Comparison Sorts

Topical Review Session 06

CS 61B // Spring 2021

Agenda

Topic

- ❏ Selection Sort
- ❏ Insertion Sort
- ❏ Heap Sort
- ❏ Merge Sort
- ❏ Quick Sort

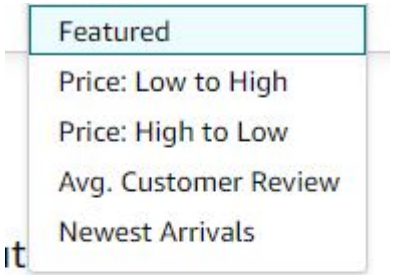
Intro To Sorting

Sorts

What is sorting? Putting items in order!

Real World Examples:

Amazon



Utensils Drawer (iStock)

Stable Sort

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. (Source)

- Unsorted is like dirty utensils (F_c, F_a, S_a, K_b, F_b)
- Sorting algorithm is like the dishwashing method ($F < S < K$)
- Stable Sort: F_c, F_a, F_b, S_a, K_b
- Unstable Sort: F_c, F_b, F_a, S_a, K_b

Notice how the subscript ordering of F is not maintained

Selection Sort

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. Find the minimum element in the unsorted section
2. Swap it with the first element in the unsorted section
3. Repeat until the entire array is sorted

2_A	7	3	2_B	1_A	1_B	6

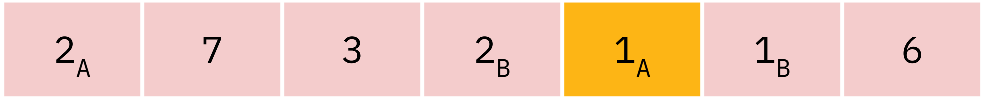
Red: Unsorted
Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes O(N^2) time.

Steps:

1. **Find the minimum element in the unsorted section**
2. Swap it with the first element in the unsorted section
3. Repeat until the entire array is sorted



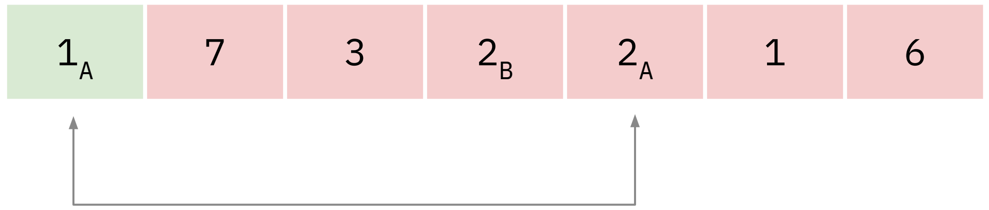
🟥 Red: Unsorted
🟩 Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. Find the minimum element in the unsorted section
2. **Swap it with the first element in the unsorted section**
3. Repeat until the entire array is sorted



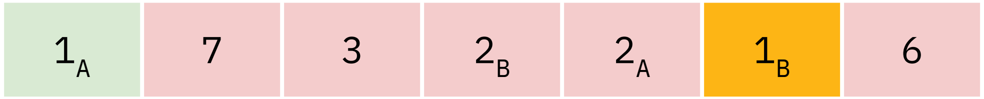
🟥 Red: Unsorted
🟩 Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. **Find the minimum element in the unsorted section**
2. Swap it with the first element in the unsorted section
3. Repeat until the entire array is sorted



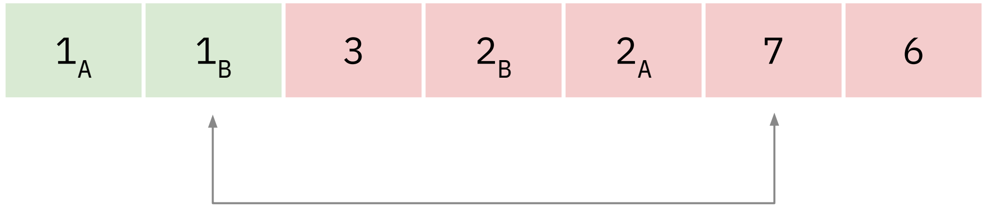
Red: Unsorted
Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. Find the minimum element in the unsorted section
2. **Swap it with the first element in the unsorted section**
3. Repeat until the entire array is sorted



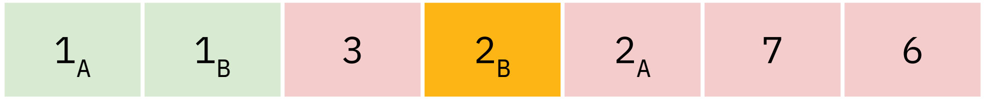
🟥 Red: Unsorted
🟩 Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. **Find the minimum element in the unsorted section**
2. Swap it with the first element in the unsorted section
3. Repeat until the entire array is sorted



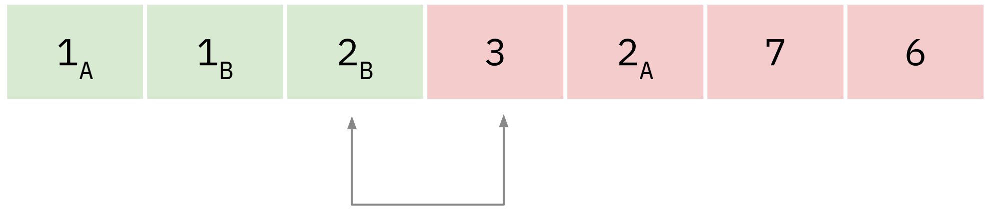
🟥 Red: Unsorted
🟩 Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. Find the minimum element in the unsorted section
2. **Swap it with the first element in the unsorted section**
3. Repeat until the entire array is sorted



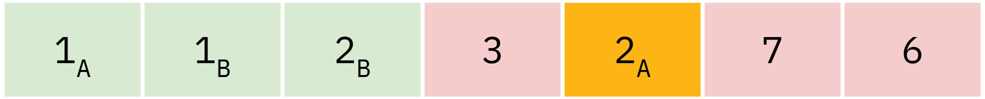
🟥 Red: Unsorted
🟩 Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. **Find the minimum element in the unsorted section**
2. Swap it with the first element in the unsorted section
3. Repeat until the entire array is sorted



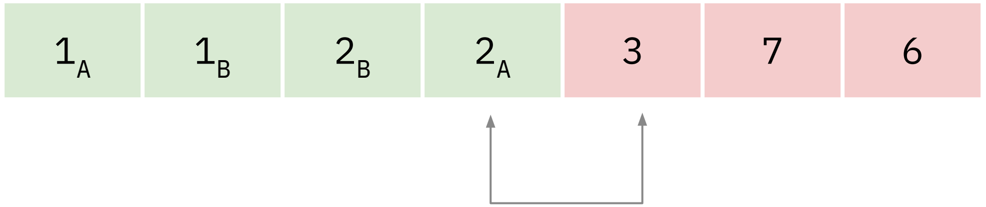
Red: Unsorted
Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. Find the minimum element in the unsorted section
2. **Swap it with the first element in the unsorted section**
3. Repeat until the entire array is sorted



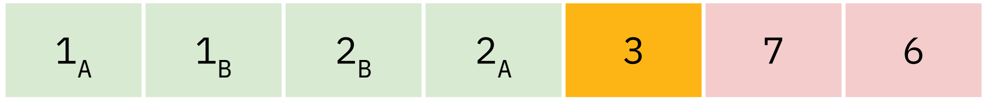
🟥 Red: Unsorted
🟩 Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. **Find the minimum element in the unsorted section**
2. Swap it with the first element in the unsorted section
3. Repeat until the entire array is sorted



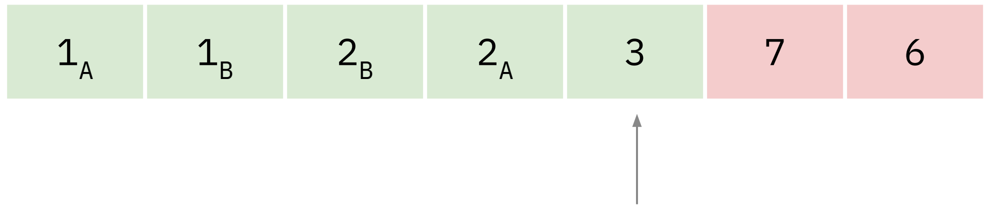
Red: Unsorted
Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. Find the minimum element in the unsorted section
2. **Swap it with the first element in the unsorted section**
3. Repeat until the entire array is sorted



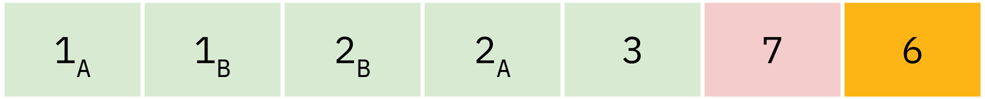
🟥 Red: Unsorted
🟩 Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. **Find the minimum element in the unsorted section**
2. Swap it with the first element in the unsorted section
3. Repeat until the entire array is sorted



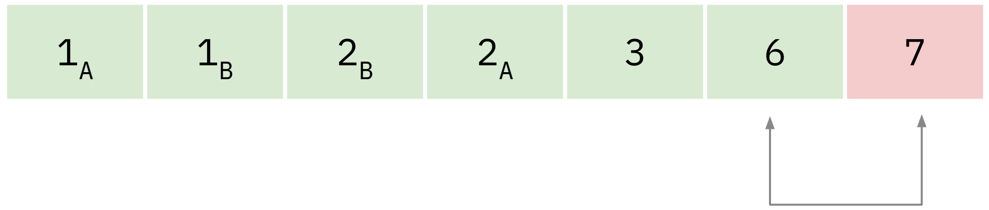
Red: Unsorted
Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. Find the minimum element in the unsorted section
2. **Swap it with the first element in the unsorted section**
3. Repeat until the entire array is sorted



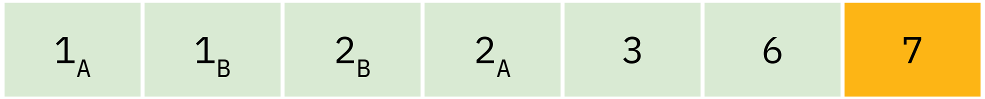
Red: Unsorted
Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes O(N^2) time.

Steps:

1. **Find the minimum element in the unsorted section**
2. Swap it with the first element in the unsorted section
3. Repeat until the entire array is sorted



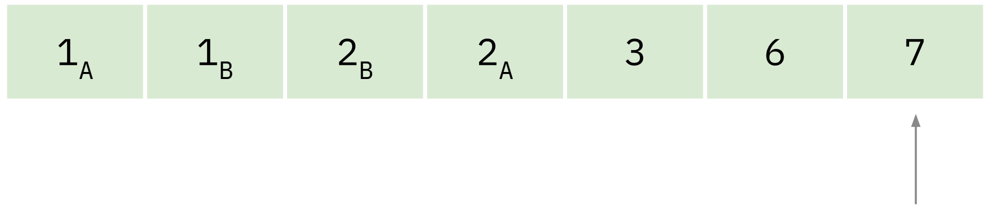
Red: Unsorted
Green: Sorted

Selection Sort

Selection sort is an unstable sort that takes $O(N^2)$ time.

Steps:

1. Find the minimum element in the unsorted section
2. **Swap it with the first element in the unsorted section**
3. Repeat until the entire array is sorted



Red: Unsorted
Green: Sorted

Selection Sort

Best Case Time Complexity - $\Theta(N^2)$

Worst Case Time Complexity - $\Theta(N^2)$

Why are they the same?
Selection sort will go through the same process even if everything is sorted already. (1, 2, 3, 4, 5)

Space Complexity - $\Theta(1)$ (in place!)

Stable - No

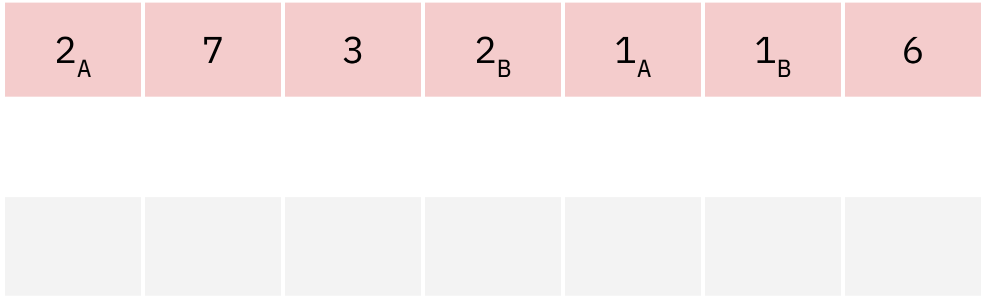
Insertion Sort

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. Take the first element in the array that has not been sorted yet
2. Add it to the correct spot in the empty array
3. Repeat until the entire array is sorted



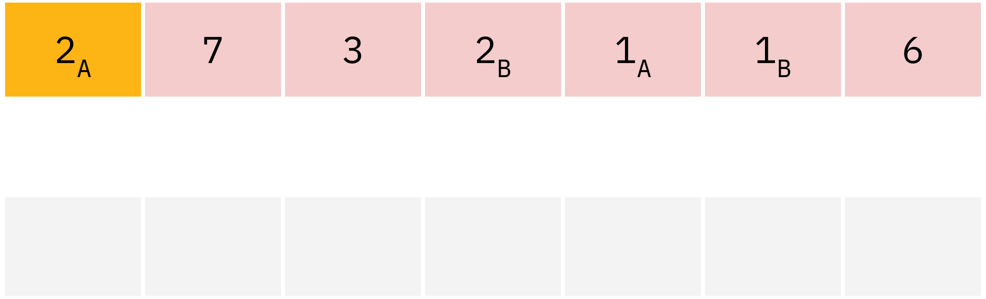
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. **Take the first element in the array that has not been sorted yet**
2. Add it to the correct spot in the empty array
3. Repeat until the entire array is sorted



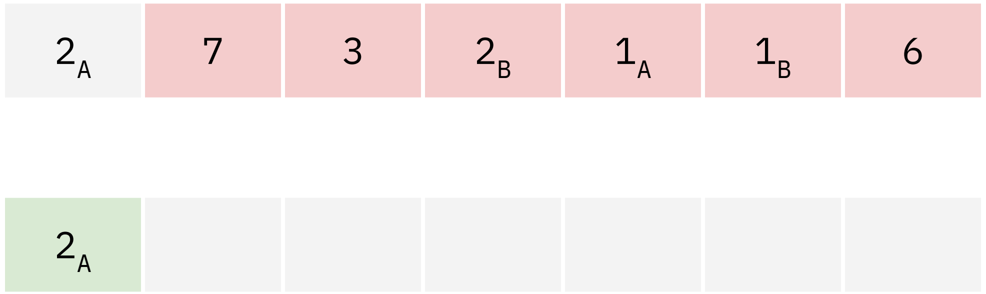
🟥 Red: Unsorted
🟩 Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. Take the first element in the array that has not been sorted yet
2. **Add it to the correct spot in the empty array**
3. Repeat until the entire array is sorted



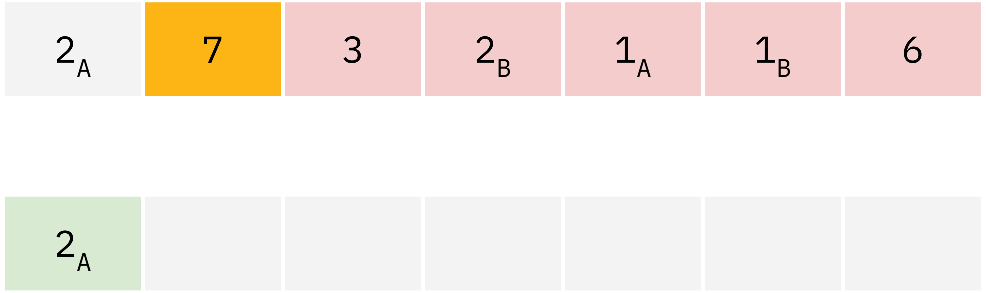
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. **Take the first element in the array that has not been sorted yet**
2. Add it to the correct spot in the empty array
3. Repeat until the entire array is sorted



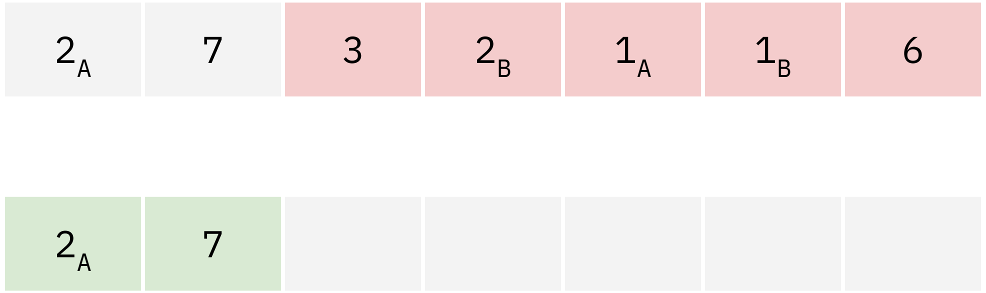
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. Take the first element in the array that has not been sorted yet
2. **Add it to the correct spot in the empty array**
3. Repeat until the entire array is sorted



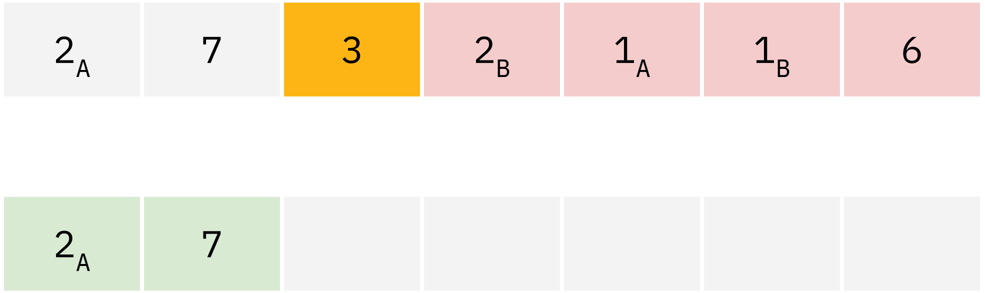
🟥 Red: Unsorted
🟩 Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. **Take the first element in the array that has not been sorted yet**
2. Add it to the correct spot in the empty array
3. Repeat until the entire array is sorted



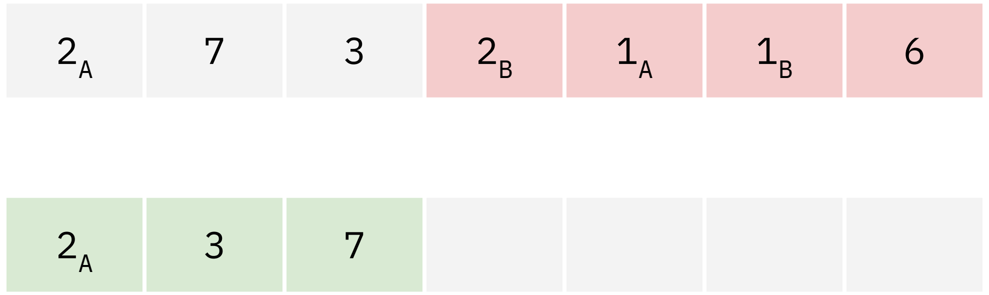
🟥 Red: Unsorted
🟩 Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. Take the first element in the array that has not been sorted yet
2. **Add it to the correct spot in the empty array**
3. Repeat until the entire array is sorted



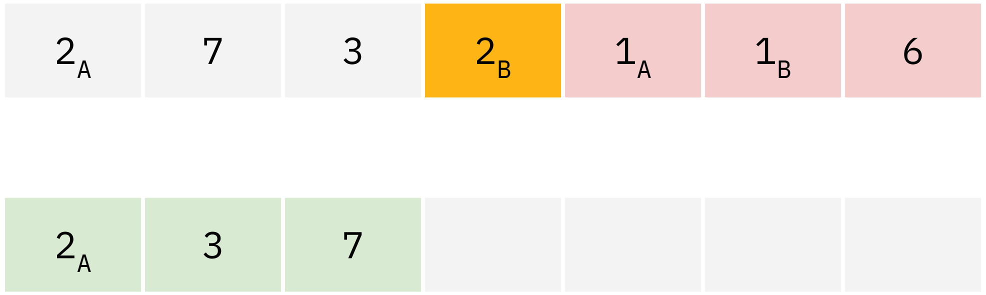
🟥 Red: Unsorted
🟩 Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. **Take the first element in the array that has not been sorted yet**
2. Add it to the correct spot in the empty array
3. Repeat until the entire array is sorted



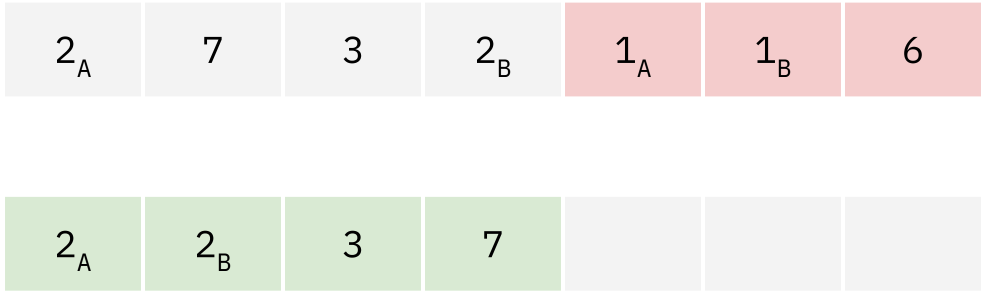
🟥 Red: Unsorted
🟩 Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. Take the first element in the array that has not been sorted yet
2. **Add it to the correct spot in the empty array**
3. Repeat until the entire array is sorted



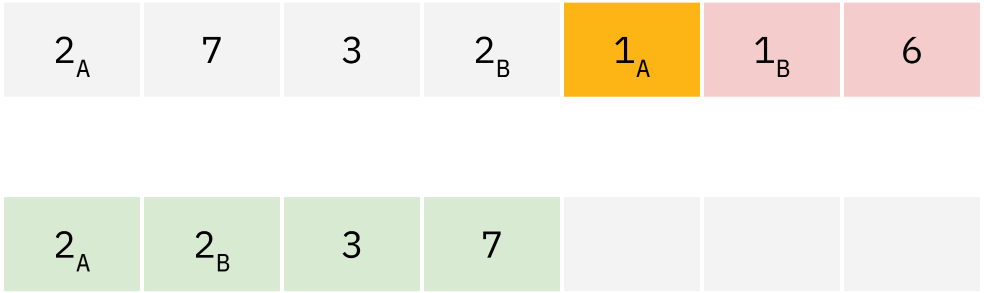
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. **Take the first element in the array that has not been sorted yet**
2. Add it to the correct spot in the empty array
3. Repeat until the entire array is sorted



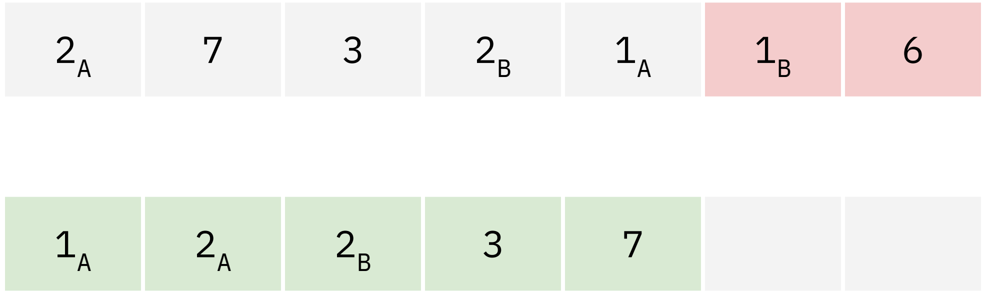
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. Take the first element in the array that has not been sorted yet
2. **Add it to the correct spot in the empty array**
3. Repeat until the entire array is sorted



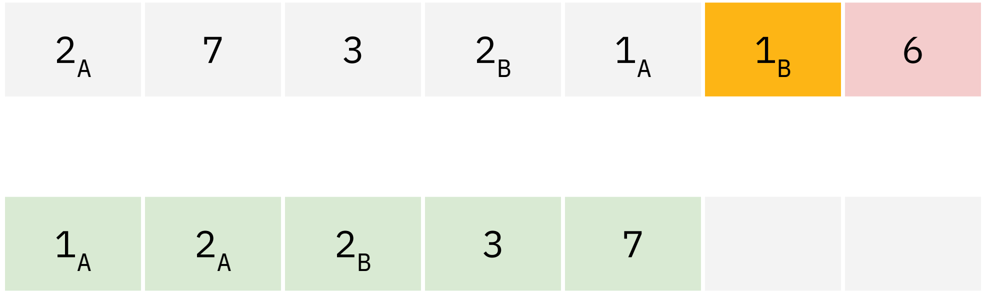
🟥 Red: Unsorted
🟩 Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. **Take the first element in the array that has not been sorted yet**
2. Add it to the correct spot in the empty array
3. Repeat until the entire array is sorted



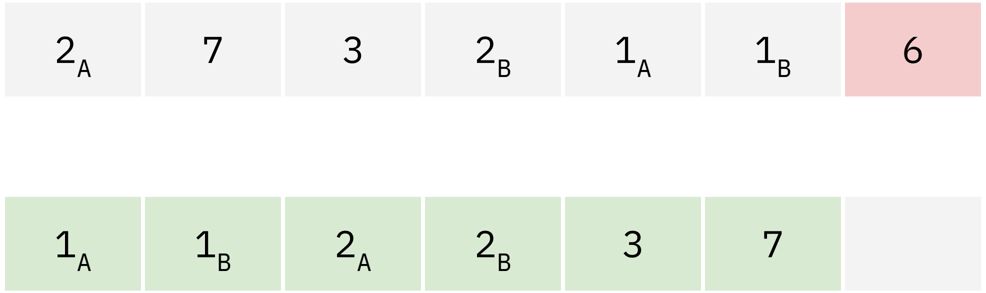
🟥 Red: Unsorted
🟩 Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. Take the first element in the array that has not been sorted yet
2. **Add it to the correct spot in the empty array**
3. Repeat until the entire array is sorted



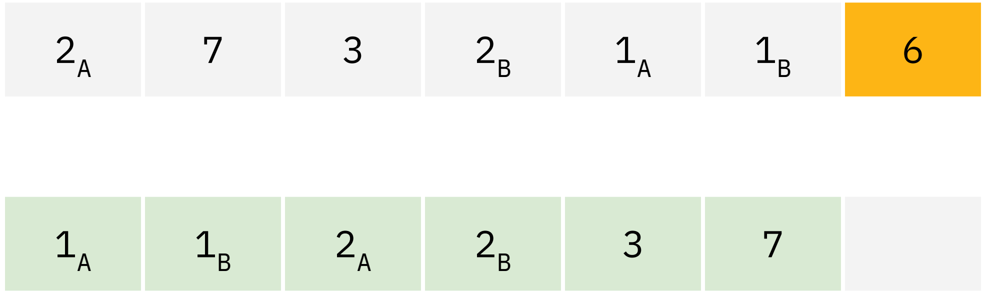
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. **Take the first element in the array that has not been sorted yet**
2. Add it to the correct spot in the empty array
3. Repeat until the entire array is sorted



Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort is a stable sort that takes $O(N^2)$ time.

Steps:

1. **Take the first element in the array that has not been sorted yet**
2. Add it to the correct spot in the empty array
3. Repeat until the entire array is sorted

2_A	7	3	2_B	1_A	1_B	6

1_A	1_B	2_A	2_B	3	6	7

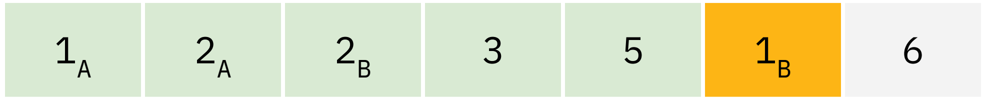
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort can also be done in-place, which allows it to take up less space

Steps:

1. Take the first element in the array that has not been sorted yet
2. Swap it with the item on its left if that item is greater than it
3. Repeat swapping until the item on the left is smaller than the current item
4. Repeat until the entire array is sorted



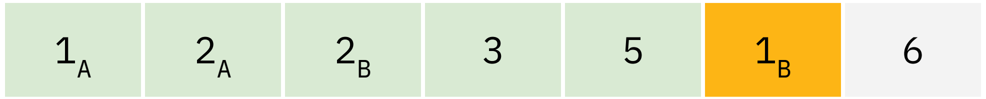
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort can also be done in-place, which allows it to take up less space

Steps:

1. **Take the first element in the array that has not been sorted yet**
2. Swap it with the item on its left if that item is greater than it
3. Repeat swapping until the item on the left is smaller than the current item
4. Repeat until the entire array is sorted



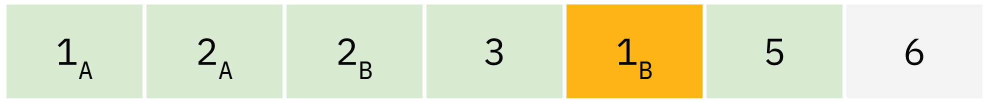
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort can also be done in-place, which allows it to take up less space

Steps:

1. Take the first element in the array that has not been sorted yet
2. **Swap it with the item on its left if that item is greater than it**
3. Repeat swapping until the item on the left is smaller than the current item
4. Repeat until the entire array is sorted



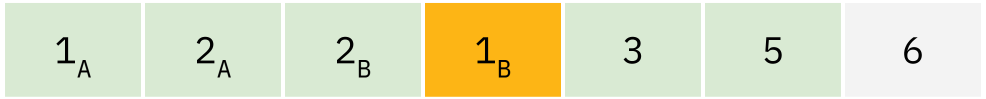
🟥 Red: Unsorted
🟩 Green: Sorted

Insertion Sort

Insertion sort can also be done in-place, which allows it to take up less space

Steps:

1. Take the first element in the array that has not been sorted yet
2. **Swap it with the item on its left if that item is greater than it**
3. Repeat swapping until the item on the left is smaller than the current item
4. Repeat until the entire array is sorted



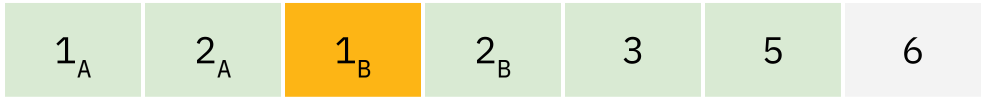
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort can also be done in-place, which allows it to take up less space

Steps:

1. Take the first element in the array that has not been sorted yet
2. **Swap it with the item on its left if that item is greater than it**
3. Repeat swapping until the item on the left is smaller than the current item
4. Repeat until the entire array is sorted



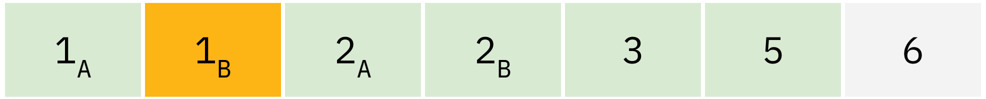
Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort can also be done in-place, which allows it to take up less space

Steps:

1. Take the first element in the array that has not been sorted yet
2. **Swap it with the item on its left if that item is greater than it**
3. Repeat swapping until the item on the left is smaller than the current item
4. Repeat until the entire array is sorted



Red: Unsorted
Green: Sorted

Insertion Sort

Insertion sort can also be done in-place, which allows it to take up less space

Steps:

1. Take the first element in the array that has not been sorted yet
2. Swap it with the item on its left if that item is greater than it
3. Repeat swapping until the item on the left is smaller than the current item
4. Repeat until the entire array is sorted

1_A	1_B	2_A	2_B	3	5	6

Red: Unsorted
Green: Sorted

Insertion Sort

Assume swapping is O(1)

Best Case Time Complexity - $\Theta(n)$

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

Worst Case Time Complexity - $\Theta(n+n^2) \rightarrow \Theta(n^2)$

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

Space Complexity - O(1), if done in place with pointers

Stable? Yes

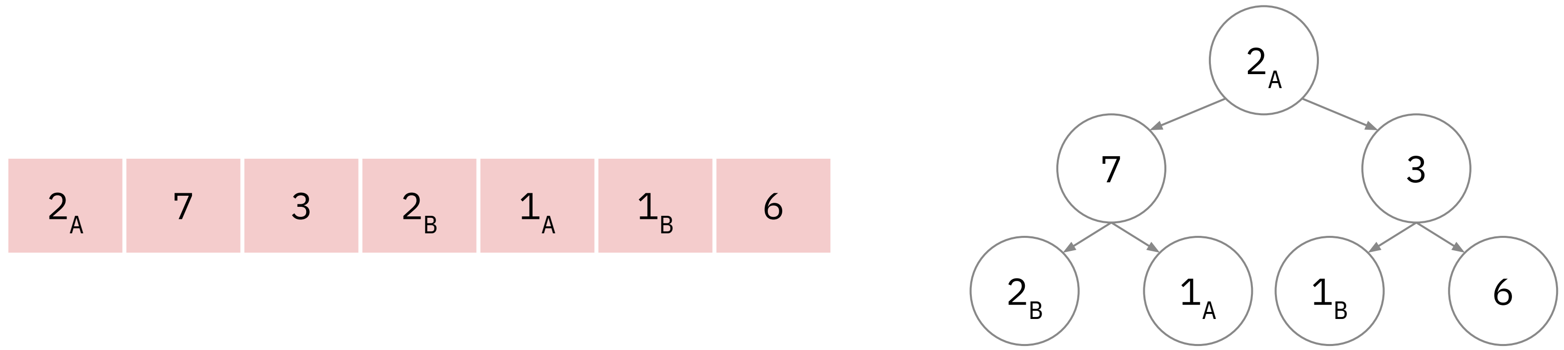
Heap Sort

Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. Heapify the elements of the unsorted portion of the array into a max heap
2. Pop the largest element off the array by swapping it with the last element in the unsorted portion
3. Repeat until the entire array is sorted



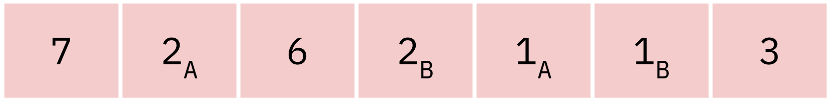
Red: Unsorted
Green: Sorted

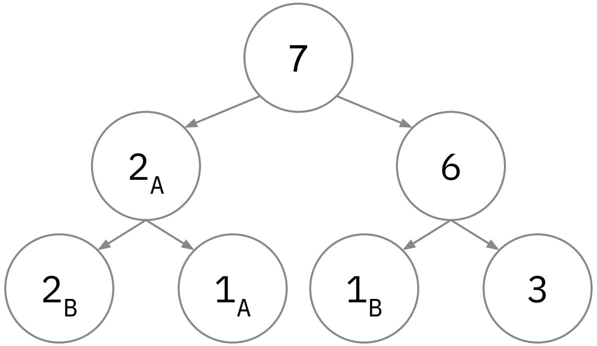
Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. **Heapify the elements of the unsorted portion of the array into a max heap**
2. Pop the largest element off the array by swapping it with the last element in the unsorted portion
3. Repeat until the entire array is sorted





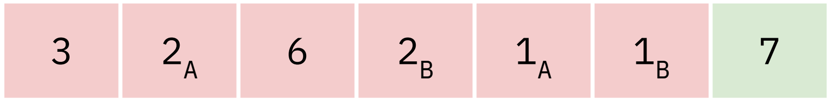
Red: Unsorted
Green: Sorted

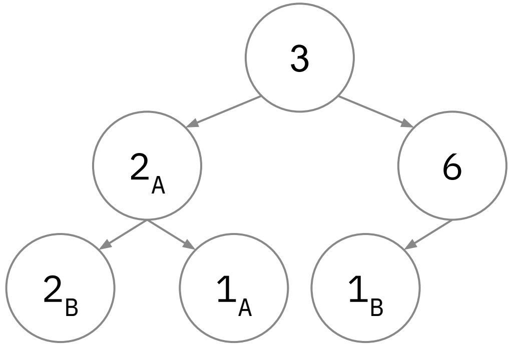
Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. Heapify the elements of the unsorted portion of the array into a max heap
2. **Pop the largest element off the array by swapping it with the last element in the unsorted portion**
3. Repeat until the entire array is sorted





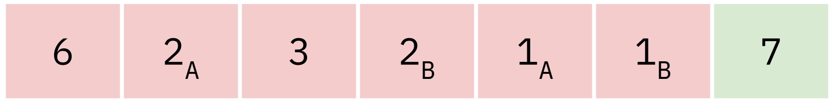
Red: Unsorted
Green: Sorted

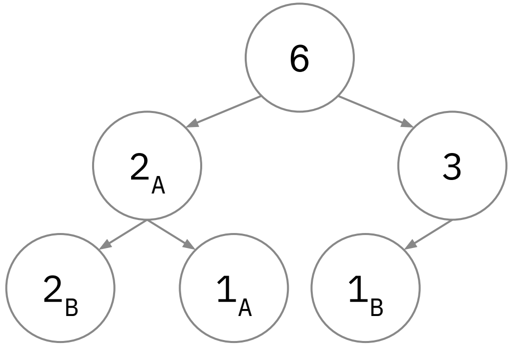
Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. **Heapify the elements of the unsorted portion of the array into a max heap**
2. Pop the largest element off the array by swapping it with the last element in the unsorted portion
3. Repeat until the entire array is sorted





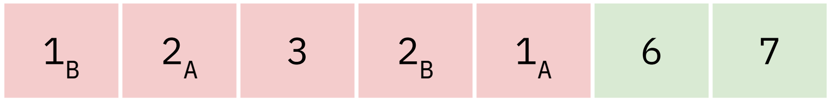
🟥 Red: Unsorted
🟩 Green: Sorted

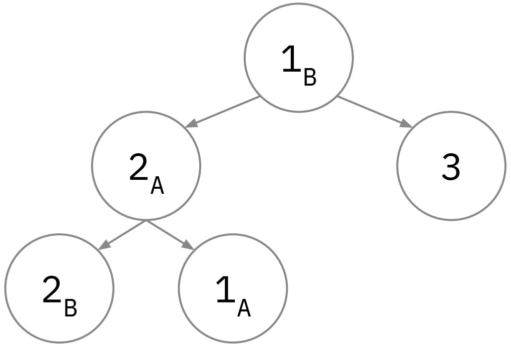
Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. Heapify the elements of the unsorted portion of the array into a max heap
2. **Pop the largest element off the array by swapping it with the last element in the unsorted portion**
3. Repeat until the entire array is sorted





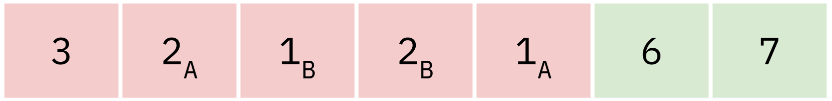
Red: Unsorted
Green: Sorted

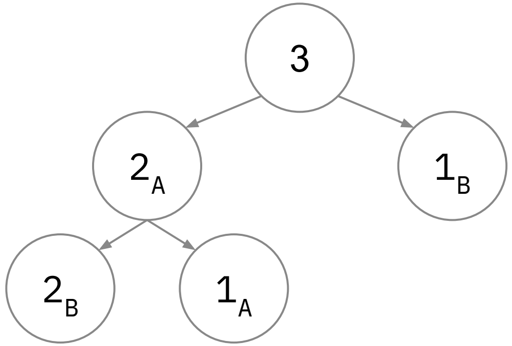
Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. **Heapify the elements of the unsorted portion of the array into a max heap**
2. Pop the largest element off the array by swapping it with the last element in the unsorted portion
3. Repeat until the entire array is sorted





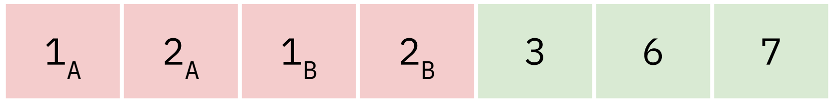
Red: Unsorted
Green: Sorted

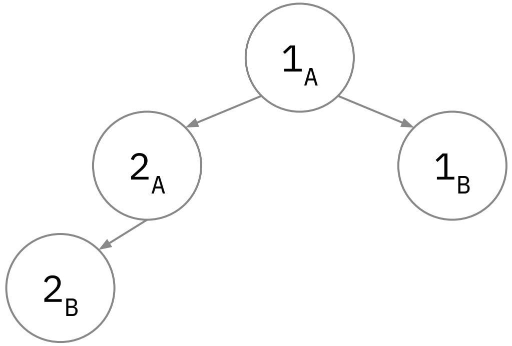
Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. Heapify the elements of the unsorted portion of the array into a max heap
2. **Pop the largest element off the array by swapping it with the last element in the unsorted portion**
3. Repeat until the entire array is sorted





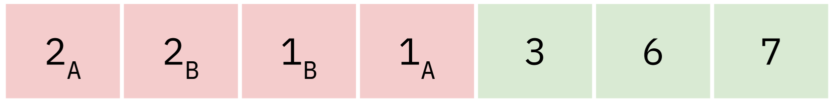
🟥 Red: Unsorted
🟩 Green: Sorted

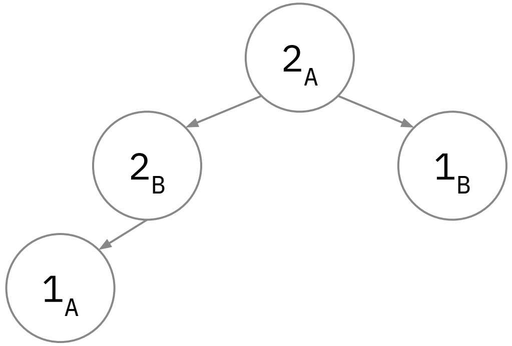
Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. **Heapify the elements of the unsorted portion of the array into a max heap**
2. Pop the largest element off the array by swapping it with the last element in the unsorted portion
3. Repeat until the entire array is sorted





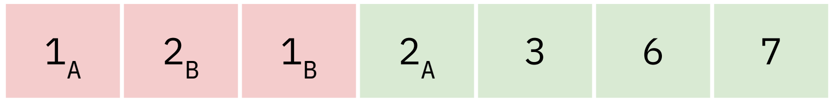
🟥 Red: Unsorted
🟩 Green: Sorted

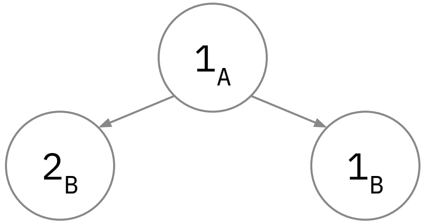
Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. Heapify the elements of the unsorted portion of the array into a max heap
2. **Pop the largest element off the array by swapping it with the last element in the unsorted portion**
3. Repeat until the entire array is sorted





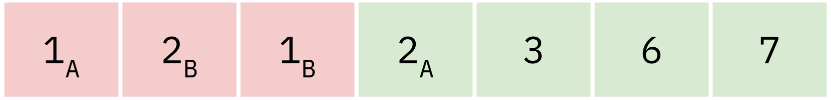
Red: Unsorted
Green: Sorted

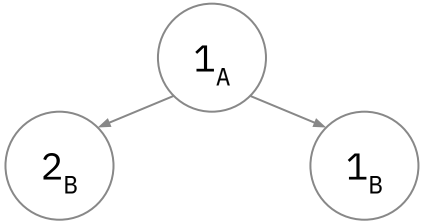
Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. Heapify the elements of the unsorted portion of the array into a max heap
2. **Pop the largest element off the array by swapping it with the last element in the unsorted portion**
3. Repeat until the entire array is sorted





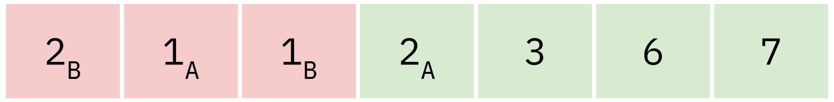
Red: Unsorted
Green: Sorted

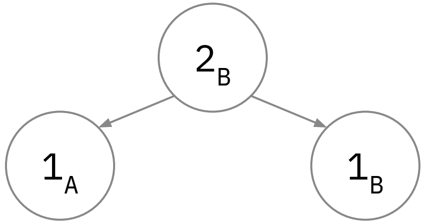
Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. **Heapify the elements of the unsorted portion of the array into a max heap**
2. Pop the largest element off the array by swapping it with the last element in the unsorted portion
3. Repeat until the entire array is sorted





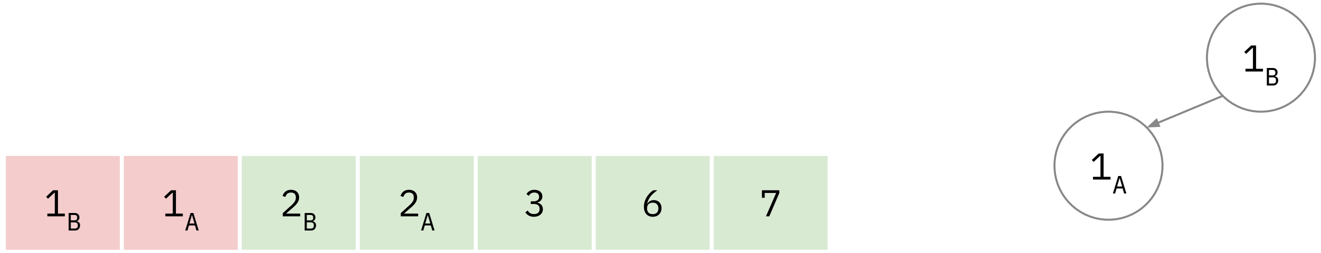
🟥 Red: Unsorted
🟩 Green: Sorted

Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. Heapify the elements of the unsorted portion of the array into a max heap
2. **Pop the largest element off the array by swapping it with the last element in the unsorted portion**
3. Repeat until the entire array is sorted



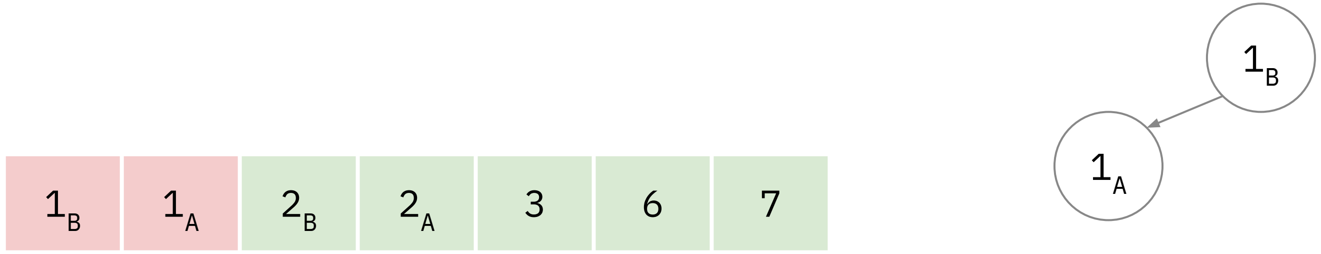
Red: Unsorted
Green: Sorted

Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. **Heapify the elements of the unsorted portion of the array into a max heap**
2. Pop the largest element off the array by swapping it with the last element in the unsorted portion
3. Repeat until the entire array is sorted



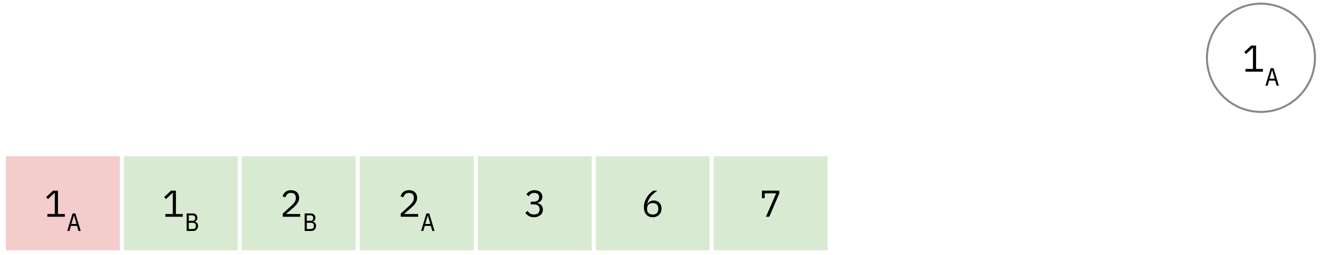
🟥 Red: Unsorted
🟩 Green: Sorted

Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. Heapify the elements of the unsorted portion of the array into a max heap
2. **Pop the largest element off the array by swapping it with the last element in the unsorted portion**
3. Repeat until the entire array is sorted



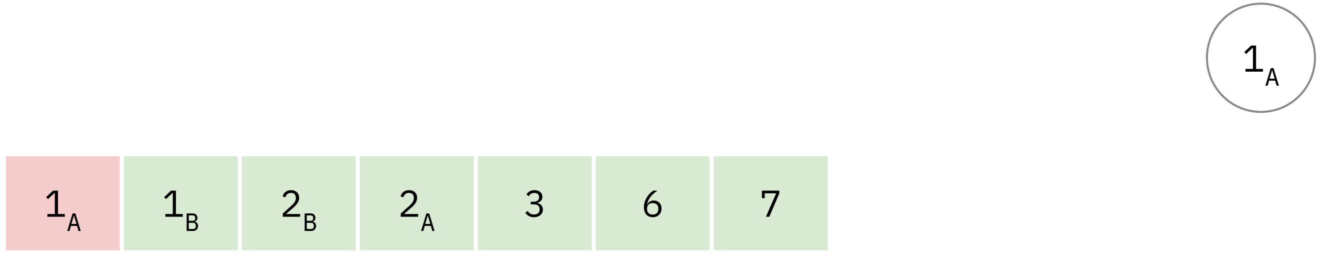
Red: Unsorted
Green: Sorted

Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. **Heapify the elements of the unsorted portion of the array into a max heap**
2. Pop the largest element off the array by swapping it with the last element in the unsorted portion
3. Repeat until the entire array is sorted



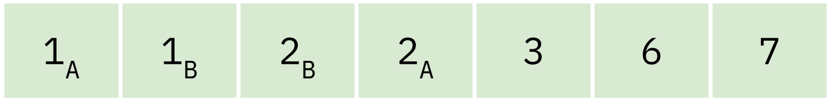
🟥 Red: Unsorted
🟩 Green: Sorted

Heap Sort

Heap sort is an unstable sort that takes O(NlogN) time.

Steps:

1. Heapify the elements of the unsorted portion of the array into a max heap
2. **Pop the largest element off the array by swapping it with the last element in the unsorted portion**
3. Repeat until the entire array is sorted



Red: Unsorted
Green: Sorted

Heap Sort

Best Case - $\Theta(n)$ $\rightarrow$ [1, 1, 1, 1]

Worst Case - $\Theta(n\log n)$

Space Complexity - $\Theta(1)$

Stable? No

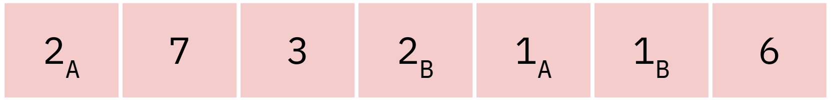
Merge Sort

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. Merge the two halves by zippering them together



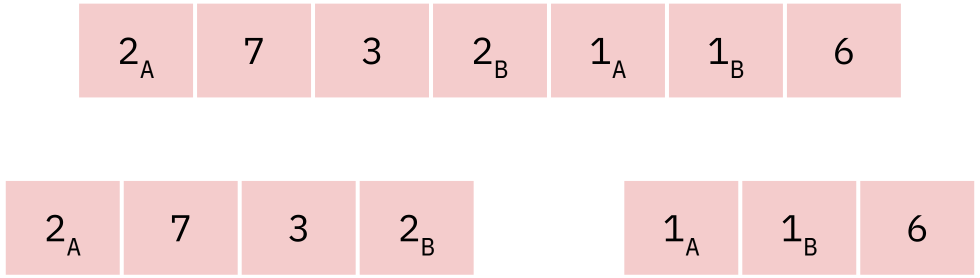
Red: Unsorted
Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. **Split the array into two even halves**
2. Perform merge sort on each half
3. Merge the two halves by zippering them together



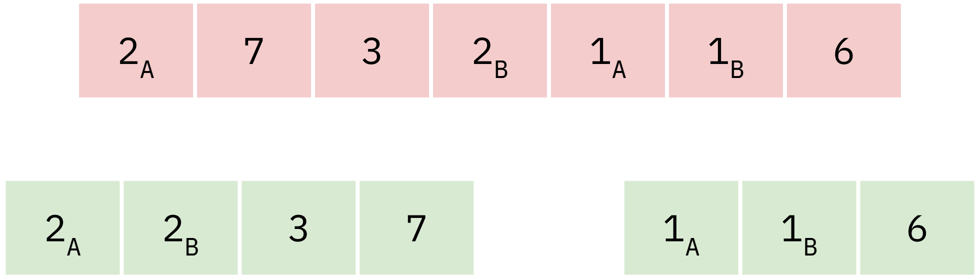
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. **Perform merge sort on each half**
3. Merge the two halves by zippering them together



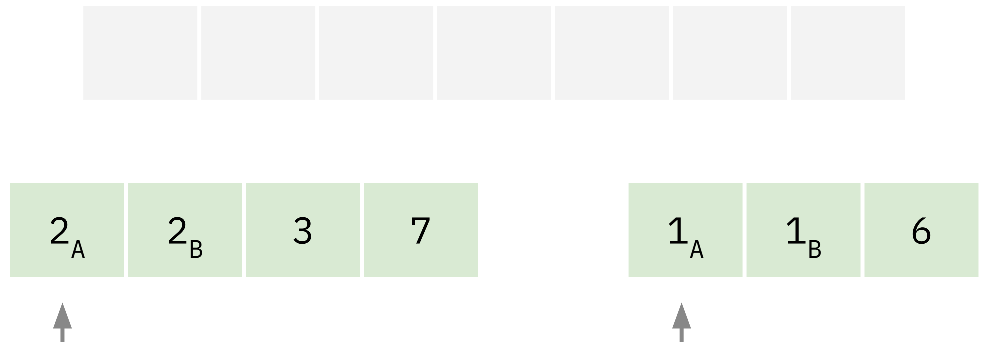
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



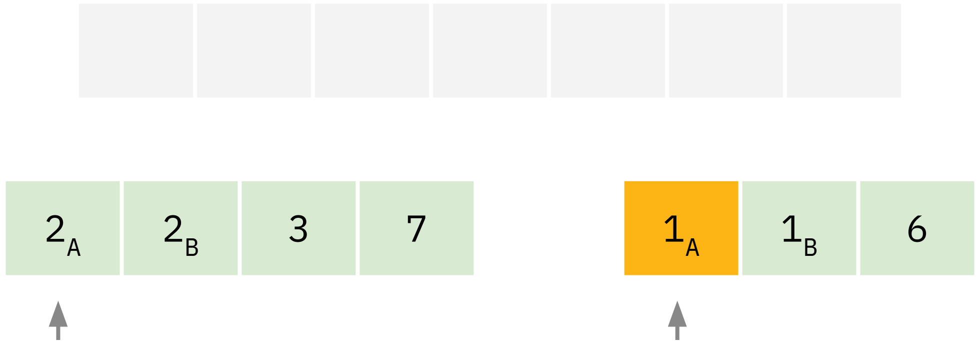
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



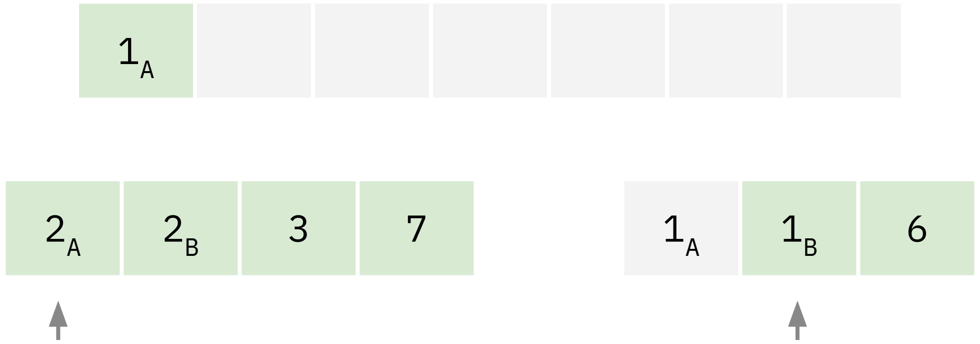
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



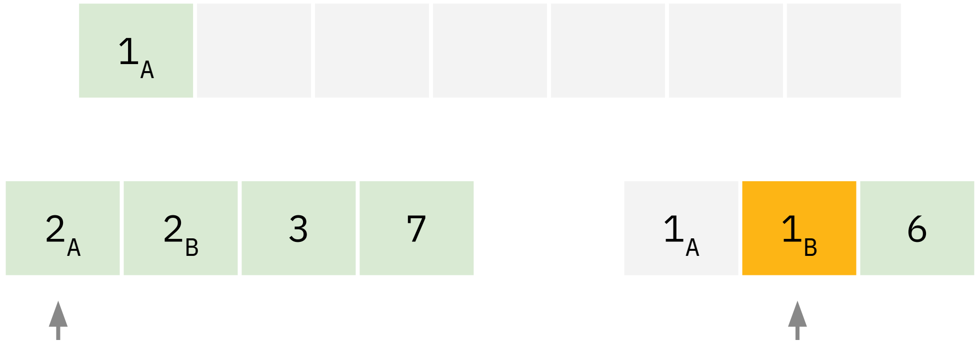
Red: Unsorted
Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



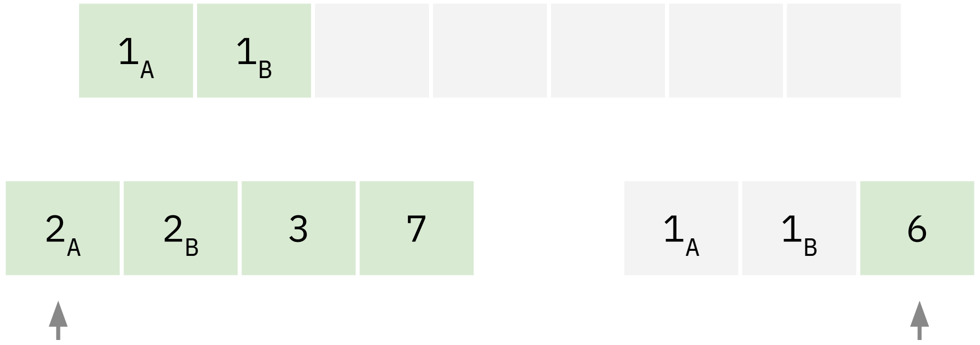
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



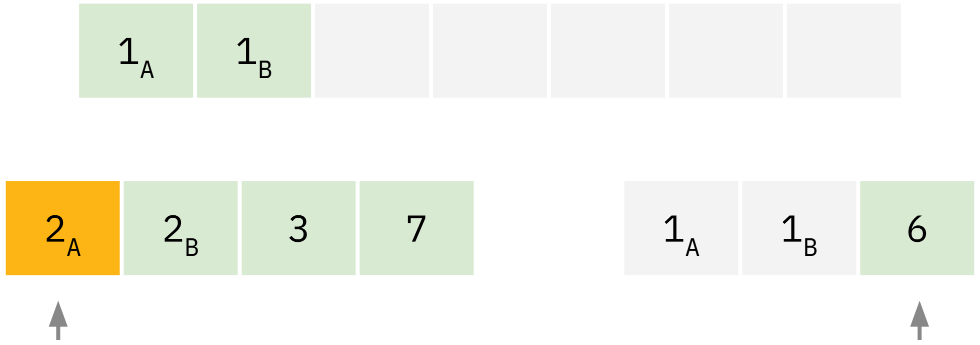
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



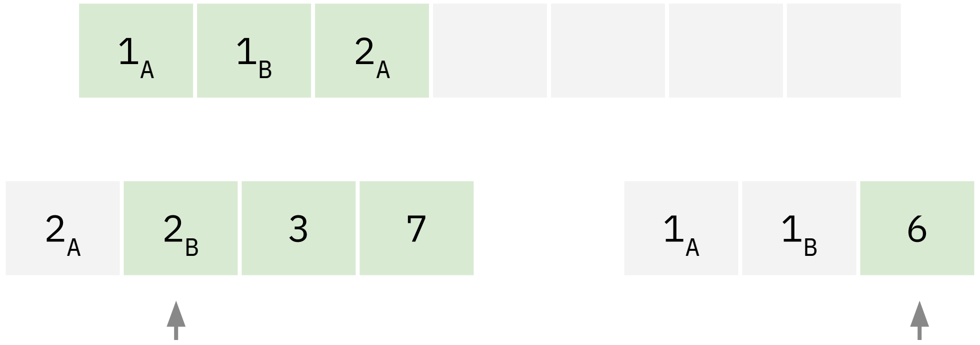
Red: Unsorted
Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



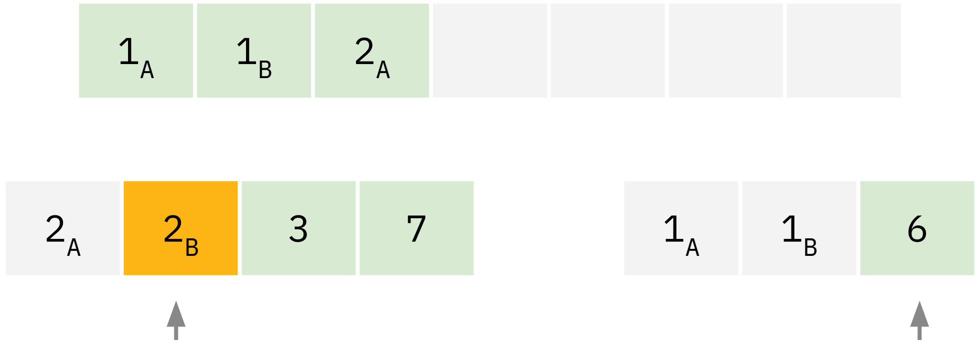
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



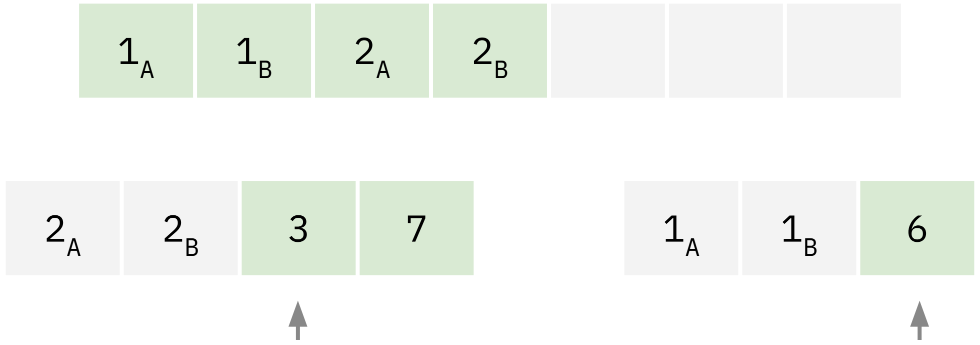
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



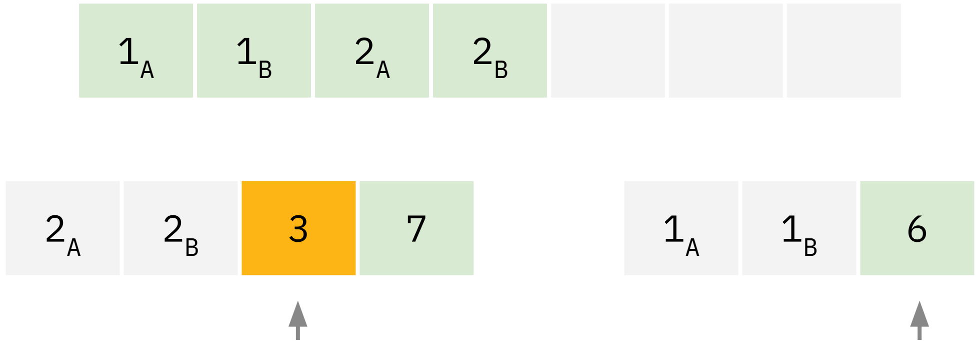
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



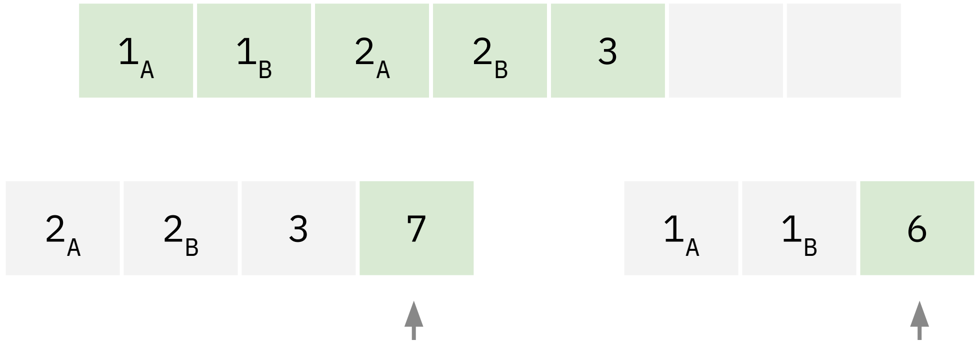
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



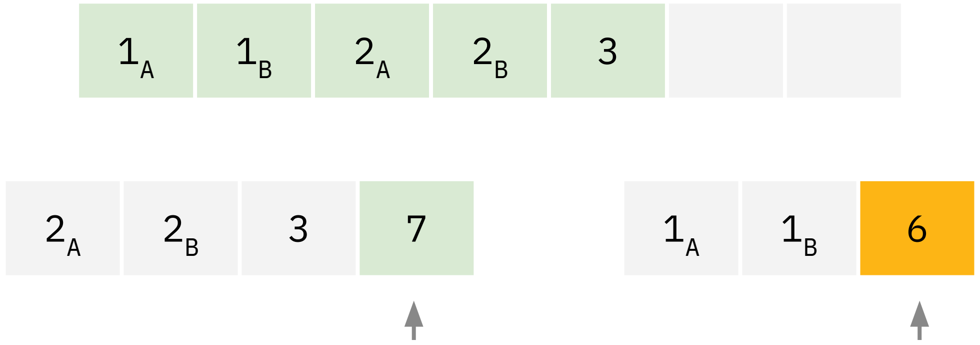
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



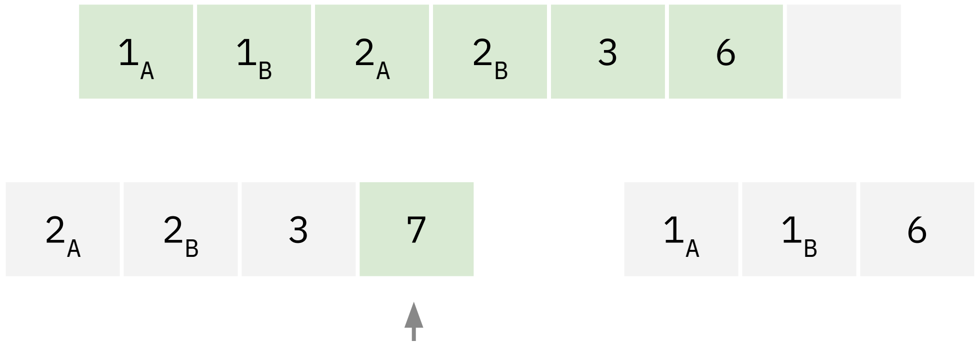
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



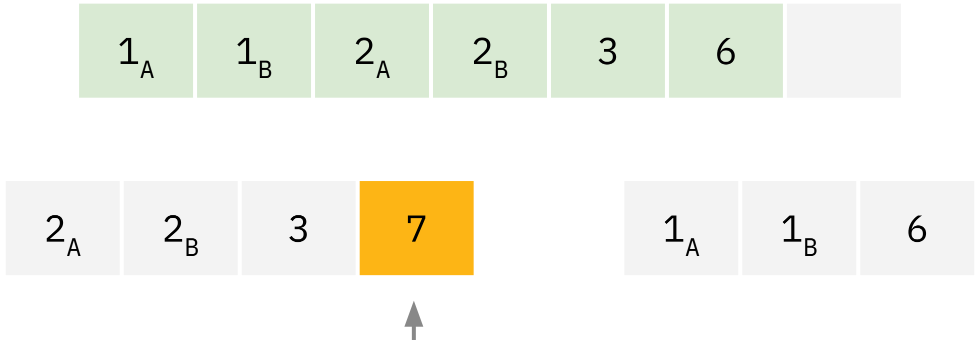
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



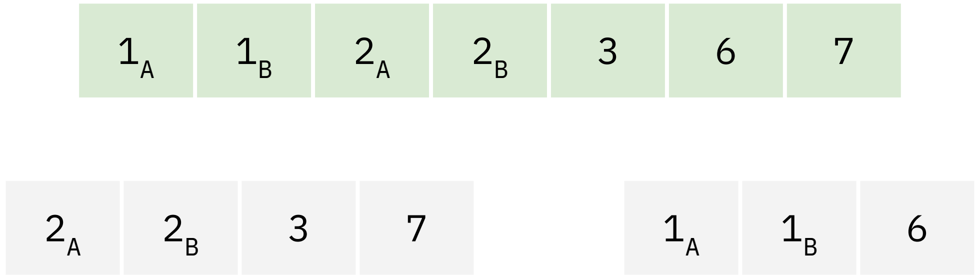
🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Merge sort is a stable sort that takes O(NlogN) time.

Steps:

1. Split the array into two even halves
2. Perform merge sort on each half
3. **Merge the two halves by zippering them together**



🟥 Red: Unsorted
🟩 Green: Sorted

Merge Sort

Best Case - $\Theta(n\log n)$

Worst Case - $\Theta(n\log n)$

Space Complexity - $\Theta(n)$

Stable? Yes

Quick Sort

Pivots

In general, for quicksort, we choose a “pivot” and then continue the rest of the algorithm:

1. We want to choose a pivot so we can “partition” the array into two parts - all elements less than the pivot and all elements greater than the pivot
2. When we rearrange these elements, this ensures that the pivot is placed in its final sorted position!
   - How we rearrange these elements depends on the partition scheme.
3. Now, repeat above steps on the two unsorted partitions until the entire list is sorted
   - This stopping condition also depends on the partition scheme

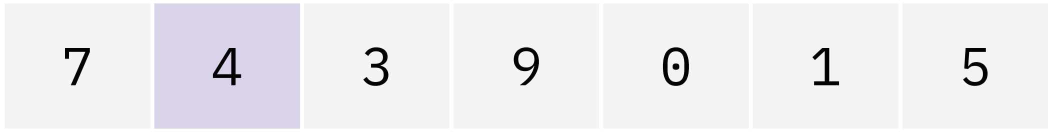
Pivots

To demonstrate, let us choose an arbitrary pivot in the following unsorted list.

7	4	3	9	0	1	5

Pivots

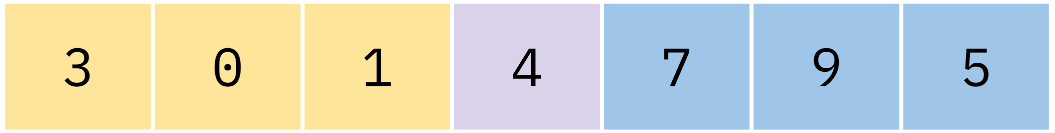
Let’s choose the element 4 as our pivot



Purple: Pivot

Pivots

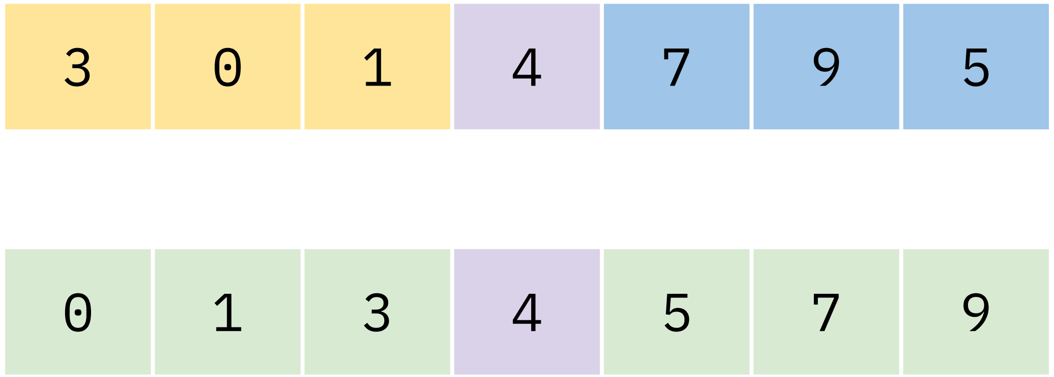
After partitioning the list, we have the following:



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Pivots

Compare the pivot's positioning with that of the sorted list. It is in the correct position!



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Pivots

- Pivot selection plays a big role in the runtime of quicksort, as we will see later
- There are many ways to choose our pivot:
  - Always choose left pivot
  - Choose median element
  - Randomly select pivot
  - Many more!

Partition Schemes

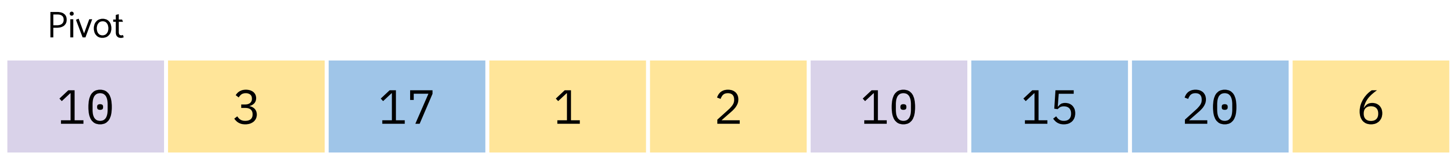
- The success of any quicksort algorithm depends on also depends on its partition scheme
- We'll discuss two relevant schemes:
  - 3 Scan
  - Hoare's

3 Scan

- 3 Scan is not in place - it requires the use of extra arrays
- We select a pivot in some way, for demonstration, we'll be choosing the leftmost pivot
- We do 3 Scans:
    1. Scan all elements less than the pivot from the original array and copy them into a new array
    2. Scan for elements equal to the pivot and copy them into a new array
    3. Scan for all elements greater than from the original array and copy them into a new array
- Now, repeat the process on each partition until the list is sorted
    - The old pivots should go into the same place in the new list before performing scans on partitions
- This means that our partitions are of size 1 or less

3 Scan

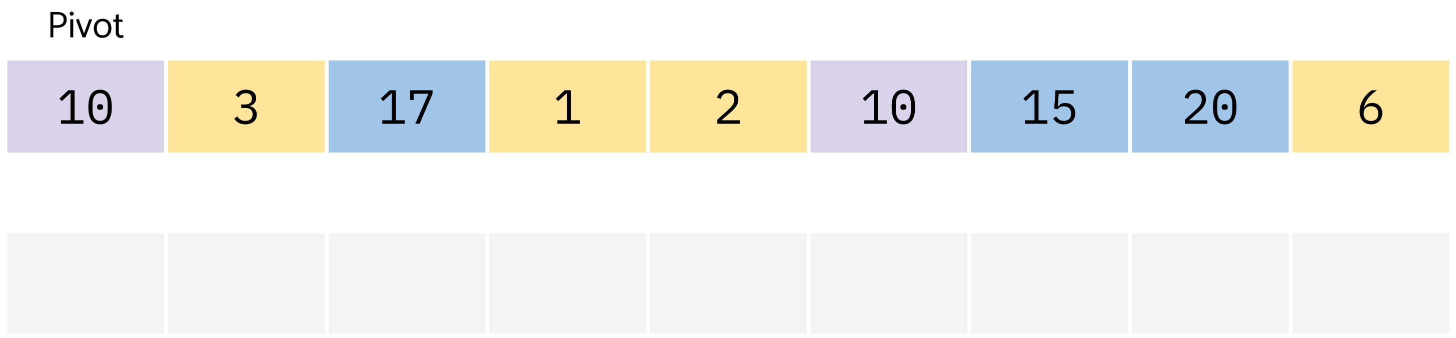
Let’s start with this array and choose a pivot:



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

3 Scan

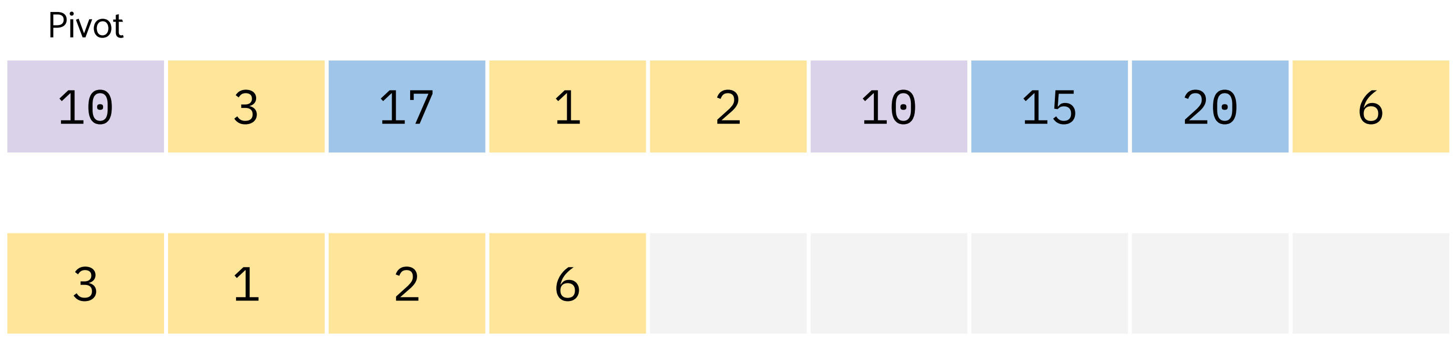
Initialize an empty array:



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

3 Scan

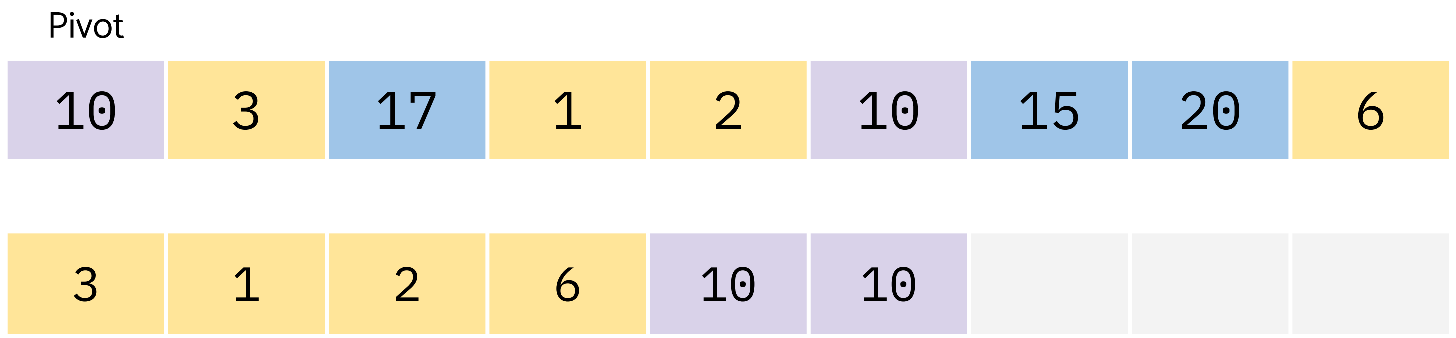
Scan and copy items that are less than the pivot:



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

3 Scan

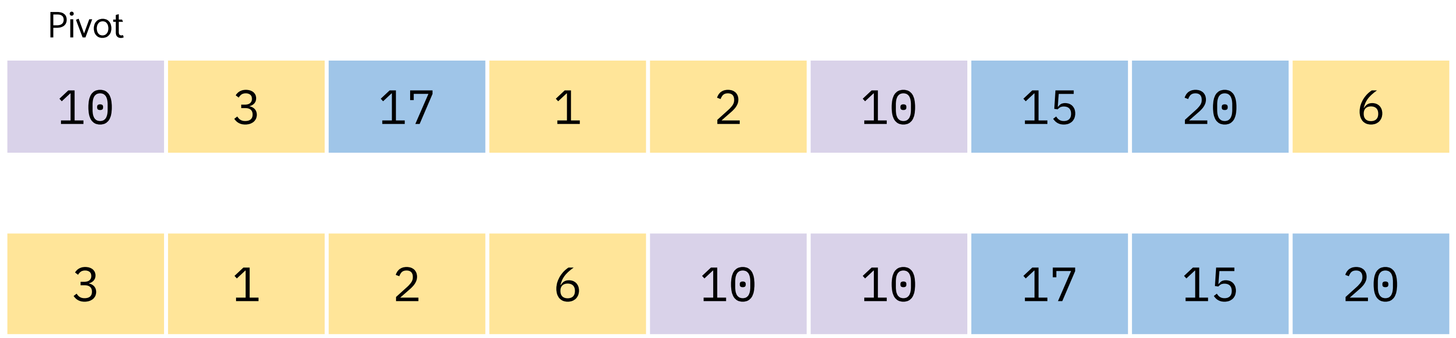
Scan and copy items equal to the pivot



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

3 Scan

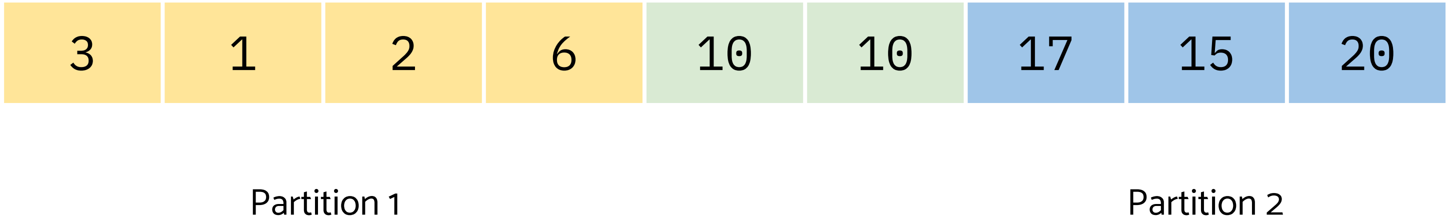
Scan and copy items greater than the pivot



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

3 Scan

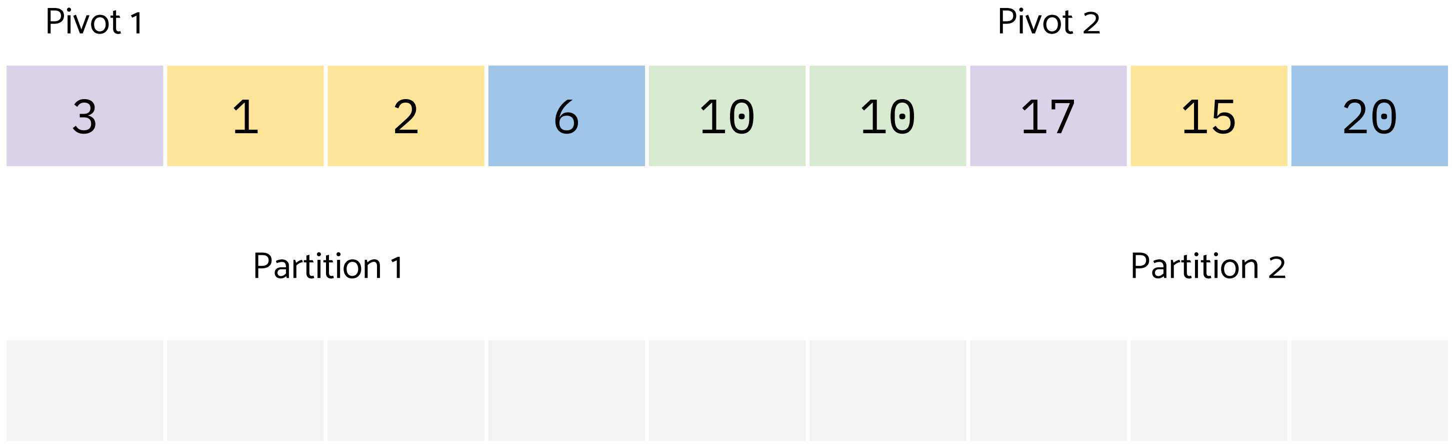
We now have a new array, partitioned into two parts:



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot
Green: Sorted

3 Scan

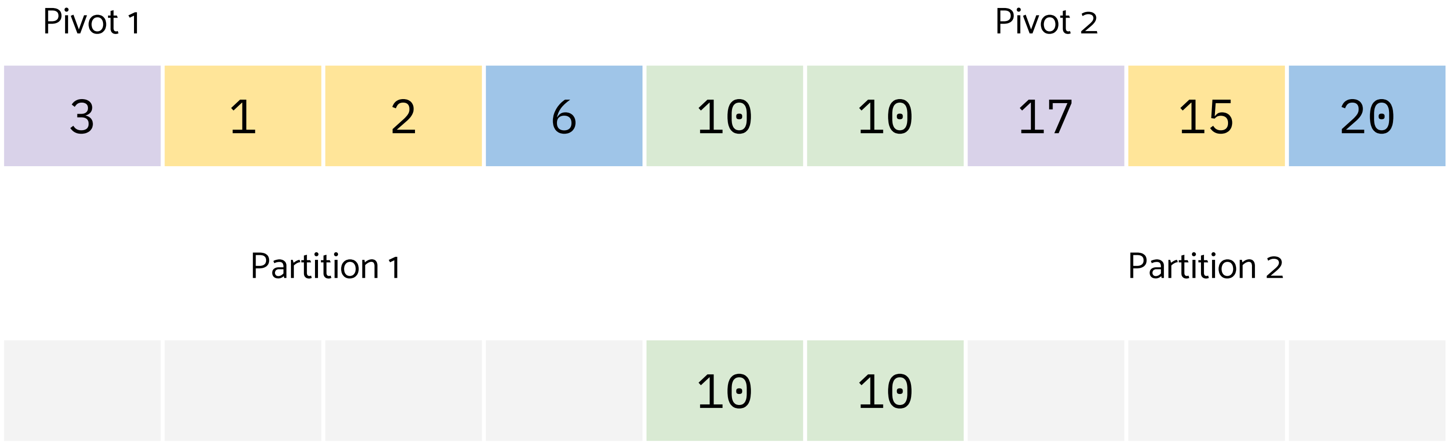
Initialize a new array



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot
Green: Sorted

3 Scan

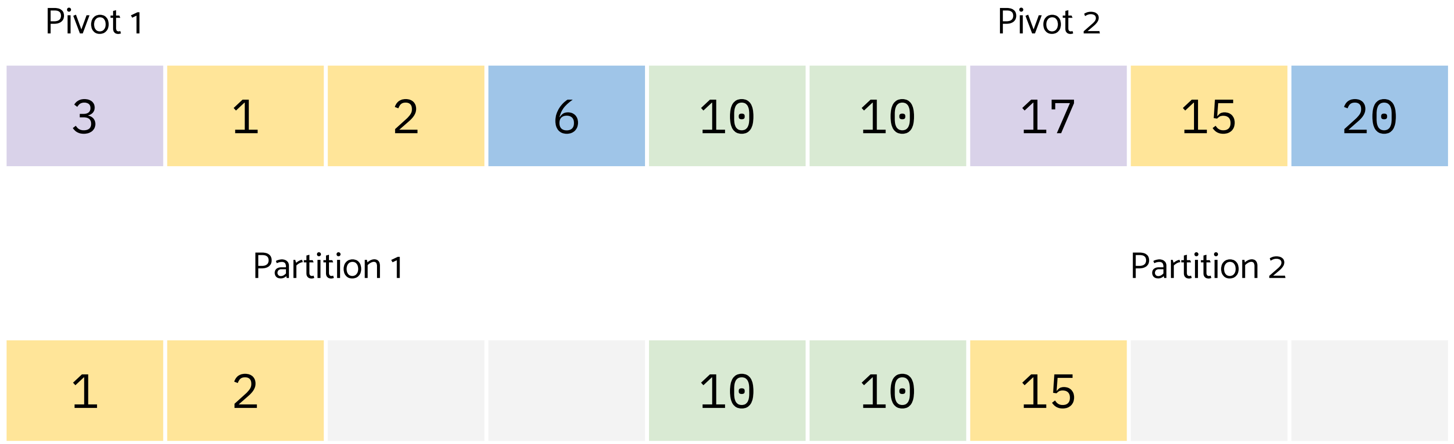
Place sorted items in place:



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot
Green: Sorted

3 Scan

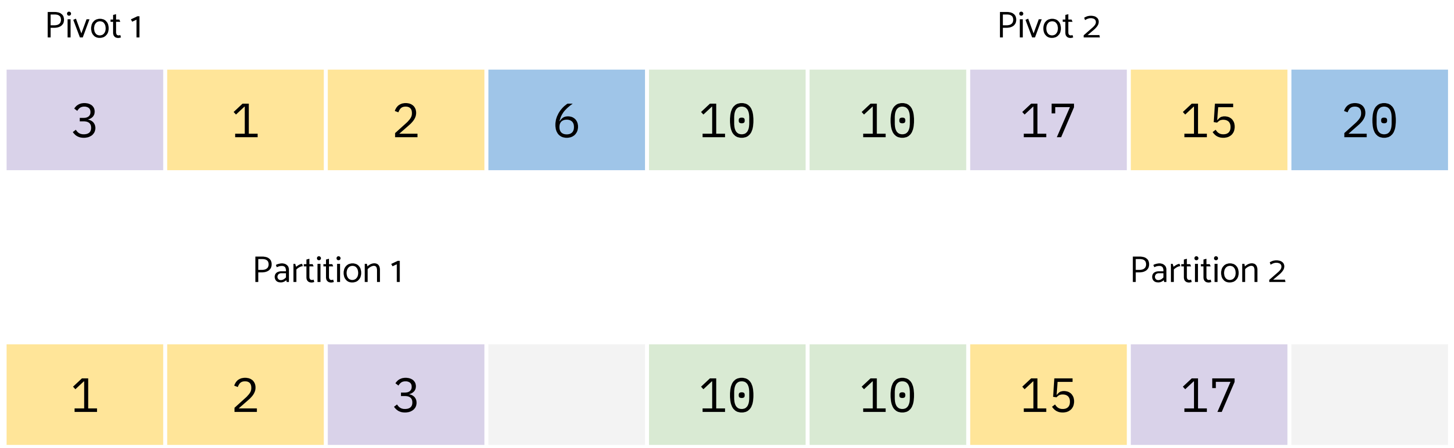
Scan for items less than the pivot:



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot
Green: Sorted

3 Scan

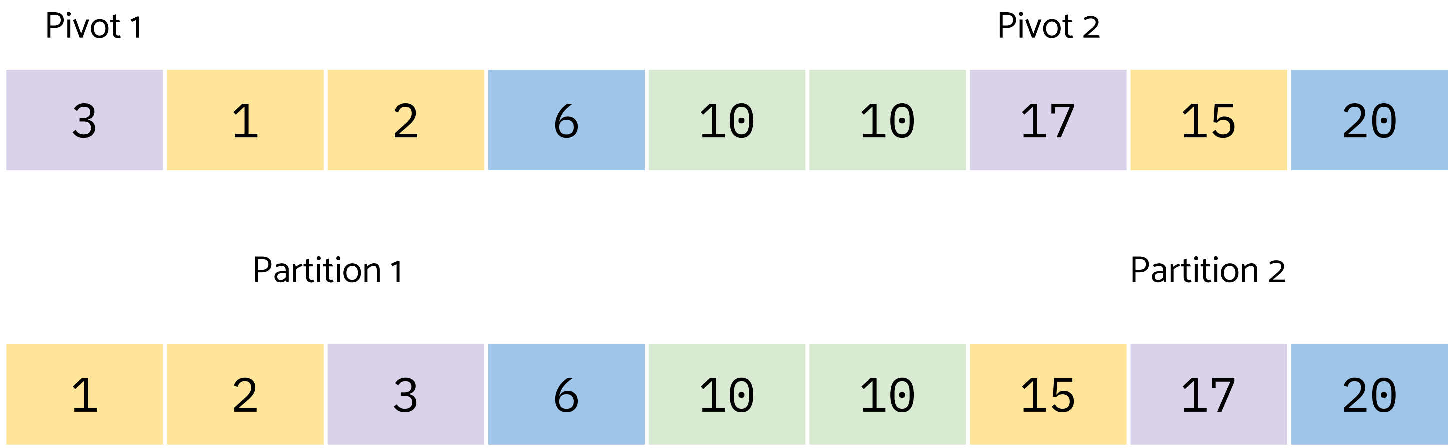
Scan for items equal to the pivot:



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot
Green: Sorted

3 Scan

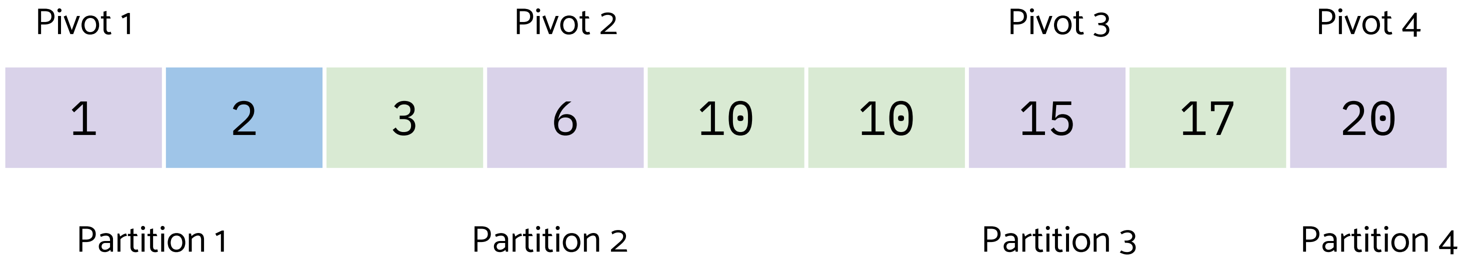
Scan for items greater than the pivot:



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot
Green: Sorted

3 Scan

We now have new partitions with their own pivots:



Notice that Partitions 2, 3 and 4 are of size 1, so we don't need to continue the process with them.

Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot
Green: Sorted

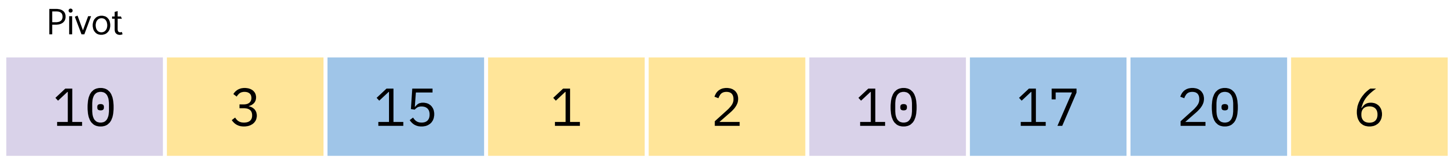
Hoare's Partitioning

Procedure:

1. Select your pivot point.
2. Initialize a left and right pointer to the left and right ends of your array respectively. Do not include the pointer when doing this initialization.
    a. The left pointer likes values that are smaller than the pivot
    b. The right pointer likes values that are larger than the pivot
    c. Note that both pointers dislike values that are equal to the pivot
3. Walk the left and right pointers toward each other.
    a. Stop moving a pointer once it reaches a value it DISLIKES
4. Once both pointers have stopped, swap the values they are currently pointing to then move both pointers forward by one.
5. Repeat 3-4 until the pointers have crossed, then you are done walking.
6. Swap the right pointer with the pivot

Hoare's Partitioning

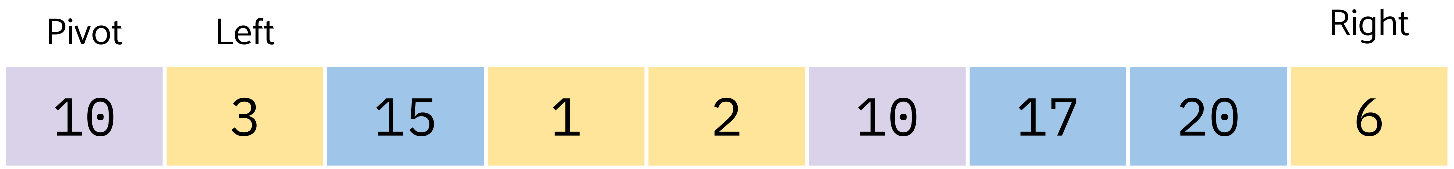
For simplicity, choose the leftmost element as pivot



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Hoare's Partitioning

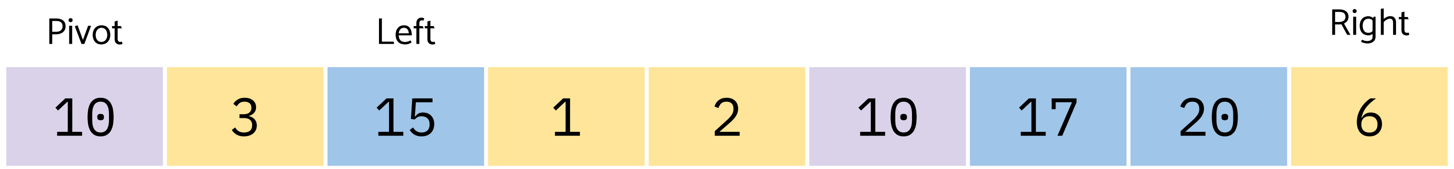
Initialize the left and right pointers at the left and right ends of the array (remember to exclude the pivot)



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Hoare's Partitioning

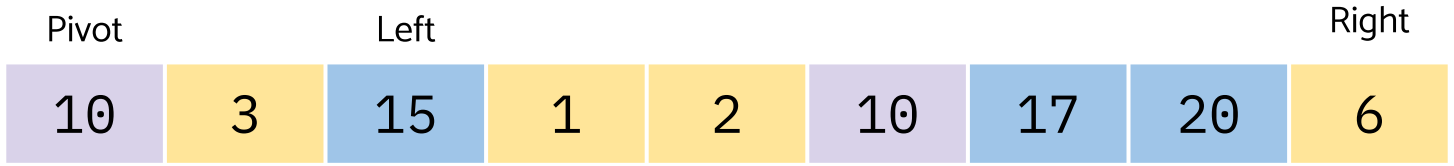
The left pointer likes the value 3 because it is less than the pivot, so it walks one value toward the right pointer. It dislikes 15 because it is larger than the pivot so the left pointer stops.



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Hoare's Partitioning

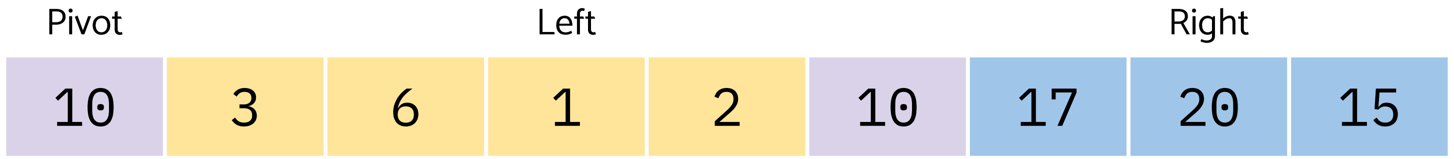
The right pointer dislikes the value 6 because it is smaller than the pivot, so it does not walk at all.



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Hoare's Partitioning

Swap the left and right pointer values since both have stopped walking and move both pointers forward by one.



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Hoare's Partitioning

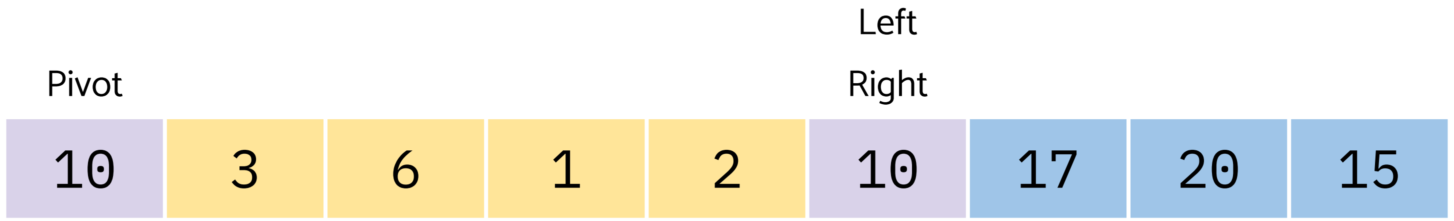
The left pointer likes both 1 and 2 so it continues walking. It dislikes the value 10, which is equal to the pivot so it stops there.

Pivot					Left		Right	
10	3	6	1	2	10	17	20	15

Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Hoare's Partitioning

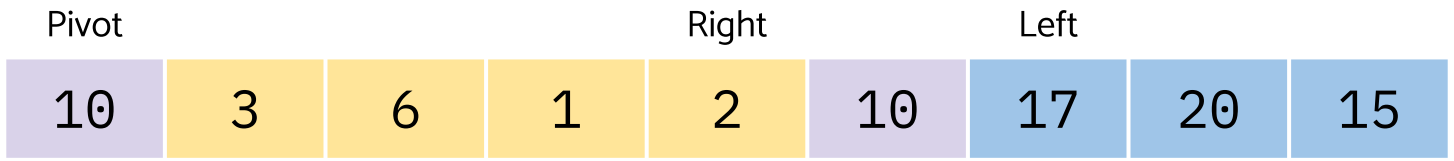
The right pointer likes both 20 and 17, so it continues walking until it reaches 10, which it dislikes.



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Hoare's Partitioning

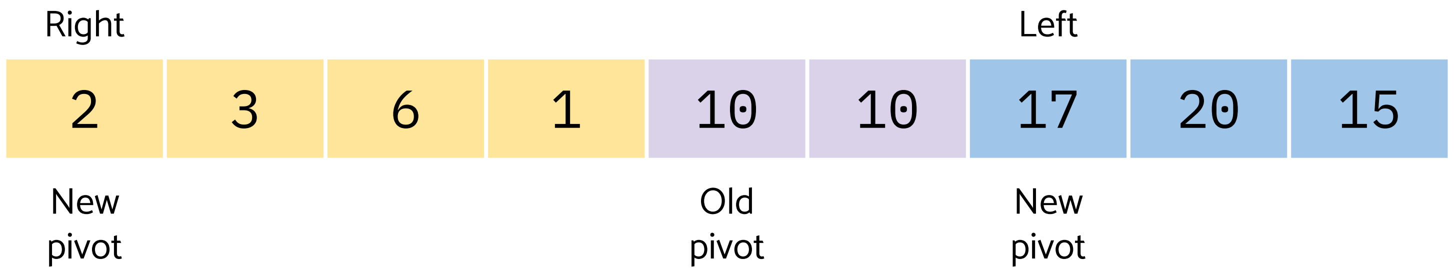
Because both pointers stopped, we try to swap, but it does nothing. We advanced both pointers by one. Now because the pointers have crossed, we are done walking.



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Hoare's Partitioning

Finally, swap the right pointer with the pivot. For our next iteration of the partitioning scheme, 2 and 17 will be the pivots because we select the leftmost element as our pivot



Purple: Pivot
Yellow: Less than pivot
Blue: Greater than pivot

Quick Sort: Summary

Runtime

- Best Case: Θ(NlogN)
- Worst Case: Θ(N^2)

Space Requirement: Θ(logN)

- Note that although some partitioning schemes are in-place and don't technically allocate additional memory, the space requirement comes from the recursive calls made while partitioning

Stable? Depends on partitioning scheme

- Hoare's: Not stable
- 3-scan: Stable

Example of worst case runtime:

- Pick leftmost element as pivot

1	2	3	4	5

Conclusion:

- Heapsort, Mergesort, and Quicksort emerge as the fastest sorts
- Quicksort has a slower worst-case runtime than Mergesort, but it is empirically faster
  - Used in Java’s own sort!
- Mergesort is “stable” but Quicksort is not
- Insertion sort is ideal for arrays of size < 15
- We cannot do better than a worst-case runtime of **Θ(n log n)** for comparison sorts

Appendix

More notes can be found at: https://inst.eecs.berkeley.edu/~cs61b/sp20/docs/sorting_notes.html