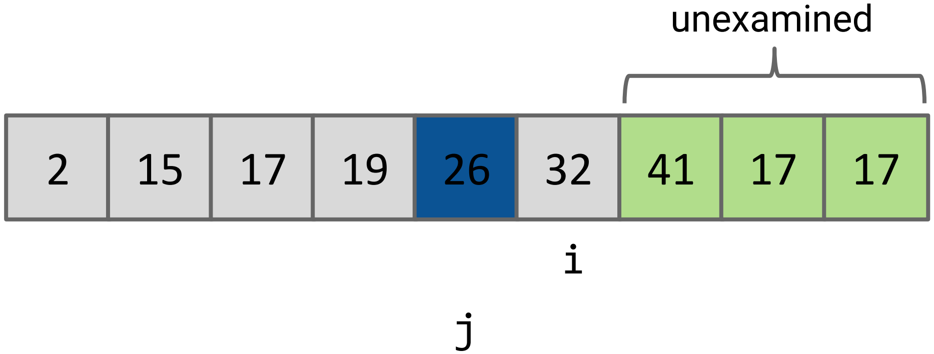
In example above: Use j pointer to track current spot of traveling item.

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:



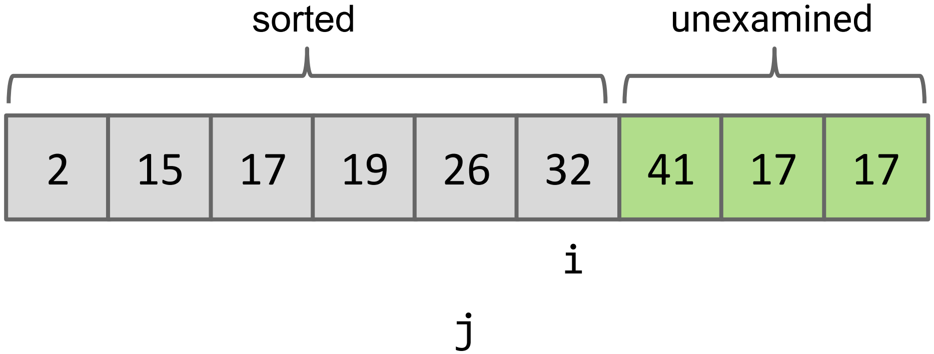
In example above: Use j pointer to track current spot of traveling item.

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:



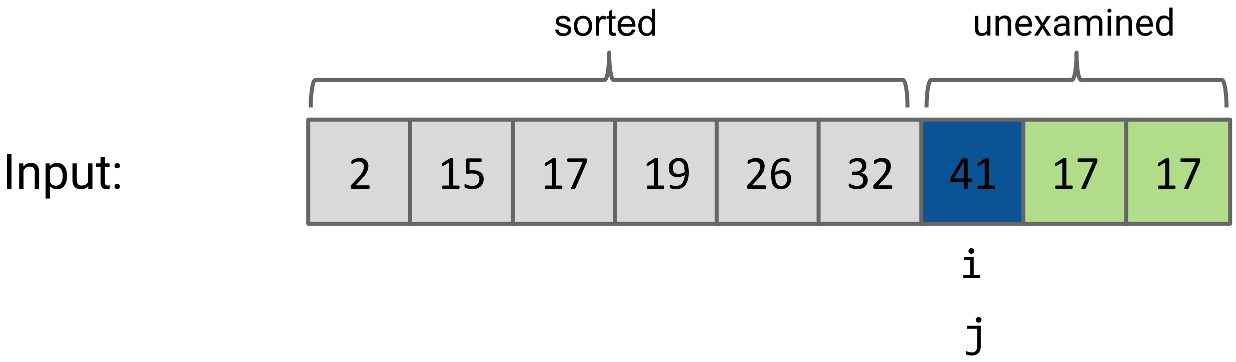
2	15	17	19	26	32	41	17	17

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.



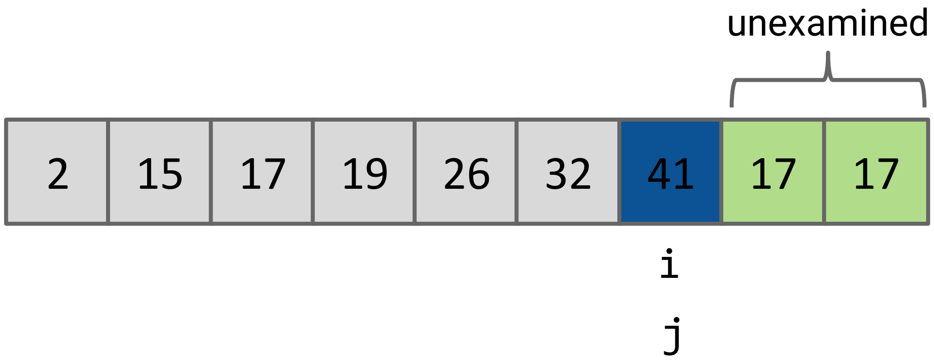
In example above: Use j pointer to track current spot of traveling item.

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:



2	15	17	19	26	32	41	17	17

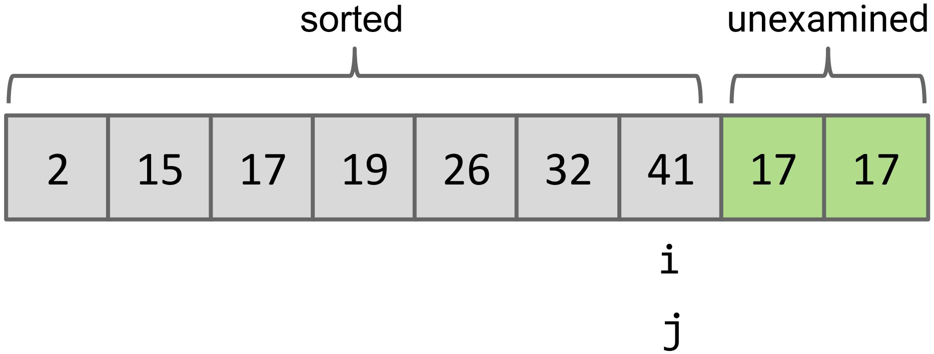
In example above: Use j pointer to track current spot of traveling item.

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:



2	15	17	19	26	32	41	17	17

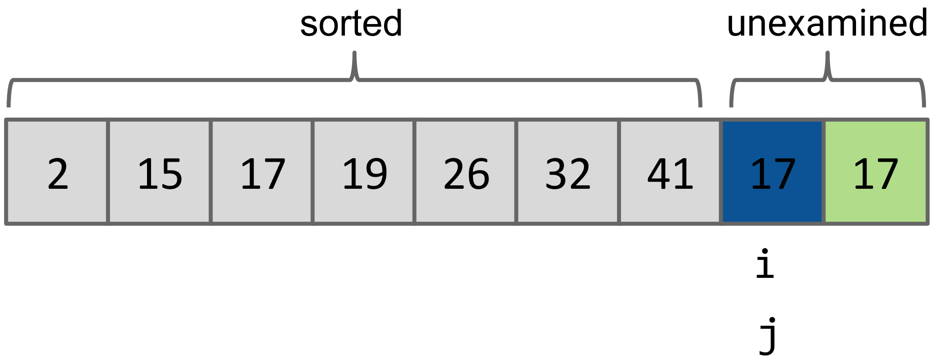
In example above: Use j pointer to track current spot of traveling item.

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:



sorted

unexamined

2	15	17	19	26	32	41	17	17

i

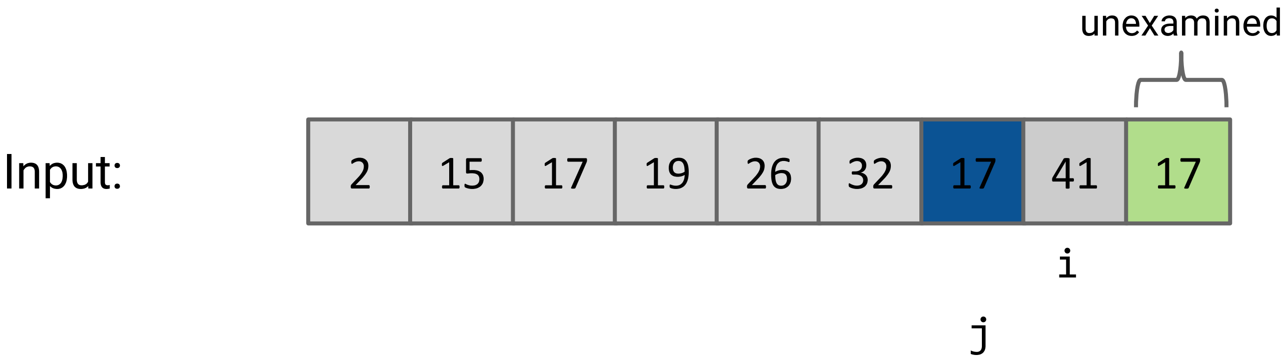
j

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.**

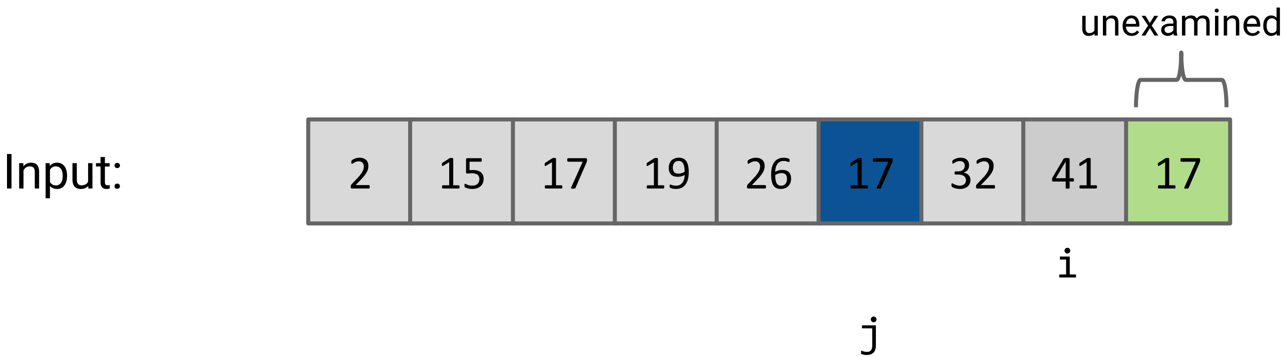


In example above: Use j pointer to track current spot of traveling item.

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.**

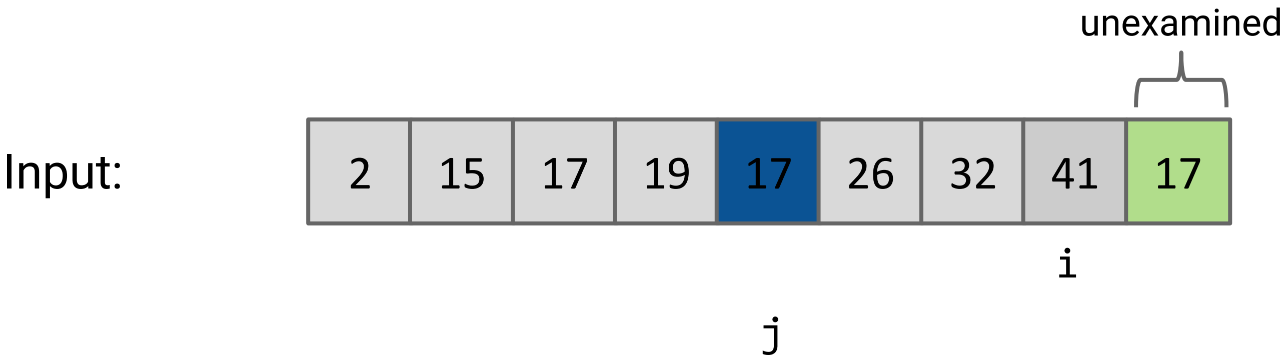


In example above: Use j pointer to track current spot of traveling item.

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.**



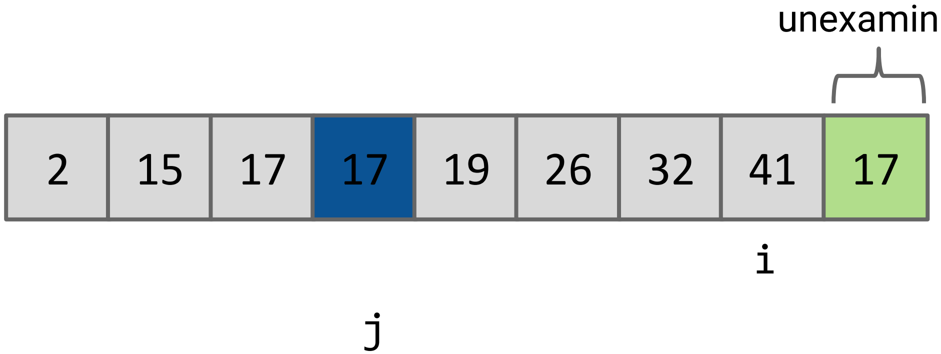
In example above: Use j pointer to track current spot of traveling item.

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:



2	15	17	17	19	26	32	41	17

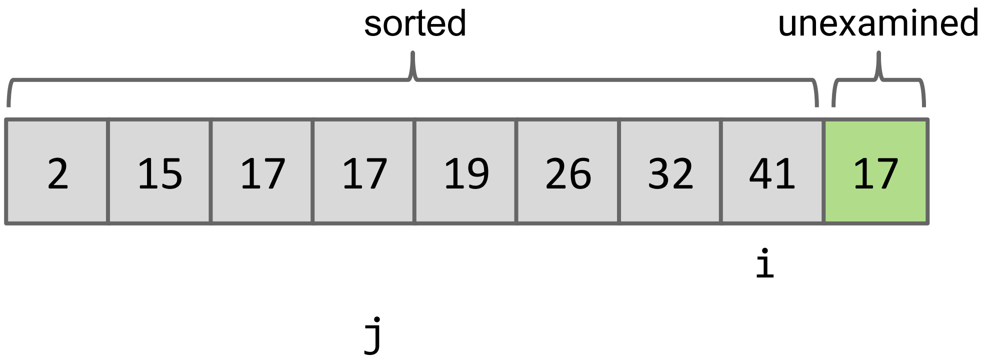
In example above: Use j pointer to track current spot of traveling item.

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:



sorted

unexamined

2	15	17	17	19	26	32	41	17

i

j

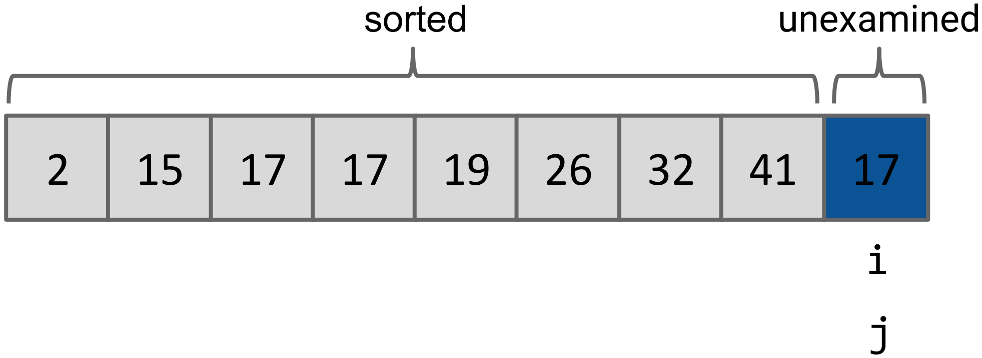
In example above: Use j pointer to track current spot of traveling item.

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:



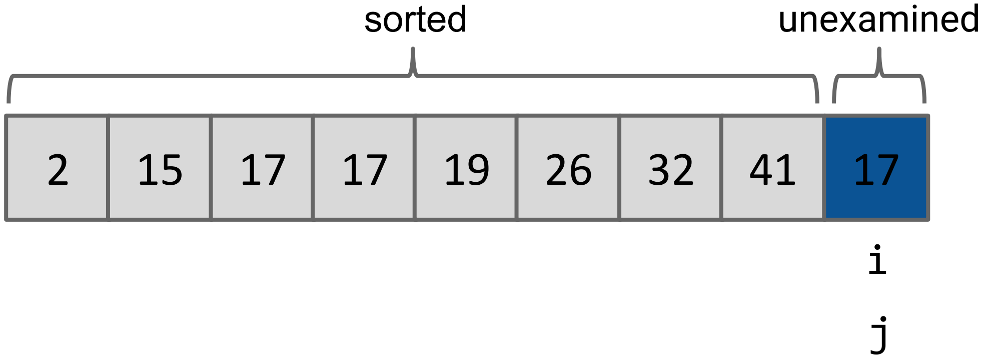
In example above: Use j pointer to track current spot of traveling item.

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:



2	15	17	17	19	26	32	41	17

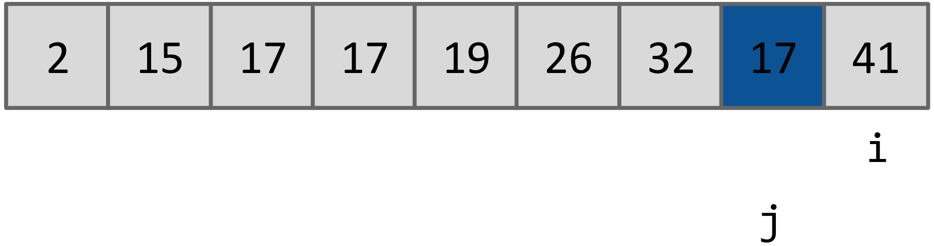
In example above: Use j pointer to track current spot of traveling item.

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:



2	15	17	17	19	26	32	17	41

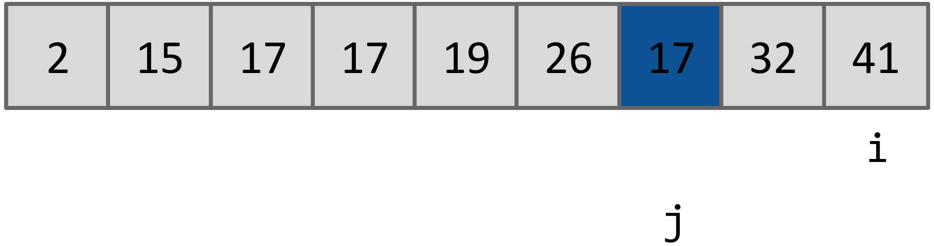
In example above: Use j pointer to track current spot of traveling item.

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:



2	15	17	17	19	26	**17**	32	41

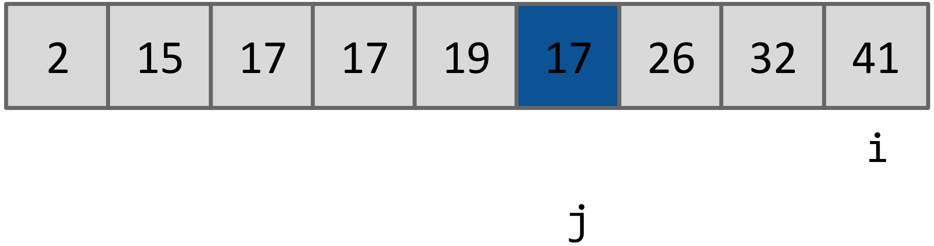
In example above: Use j pointer to track current spot of traveling item.

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:



2	15	17	17	19	17	26	32	41

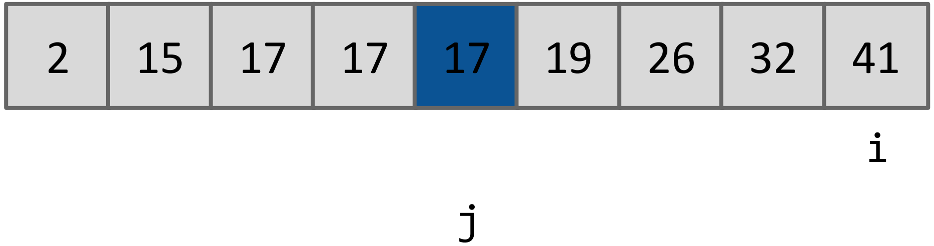
In example above: Use j pointer to track current spot of traveling item.

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:



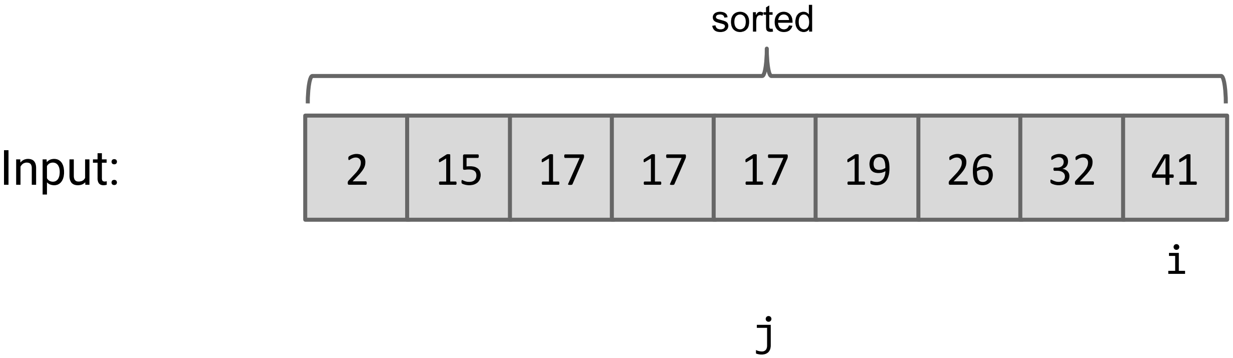
2	15	17	17	17	19	26	32	41
								i
				j				

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:

2	15	17	17	17	19	26	32	41

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

Insertion Sort Runtime

Lecture 30, CS61B, Spring 2024

Mergesort

Insertion Sort

- Naive Insertion Sort
- In-Place Insertion Sort
- **Insertion Sort Runtime**

Quicksort

- Quicksort Backstory, Partitioning
- Quicksort

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)
```

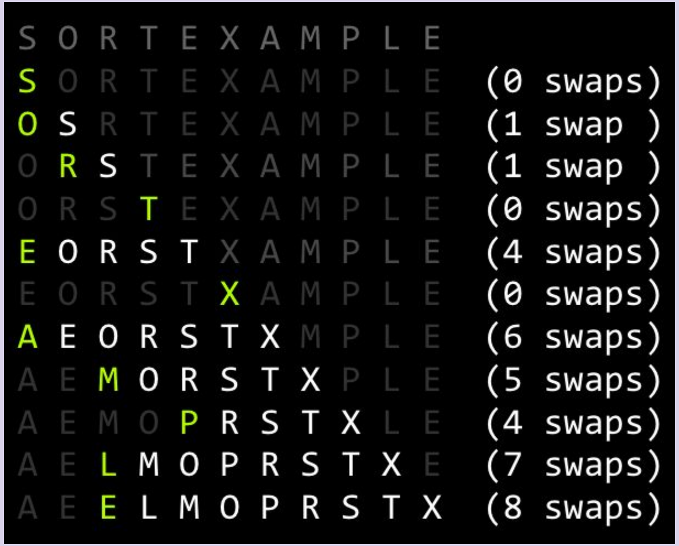
Purple: Element that we're moving left (with swaps).
Black: Elements that got swapped with purple.
Grey: Not considered this iteration.

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

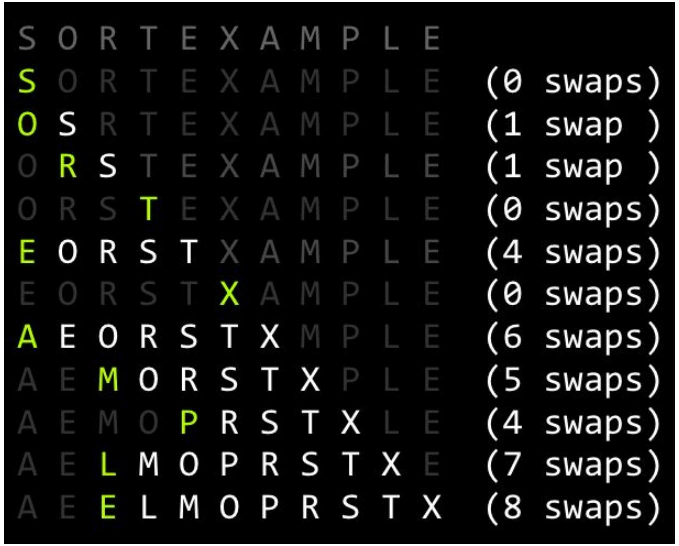
What is the runtime of insertion sort?

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 Ω(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.

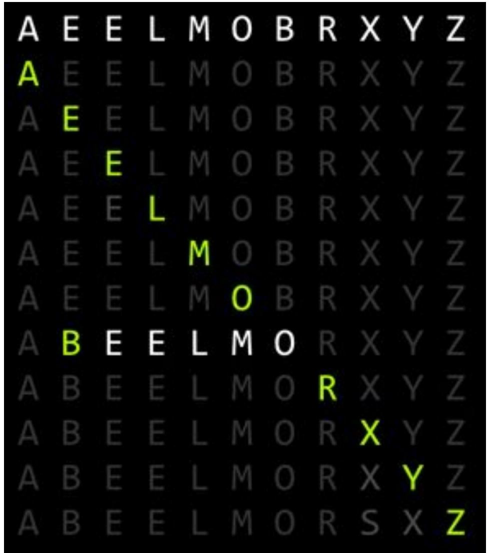
Which sorting algorithm would be the fastest choice for X?

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

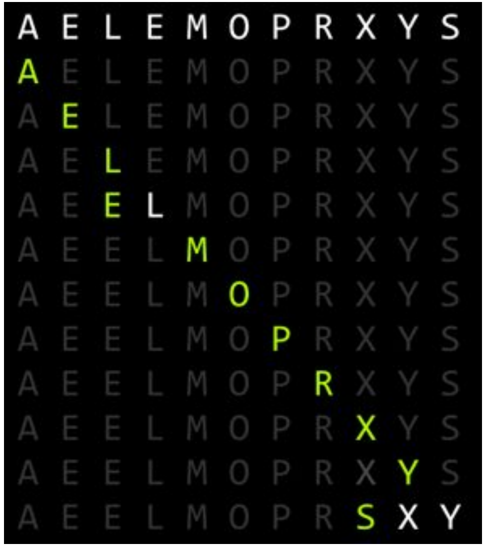
Observation: Insertion Sort on Almost Sorted Arrays

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: Θ(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 Θ(N + K) where K is number of inversions.
- Define an ***almost sorted*** array as one in which number of inversions ≤ 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.

Quicksort Backstory, Partitioning

Lecture 30, CS61B, Spring 2024

Mergesort

Insertion Sort

- Naive Insertion Sort
- In-Place Insertion Sort
- Insertion Sort Runtime

Quicksort

- **Quicksort Backstory, Partitioning**
- Quicksort

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.

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:

- Much stranger 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!)...

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

To partition an array a[] on element x=a[i] is to rearrange a[] so that:

- x moves to position j (may be the same as i)
- All entries to the left of x are <= x.
- All entries to the right of x are >= x.

Called the 'pivot'.

		i				
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

To partition an array a[] on element x=a[i] is to rearrange a[] so that:

- x moves to position j (may be the same as i)
- All entries to the left of x are <= x.
- All entries to the right of x are >= x.

Called the 'pivot'.

		i				
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?

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 2024

Mergesort

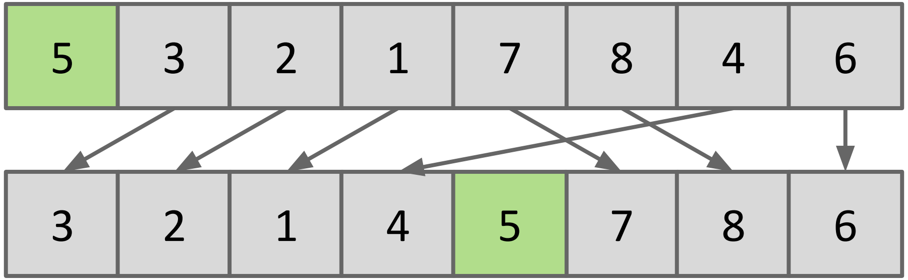
Insertion Sort

- Naive Insertion Sort
- In-Place Insertion Sort
- Insertion Sort Runtime

Quicksort

- Quicksort Backstory, Partitioning
- **Quicksort**

Partition Sort, a.k.a. Quicksort



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

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.

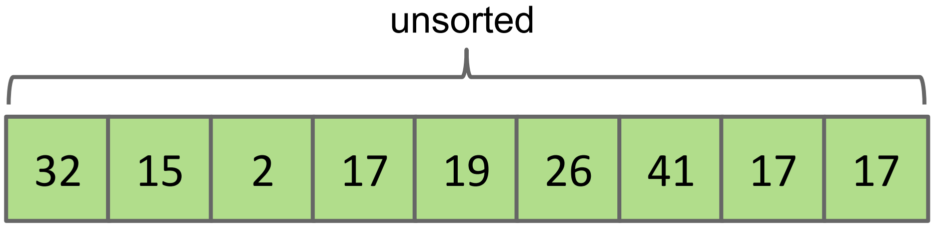
3	2	1	4	5
2	1	3	4	5

5	7	8	6
5	6	7	8

Quick Sort

Quick sorting N items:

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



unsorted

Input:	32	15	2	17	19	26	41	17	17

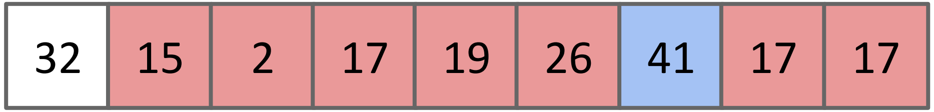
Quick Sort

Quick sorting N items:

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

partition(32)

Input:



32	15	2	17	19	26	41	17	17

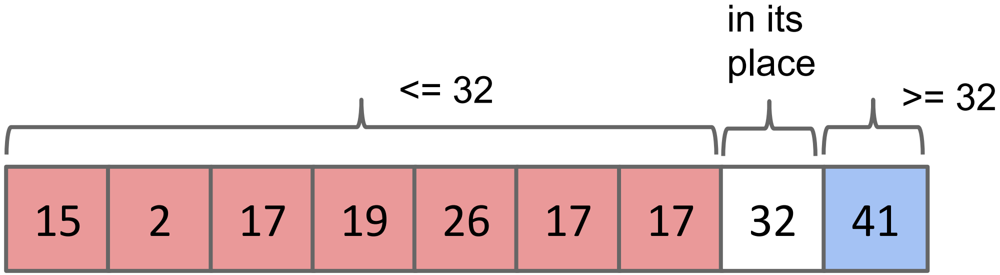
Quick Sort

Quick sorting N items:

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

partition(32)

Input:



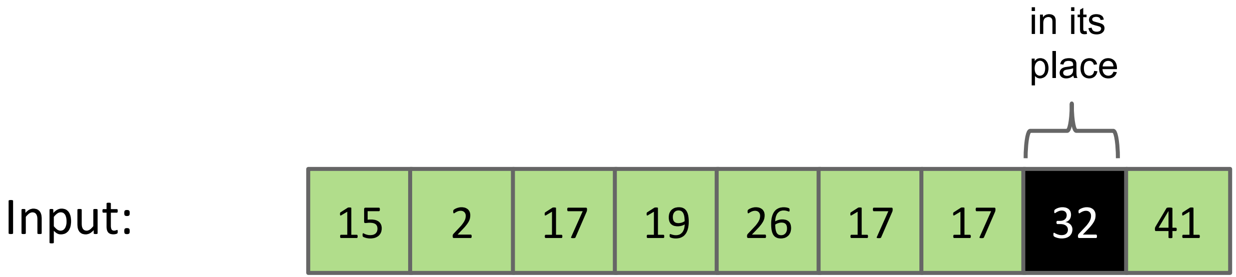
15	2	17	19	26	17	17	32	41

Quick Sort

Quick sorting N items:

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

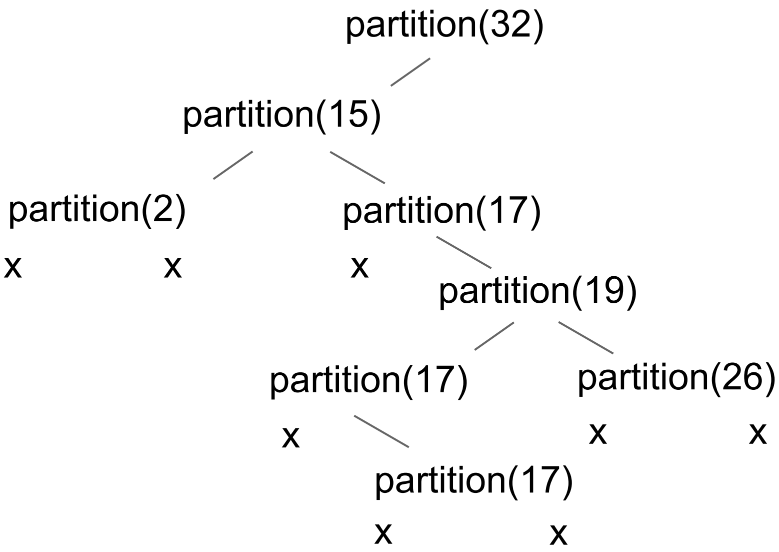
partition(32)



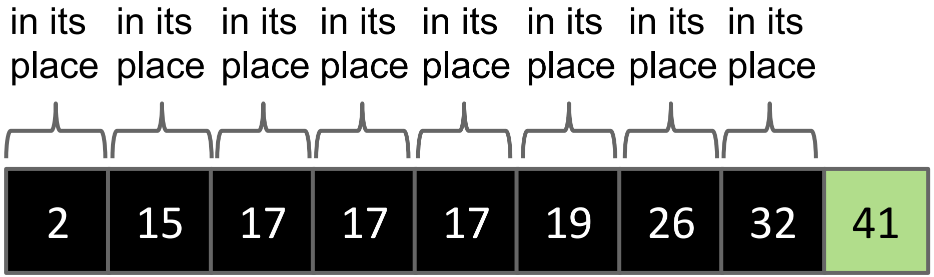
Quick Sort

Quick sorting N items:

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



Input:

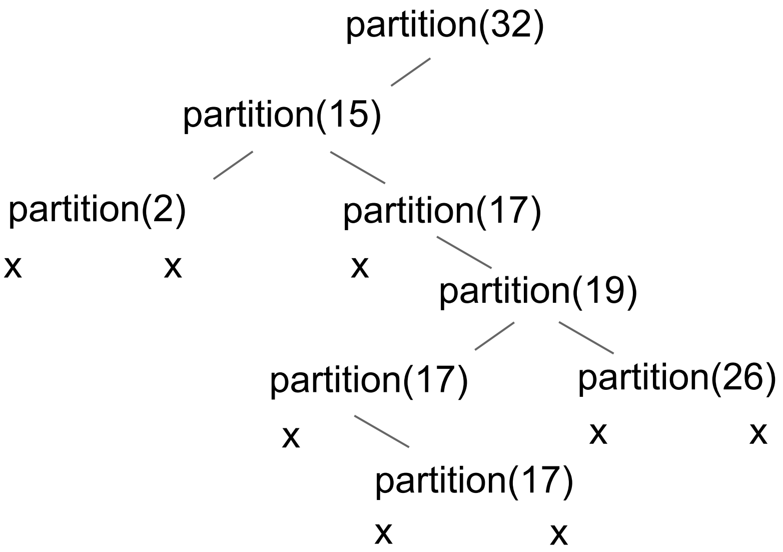


in its place	in its place	in its place	in its place	in its place	in its place	in its place	in its place	
2	15	17	17	17	19	26	32	41

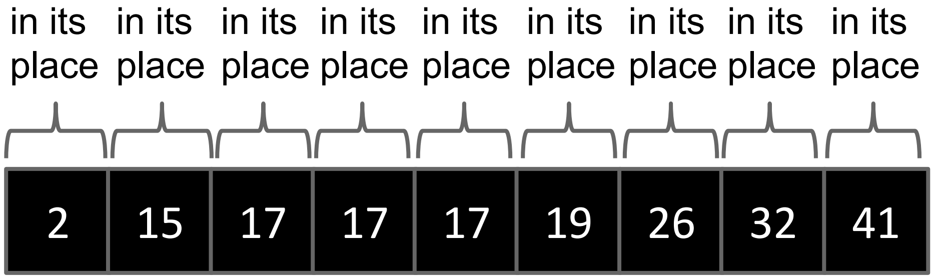
Quick Sort

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.



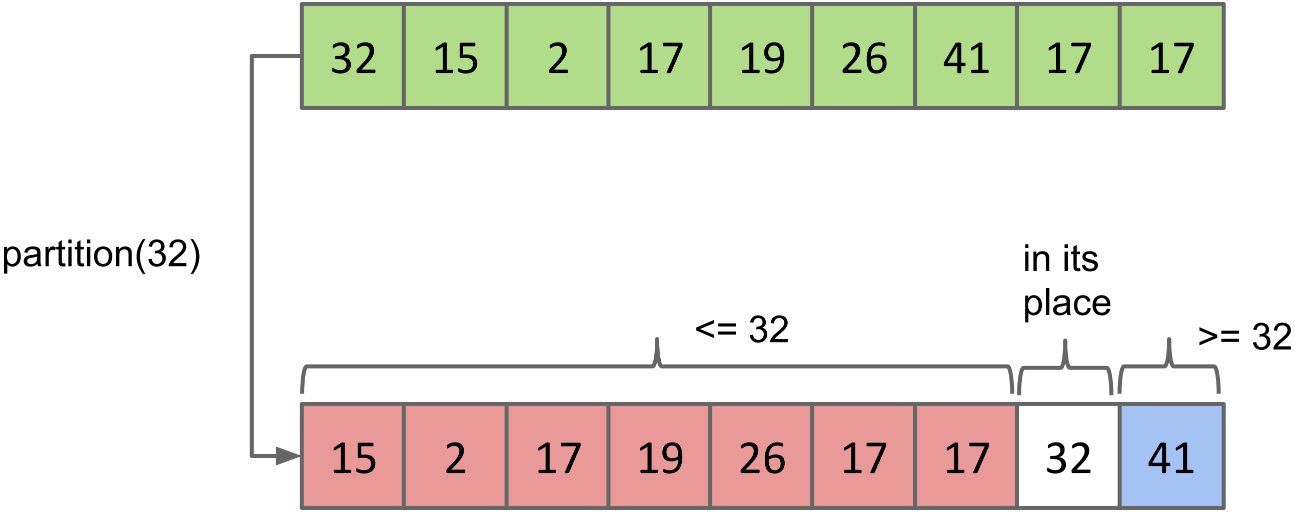
Input:

2	15	17	17	17	19	26	32	41

Partition Sort, a.k.a. Quicksort

Quick sorting N items:

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



32	15	2	17	19	26	41	17	17

15	2	17	19	26	17	17	32	41

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.