

# Sorting

---

## Discussion 11



# Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
					4/12 Lab 10 due	
	4/15 Project 3A due					



# Content Review

---



# Insertion Sort

**Insertion sort** iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4

Runtime:  $O(N^2)$



# Insertion Sort

**Insertion sort** iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4  
↑

Runtime:  $O(N^2)$



# Insertion Sort

**Insertion sort** iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4  
↑  
no change

Runtime:  $O(N^2)$



# Insertion Sort

**Insertion sort** iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4

List so far: 3 5 1 2 4

Runtime:  $O(N^2)$



# Insertion Sort

**Insertion sort** iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4

List so far: 3 5 1 2 4

↑

Runtime:  $O(N^2)$





# Insertion Sort

**Insertion sort** iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4

3 5 1 2 4

List so far: 1 3 5 2 4

Runtime:  $O(N^2)$



# Insertion Sort

**Insertion sort** iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4

3 5 1 2 4

List so far:

1 3 5 2 4



Runtime:  $O(N^2)$



# Insertion Sort

**Insertion sort** iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4

3 5 1 2 4

1 3 5 2 4

List so far: 1 2 3 5 4

Runtime:  $O(N^2)$



# Insertion Sort

**Insertion sort** iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4

3 5 1 2 4

1 3 5 2 4

List so far:

1 2 3 5 4



Runtime:  $O(N^2)$



# Insertion Sort

**Insertion sort** iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4

3 5 1 2 4

1 3 5 2 4

1 2 3 5 4

List so far:

1 2 3 4 5

Sorted!

Runtime:  $O(N^2)$



# Selection Sort

**Selection sort** finds the smallest remaining element in the unsorted portion of the list at each time step and swaps it into the correct position.

3 5 1 2 4

Runtime:  $\Theta(N^2)$



# Selection Sort

**Selection sort** finds the smallest remaining element in the unsorted portion of the list at each time step and swaps it into the correct position.

3 5 1 2 4  
└─────────┘  
unsorted

Runtime:  $\Theta(N^2)$



# Selection Sort

**Selection sort** finds the smallest remaining element in the unsorted portion of the list at each time step and swaps it into the correct position.

3 5 1 2 4

1 3 5 2 4



unsorted

Runtime:  $\Theta(N^2)$





# Selection Sort

**Selection sort** finds the smallest remaining element in the unsorted portion of the list at each time step and swaps it into the correct position.

3 5 1 2 4

1 3 5 2 4

1 2 3 5 4

  
unsorted

Runtime:  $\Theta(N^2)$



# Selection Sort

**Selection sort** finds the smallest remaining element in the unsorted portion of the list at each time step and swaps it into the correct position.

3 5 1 2 4

1 3 5 2 4

1 2 3 5 4

1 2 3 5 4



unsorted

Runtime:  $\Theta(N^2)$



# Selection Sort

**Selection sort** finds the smallest remaining element in the unsorted portion of the list at each time step and swaps it into the correct position.

3 5 1 2 4

1 3 5 2 4

1 2 3 5 4

1 2 3 5 4

1 2 3 4 5



unsorted

Runtime:  $\Theta(N^2)$



# Selection Sort

**Selection sort** finds the smallest remaining element in the unsorted portion of the list at each time step and swaps it into the correct position.

3 5 1 2 4

1 3 5 2 4

1 2 3 5 4

1 2 3 5 4

1 2 3 4 5    Sorted!

Runtime:  $\Theta(N^2)$



# Merge Sort

**Merge sort** splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.

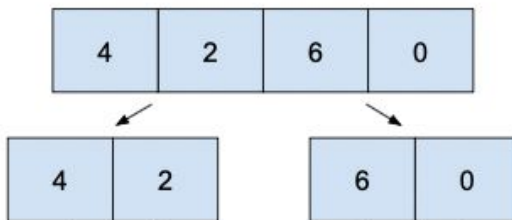
4	2	6	0
---	---	---	---

Runtime:  $\Theta(N \log N)$



# Merge Sort

**Merge sort** splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.

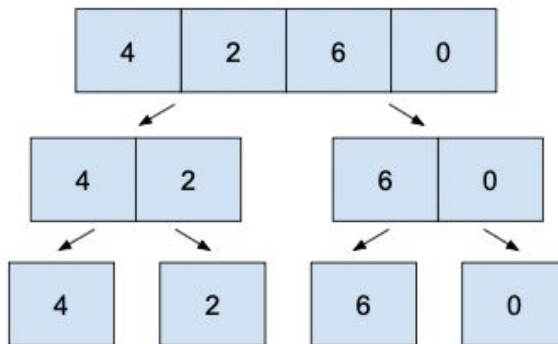


Runtime:  $\Theta(N \log N)$



# Merge Sort

**Merge sort** splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.

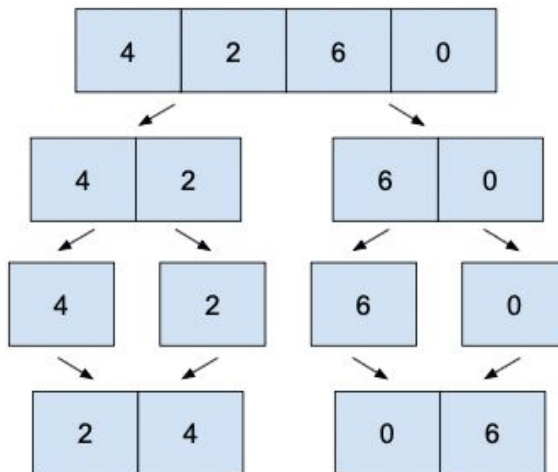


Runtime:  $\Theta(N \log N)$



# Merge Sort

**Merge sort** splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.



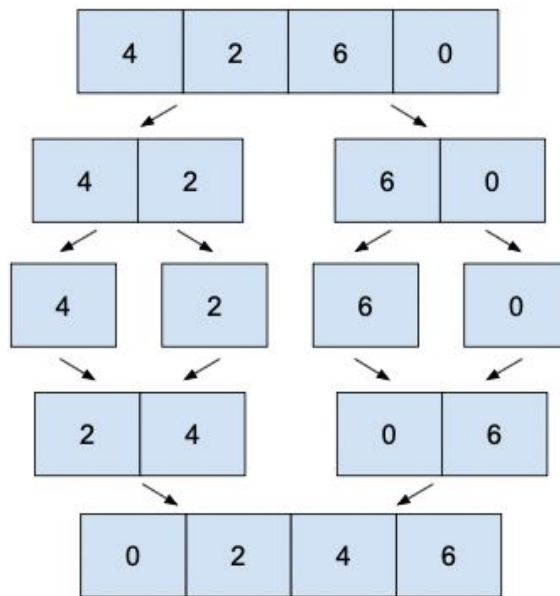
Runtime:  $\Theta(N \log N)$





# Merge Sort

**Merge sort** splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.



Runtime:  $\Theta(N \log N)$



# Quicksort

**Quicksort** picks a pivot (ie. first element) and uses Hoare partitioning to divide the list so that everything greater than the pivot is on its right and everything less than the pivot is on its left.

3 5 1 2 4

Runtime: Average case  $O(N \log N)$ , slowest case  $O(N^2)$  (dependent on pivot selection)



# Heap Sort

**Heapsort** heapifies the array into a max heap and pops the largest element off and appends it to the end until there are no elements left in the heap. You can heapify by sinking nodes in reverse level order.

3 5 1 2 4

Runtime:  $O(N \log N)$



# Summary for comparison sorts

**Stability:** a sort is stable if duplicate values remain in the same relative order after sorting as they were initially. In other words, is 2a guaranteed to be before 2b after sorting the list [2a, 2b, 1]?

	Worst Case	Best Case	Stable?
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	No
Insertion Sort	$\Theta(N^2)$	$\Theta(N)$	Yes
Merge Sort	$\Theta(N \log N)$	$\Theta(N \log N)$	Yes
Quicksort	$\Theta(N^2)$	$\Theta(N \log N)$	No*
Heapsort	$\Theta(N \log N)$	$\Theta(N)$	No

Try reasoning out or coming up with examples for these best and worst case runtimes!

\*depends on the partitioning scheme



# Worksheet

---



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 4 2 7 6 1 3 5

Green = sorted portion of array. **Green** = element we're currently moving back.



# 1A All Sorts of Sorts

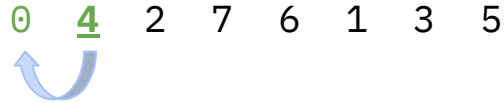
Show the steps taken by insertion sort on this list.

0 4 2 7 6 1 3 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.






# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.


0 4 2 7 6 1 3 5



# 1A All Sorts of Sorts

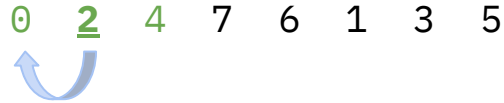
Show the steps taken by insertion sort on this list.

0 2 4 7 6 1 3 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 2 4 7 6 1 3 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 2 4 6 7 1 3 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 2 4 6 7 1 3 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 2 4 6 1 7 3 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 2 4 1 6 7 3 5





# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 2 1 4 6 7 3 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 1 2 4 6 7 3 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 1 2 4 6 7 3 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 1 2 4 3 6 7 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 1 2 3 4 6 7 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 1 2 3 4 6 7 5



# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.

0 1 2 3 4 6 5 7





# 1A All Sorts of Sorts

Show the steps taken by insertion sort on this list.



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 4 2 7 6 1 3 5

Smallest element, after linear pass over all elements in the array: -----



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 4 2 7 6 1 3 5

Do we need to swap 0?



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 4 2 7 6 1 3 5

Next smallest element, after linear pass over all  
remaining elements in the array: \_\_\_\_\_



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 4 2 7 6 1 3 5

Do we need to swap?



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 7 6 4 3 5

Next smallest element, after linear pass over all  
remaining elements in the array: \_\_\_\_\_



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 7 6 4 3 5

Do we need to swap?



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 7 6 4 3 5

Next smallest element, after linear pass over all  
remaining elements in the array: \_\_\_\_\_





## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 7 6 4 3 5

Do we need to swap?



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 3 6 4 7 5

Next smallest element, after linear pass over all  
remaining elements in the array: \_\_\_\_\_



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 3 6 4 7 5

Do we need to swap?



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 3 4 6 7 5

Next smallest element, after linear pass over all  
remaining elements in the array: \_\_\_\_\_



# 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 3 4 6 7 5

Do we need to swap?



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 3 4 5 7 6

Next smallest element, after linear pass over all  
remaining elements in the array: \_\_\_\_\_



## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 3 4 5 7 6

Do we need to swap?



# 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 3 4 5 6 7

Next smallest element, after linear pass over all  
remaining elements in the array: -----





## 1B All Sorts of Sorts

Show the steps taken by selection sort on this list.

0 1 2 3 4 5 6 7

Do we need to swap?



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 4 2 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 4 2 7 ]

[ 6 1 3 5 ]

[ ]

[ 2 7 ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 4 2 7 ]

[ 6 1 3 5 ]

[ 0 ]

[ 2 7 ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 4 2 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 4 2 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 4 2 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]





# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 4 2 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ [ 6 1 3 5 ]

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

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



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 6 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 1 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]





# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ 6 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ 1 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ 1 3 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

0 4 2 7 6 1 3 5

[ 0 2 4 7 ]

[ 1 3 5 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]





# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ 0 4 2 7 6 1 3 5 ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ 0 ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ 0 1 ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ 0 1 2 ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ 0 1 2 3 ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ 0 1 2 3 4 ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ 0 1 2 3 4 5 ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]





# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ 0 1 2 3 4 5 6 ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ 0 1 2 3 4 5 6 7 ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1C All Sorts of Sorts

Show the steps taken by mergesort on this list.

[ 0 1 2 3 4 5 6 7 ]

[ 0 2 4 7 ]

[ 1 3 5 6 ]

[ 0 4 ]

[ 2 7 ]

[ 1 6 ]

[ 3 5 ]

[ 0 ]

[ 4 ]

[ 2 ]

[ 7 ]

[ 6 ]

[ 1 ]

[ 3 ]

[ 5 ]



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

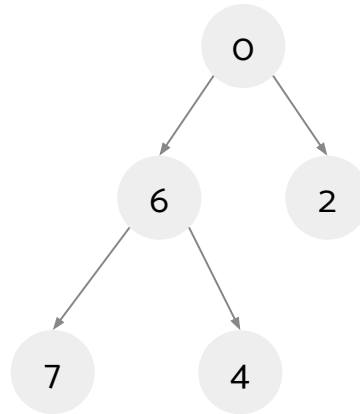
0 6 2 7 4



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

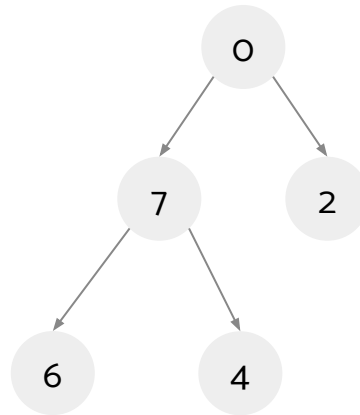
0 6 2 7 4



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

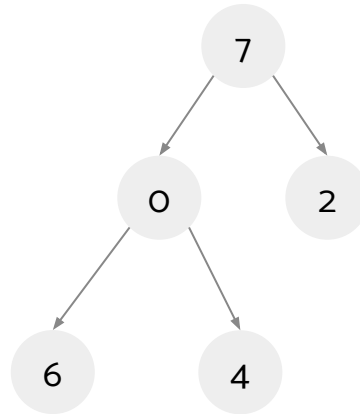
0 7 2 6 4



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

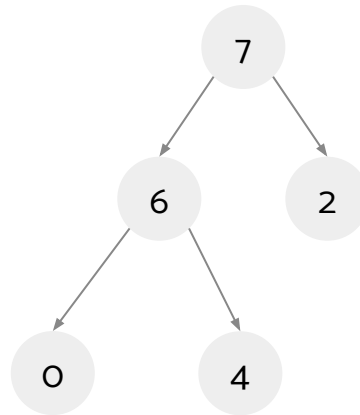
7 0 2 6 4



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

7 6 2 0 4

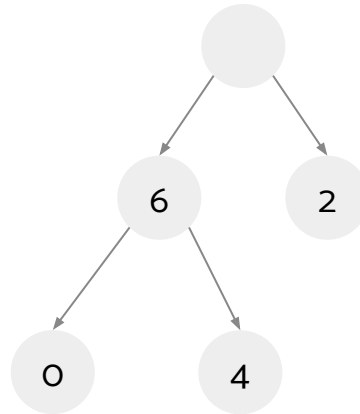




# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

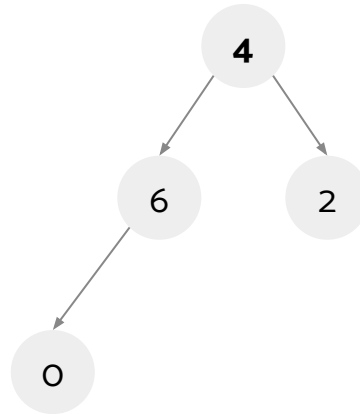
6 2 0 4 7



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

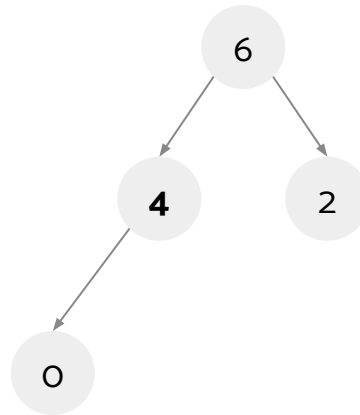
4 6 2 0 7



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

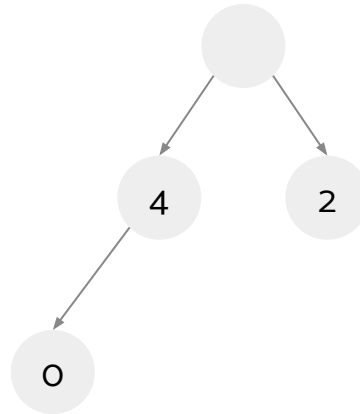
6 4 2 0 7



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

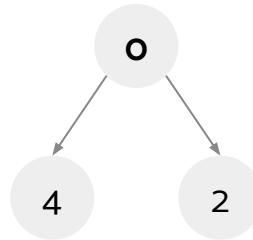
4 2 0 6 7



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

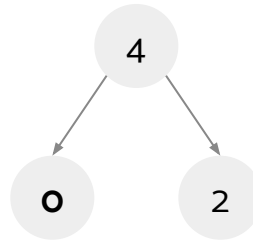
0 4 2 6 7



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

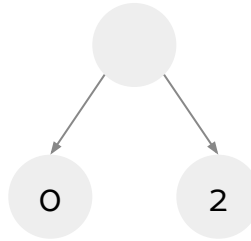
4 0 2 6 7



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

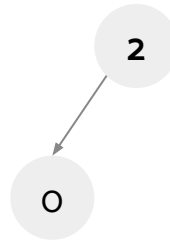
0 2 4 6 7



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

2 0 4 6 7

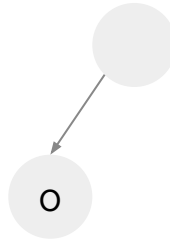




# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

0 2 4 6 7



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

0 2 4 6 7

0



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

0 2 4 6 7



# 1D All Sorts of Sorts

Show the steps taken by heapsort on this list.

0 2 4 6 7



## 2A Crystal Has Been Waiting For This

Implement a comparator to help sort TAs by height.

```
public class TA {  
    private String name;  
    private int height;  
  
    public TA(String name, int height) {  
        this.name = name;  
        this.height = height;  
    }  
}
```

```
public class TAComparator implements  
    Comparator<____> {  
    @Override  
    public int compare(____ o1, ____ o2) {  
  
    }  
}
```



## 2A Crystal Has Been Waiting For This Implement a comparator to help sort TAs by height.

```
public class TA {  
    private String name;  
    private int height;  
  
    public TA(String name, int height) {  
        this.name = name;  
        this.height = height;  
    }  
}
```

```
public class TAComparator implements  
    Comparator<TA> {  
    @Override  
    public int compare(TA o1, TA o2) {  
        if (o1.height < o2.height) {  
            return -1;  
        } else if (o1.height > o2.height) {  
            return 1;  
        }  
        return 0;  
    }  
}
```



## 2B Crystal Has Been Waiting For This

Given the following list of TAs, who would make the worst pivot for Quicksort? What about the best pivot?

```
TA eddie = new TA("Eddie", 6);  
TA ronnie = new TA("Ronnie", 9001);  
TA aditya = new TA("Aditya", 1);  
TA elana = new TA("Elana", 5);  
TA sree = new TA("Sree", 7);  
TA noah = new TA("Noah", 25);  
TA dhruti = new TA("Dhruti", 9);  
TA william = new TA("William", 4);  
TA jasmine = new TA("Jasmine", 8);  
TA austin = new TA("Austin", 8);
```



## 2B Crystal Has Been Waiting For This

Given the following list of TAs, who would make the worst pivot for Quicksort? What about the best pivot?

```
TA eddie = new TA("Eddie", 6);  
TA ronnie = new TA("Ronnie", 9001);  
TA aditya = new TA("Aditya", 1);  
TA elana = new TA("Elana", 5);  
TA sree = new TA("Sree", 7);  
TA noah = new TA("Noah", 25);  
TA dhruti = new TA("Dhruti", 9);  
TA william = new TA("William", 4);  
TA jasmine = new TA("Jasmine", 8);  
TA austin = new TA("Austin", 8);
```

worst pivots: Ronnie (maximum height) and Aditya (minimum height).

best pivots: Sree, Jasmine, Austin (median height)





## 2C Crystal Has Been Waiting For This

```
TA jasmine = new TA("Jasmine", 8);  
TA austin = new TA("Austin", 8);
```

Austin points out that even though he got in line after Jasmine, he ended up in front of Jasmine in the sorted list produced by Quicksort (which he doesn't like because that makes it seem like he's shorter than Jasmine)! How might we ensure that Austin ends up behind Jasmine?



## 2C Crystal Has Been Waiting For This

```
TA jasmine = new TA("Jasmine", 8);  
TA austin = new TA("Austin", 8);
```

Austin points out that even though he got in line after Jasmine, he ended up in front of Jasmine in the sorted list produced by Quicksort (which he doesn't like because that makes it seem like he's shorter than Jasmine)! How might we ensure that Austin ends up behind Jasmine?

We can use a stable sort to preserve the relative ordering of Jasmine and Austin in the original list, like insertion sort or merge sort!\*

\* Technically stable Quicksort is possible, but we typically don't use the stable version



## 2D Crystal Has Been Waiting For This

Our TAs have just been sorted by height, but suddenly Elisa and Wilson come running in late! Which sort will do the most minimal work to get them in their correct spots, and what is the additional runtime it will take (ie. not including the runtime for sorting all the other TAs first)?



## 2D Crystal Has Been Waiting For This

Our TAs have just been sorted by height, but suddenly Elisa and Wilson come running in late! Which sort will do the most minimal work to get them in their correct spots, and what is the additional runtime it will take (ie. not including the runtime for sorting all the other TAs first)?

- Insertion sort: it is the most efficient on an already-sorted or nearly-sorted list
  - Why? Consider what a different sort like merge sort or Quicksort would do in this situation
- Additional Runtime:  $\Theta(N)$ 
  - At worst, we'd have to make two linear passes (ie. Elisa and Wilson are the two shortest TAs)



## 3A Zero One Two-Step

Given an array that only contains 0's, 1's and 2's, write an algorithm to sort it in linear time without creating a new array. You may want to use the provided swap method.

```
public static int[] specialSort(int[] arr) {  
    int front = 0;  
    int back = arr.length - 1;  
    int curr = 0;  
  
    while (_____) {  
        if (arr[curr] < 1) {  
            _____;  
            _____;  
            _____;  
        } else if (arr[curr] > 1) {  
            _____;  
            _____;  
            _____;  
        } else {  
            _____;  
        }  
    }  
}
```

```
private static void swap(int[] arr, int  
i, int j) {  
    int temp = arr[i];  
    arr[i] = arr[j];  
    arr[j] = temp;  
}
```



## 3A Zero One Two-Step

Given an array that only contains 0's, 1's and 2's, write an algorithm to sort it in linear time without creating a new array. You may want to use the provided swap method.

```
public static int[] specialSort(int[] arr) {  
    int front = 0;  
    int back = arr.length - 1;  
    int curr = 0;  
  
    while (curr <= back) {  
        if (arr[curr] < 1) {  
            swap(arr, curr, front);  
            front += 1;  
            curr += 1;  
        } else if (arr[curr] > 1) {  
            swap(arr, curr, back);  
            back -= 1;  
        } else {  
            curr += 1;  
        }  
    }  
}
```

```
private static void swap(int[] arr, int  
i, int j) {  
    int temp = arr[i];  
    arr[i] = arr[j];  
    arr[j] = temp;  
}
```



## 3B Zero One Two-Step

We just wrote a linear time sort, how cool! Can you explain in a sentence or two why we can't always use this sort, even though it has better runtime than Mergesort or Quicksort?



## 3B Zero One Two-Step

We just wrote a linear time sort, how cool! Can you explain in a sentence or two why we can't always use this sort, even though it has better runtime than Mergesort or Quicksort?

Our array only contained 3 types of values, which is a guarantee we can't make in the real world





## 3C Zero One Two-Step

The sort we wrote above is also "in place". What does it mean to sort "in place", and why would we want this?



# 3C Zero One Two-Step

The sort we wrote above is also "in place". What does it mean to sort "in place", and why would we want this?

- “in place”: does not require significant extra space
  - “Significant extra space”:  $\geq$  linear, with respect to the number of elements we are sorting
  - In place sorting algorithms: selection sort, insertion sort, heap sort
    - Mergesort technically can be implemented in place, but it’s rather complex.
- Doing operations in place is beneficial because we typically want to use as little memory/computer resources as possible. Though in this class, we focus on time efficiency, space efficiency is important too in the real world!

