**Bubble Sort**

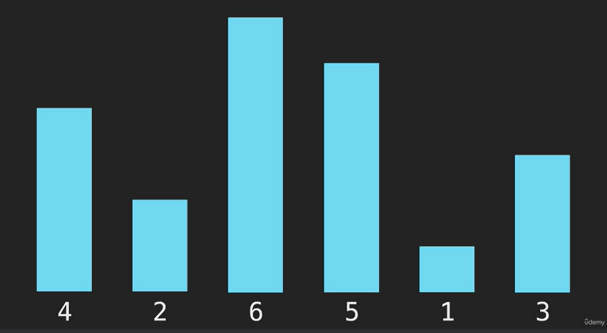
Starting from the first item, we compare that with the next item in the array.

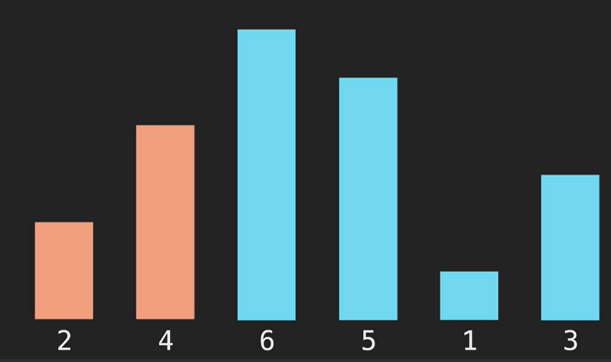
If the next item is bigger, we swap the items.

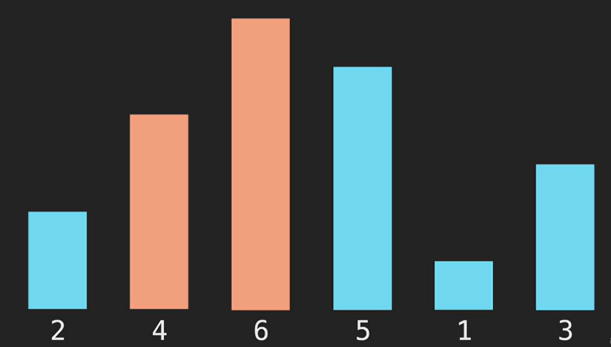
Once we get to the end of the array, the largest item will have been the last item.

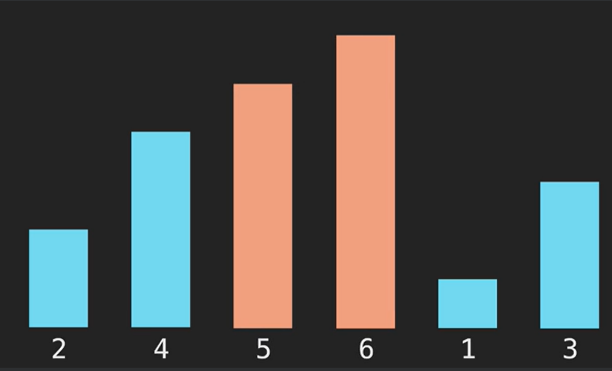
Then we repeat these steps again starting from the beginning of the array.

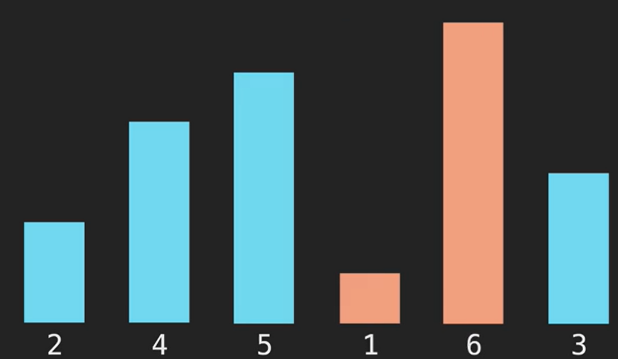
We do this until all the items are sorted.



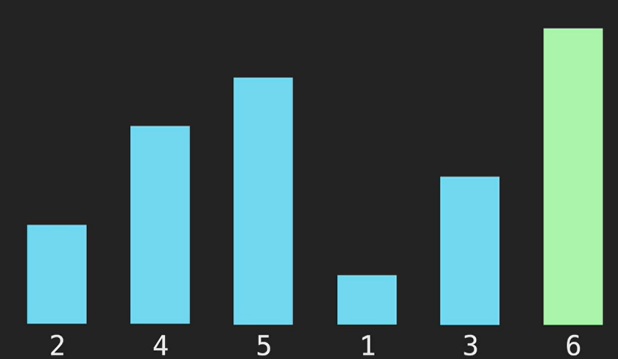




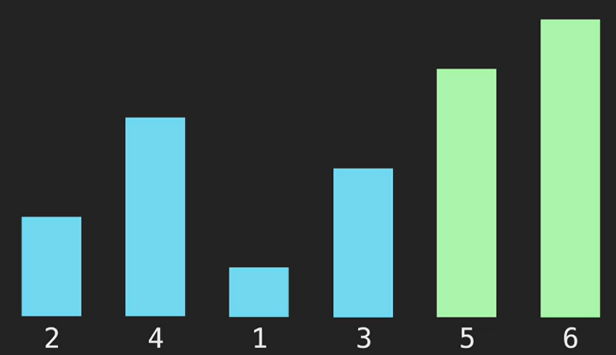


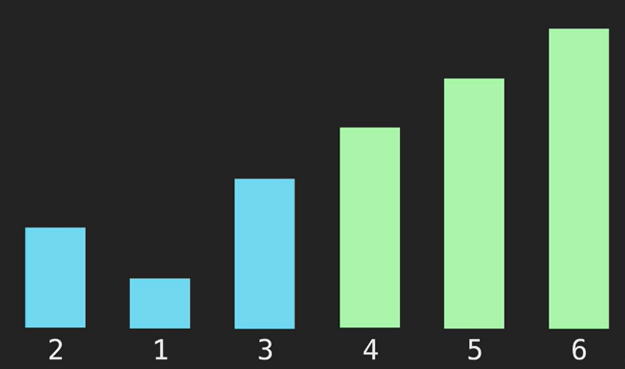


Now 6 is sorted

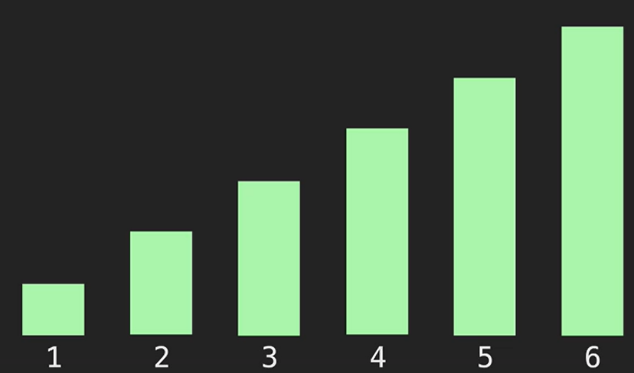


We repeat starting from beginning.





Do this until the array is sorted.



function bubbleSort(array) {

    for (let i = array.length - 1; i > 0; i--) {

        for (let j = 0; j < i; j++) {

            if (array[j] > array[j + 1]) {

                let temp = array[j];

                array[j] = array[j + 1];

                array[j + 1] = temp;

            }

        }

    }

    return array;

}

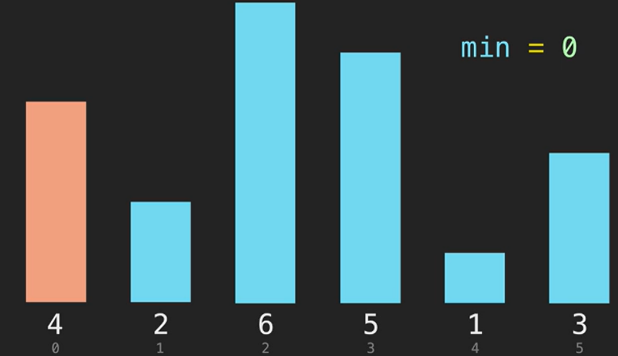
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Selection Sort**

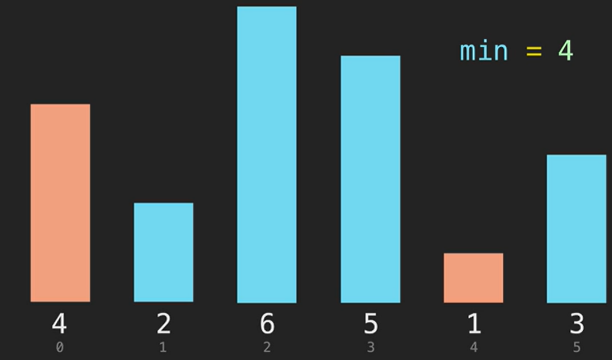
We use the array indices to sort.

Loop through the array and store the index of the lowest value in a variable called min.

Once we have a min value, we swap the value at the indices.

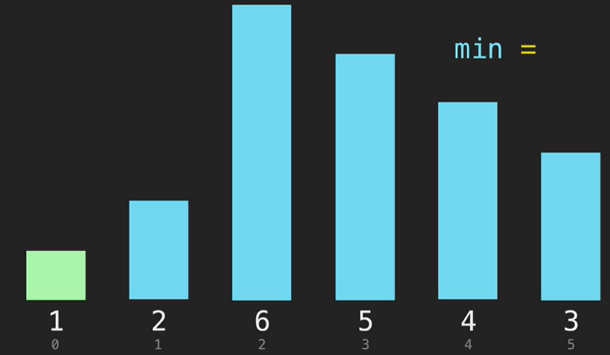


Once we loop through on the first iteration, min is index 4

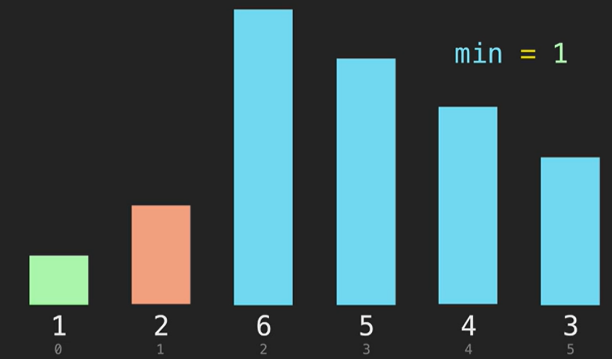


Once we find the min value on the first iteration, we switch the value at index 0 with the value at the min index.

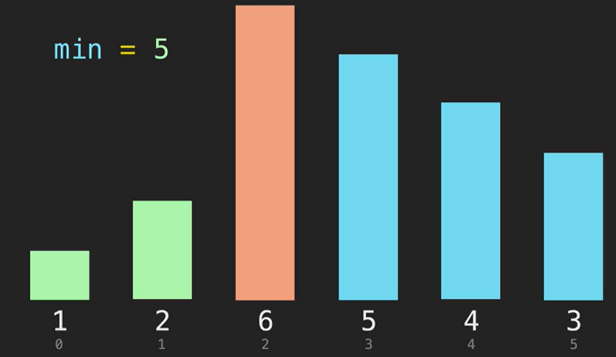
Here, the values at index 0 and index 4 are switched.

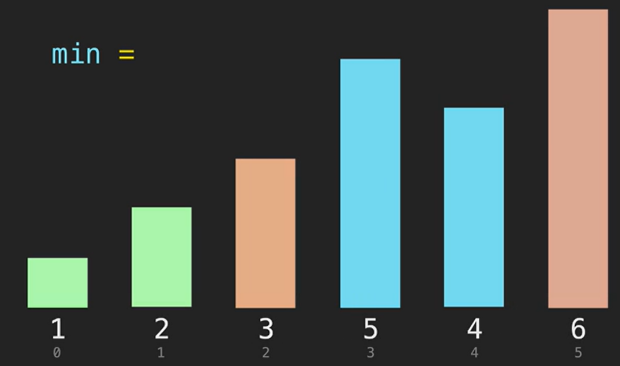


On second iteration, we start at index 1 and set that to the min.

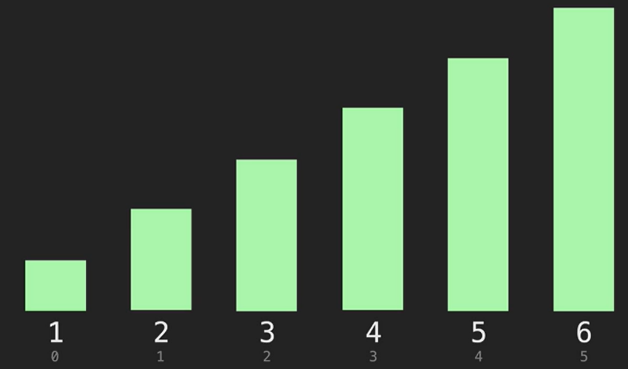


Iterate through the array again and repeat the steps swapping the min value.





Repeat until the array is sorted.



function selectionSort(array) {

    for (let i = 0; i < array.length - 1; i++) {

        let min = i;

        for (let j = i + 1; j < array.length; j++) {

            if (array[j] < array[min]) {

                min = j;

            }

        }

        if (i !== min) { // If the value at i is already the min, do not swap anything.

            let temp = array[i];

            array[i] = array[min];

            array[min] = temp;

        }

    }

    return array;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Insertion Sort**

We start at the second item in the array.

Compare it to the item before it.

If the second item is lower than the first item, then swap them.

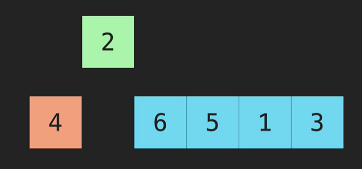
Now we move onto the third item in the array.

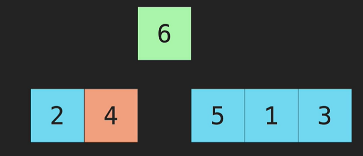
Compare it to the item before it.

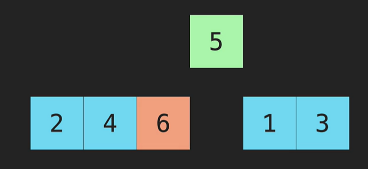
If it is lower, then swap the items.

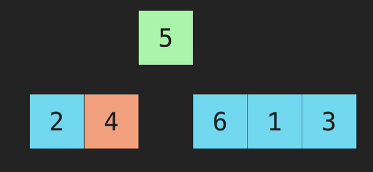
If not, then compare it to the item before that one until you reach the beginning of the array.

If it is lower than any item during the iteration, then swap those items.

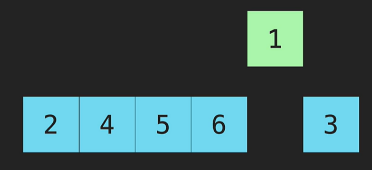


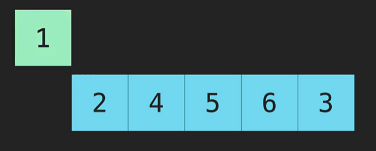


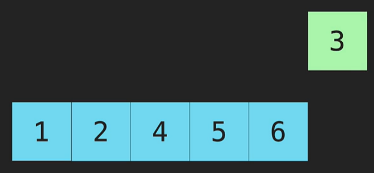


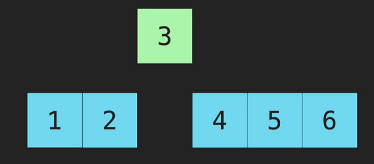














function insertionSort(array) {

    for (let i = 1; i < array.length; i++) { // i=1 because we start at the second item in array

        let temp = array[i];

        for (var j = i - 1; array[j] > temp && j > -1; j--) { // we use var j-1 because we need j to be accessible outside of that forloop. Using ‘let’ will not allow us to access j outside of forloop. j > -1 means that when j is at index[0] and when we j--, j would be at index[-1]. This would be out of scope of array and we stop running the forloop and the next line, array[j+1] would be array[0].

            array[j + 1] = array[j];

        }

        array[j + 1] = temp;

    }

    return array;

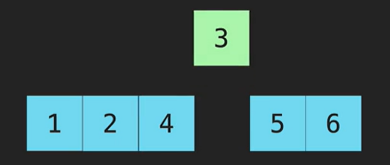
}

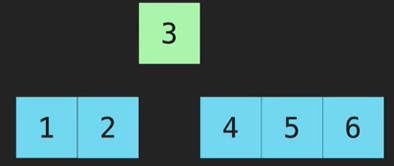
**Big O (Insertion Sort)**

Wost case = O(n^2) because of the nested for loops.

What is the array is almost sorted?









Here we only needed to move the 3.

We only needed to iterate through the array 1 time and the Big O = O(n).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Merge Sort**

Uses recursion.

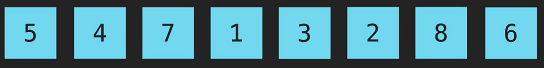
Very efficient sorting algorithm.

Merge sort will take an array, then cut in half into 2 separate arrays, then repeat until each item is its own array with only 1 item.

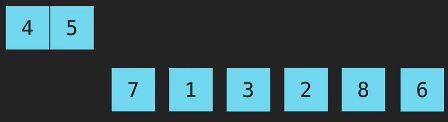








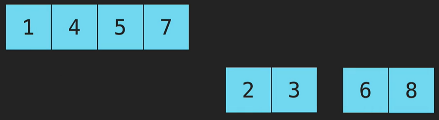
It will then take 2 arrays and combine them together to make a sorted array and repeat until it combines all the items into 1 sorted array.















Merge Code

function merge(array1, array2) {

    let combined = [];

    let i = 0;

    let j = 0;

    while (i < array1.length && j < array2.length) {

        if (array1[i] < array2[j]) {

            combined.push(array1[i]);

            i++;

        } else {

            combined.push(array2[j]);

            j++;

        }

    }

    while (i < array1.length) {

        combined.push(array1[i]);

        i++;

    }

    while (j < array2.length) {

        combined.push(array2[j]);

        j++;

    }

    return combined;

}

Merge Sort Code

1. Breaks arrays in half.
2. Base case: when array.length is 1.
3. Uses merge() to put arrays back together.

function merge(array1, array2) {

    let combined = [];

    let i = 0;

    let j = 0;

    while (i < array1.length && j < array2.length) {

        if (array1[i] < array2[j]) {

            combined.push(array1[i]);

            i++;

        } else {

            combined.push(array2[j]);

            j++;

        }

    }

    while (i < array1.length) {

        combined.push(array1[i]);

        i++;

    }

    while (j < array2.length) {

        combined.push(array2[j]);

        j++;

    }

    return combined;

}

function mergeSort(array) {

    if (array.length === 1) return array;

    let midIndex = Math.floor(array.length / 2);

    let left = mergeSort(array.slice(0, midIndex));

    let right = mergeSort(array.slice(midIndex));

    return merge(left, right);

}

**Big O (Merge Sort)**

Space complexity is O(n) because if an array has 8 items, you would have to break the array down to 8 arrays containing 1 item.

Time complexity

* Breaking an array with 8 items down to 8 individual array containing 1 item takes 3 steps resulting in O*(log n).*
* Putting the items back together in sorted pairs at each step means that we would have to iterate through each array then merge them as pairs resulting in *O(n).*
* This results in big O of O(n log n)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Quick Sort**

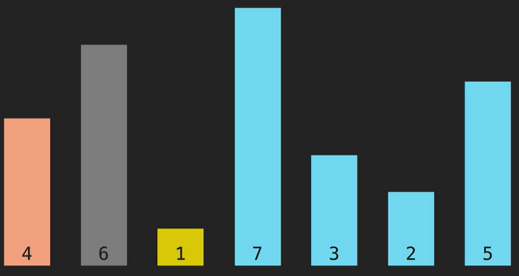
We start out with a pivot which is the first item in the array.

Then we iterate through the array and compare the values.

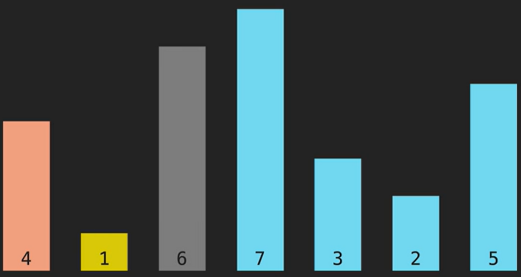
If the item is greater than the pivot, we mark it gray.

If the item is less than the pivot, we swap it with the first item that is greater than the pivot marked in gray.

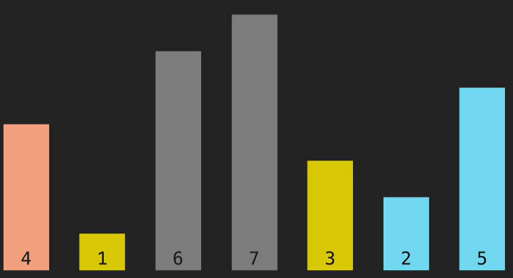
Here, the pivot is 4, 6 is greater than the pivot so it’s gray, 1 is less than the pivot so we mark it yellow.



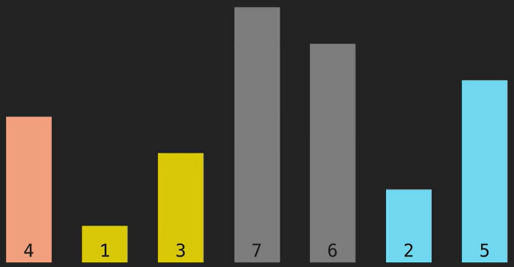
We swap the 1 and 6.



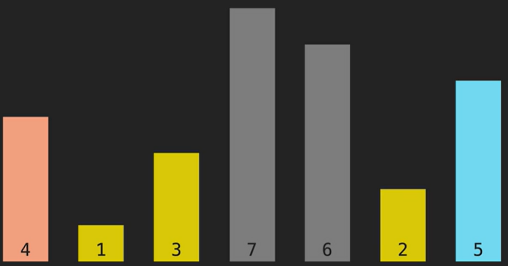
We continue to iterate. 7 is greater than the pivot (4). 3 is less than the pivot (4).

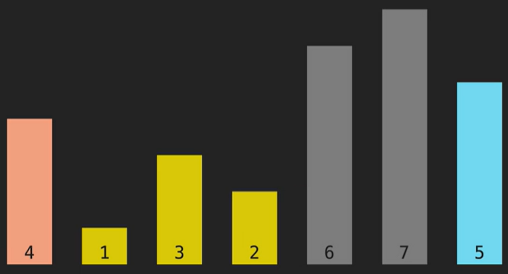


We swap 3 with the first item that is greater than the pivot (4) which is 6.

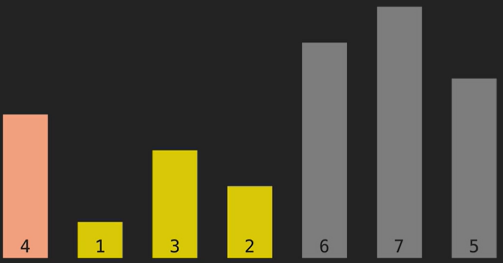


2 is less than the pivot so we swap 2 and 7.

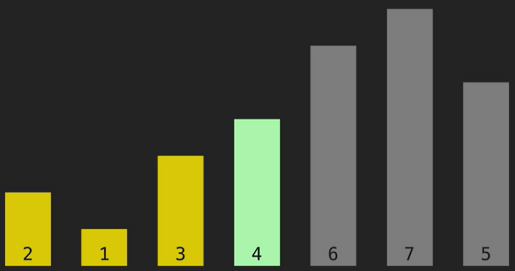




5 is greater than the pivot so we are done with the first iteration of the array.



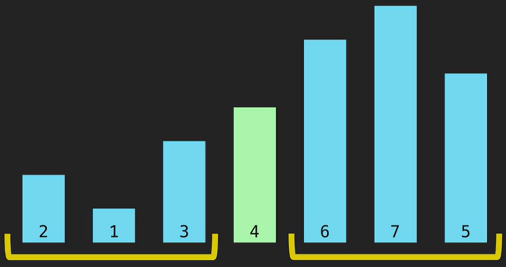
Now we swap the pivot (4) with the last item that was less than the pivot which is 2.



Now the 4 is exactly where it should be in the array.

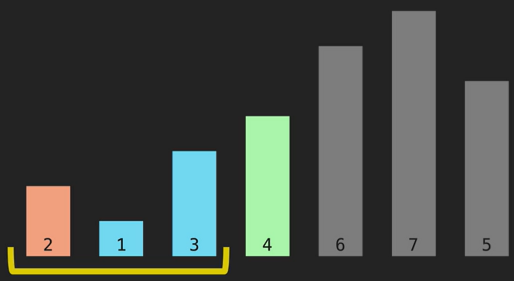
Everything less than 4 is on the left and everything greater than 4 is on the right.

We will now repeat the same operation on the items to the left of the 4 and the items to the right of the 4.

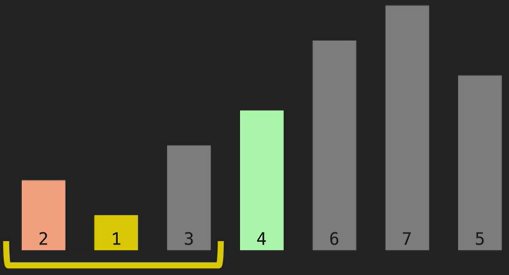


Starting on the left side.

Make the 2 the pivot.

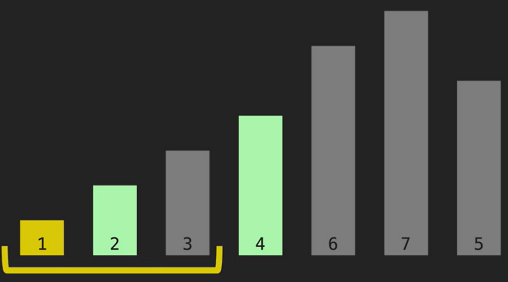


1 is less than pivot (2) and 3 is greater than pivot (2)



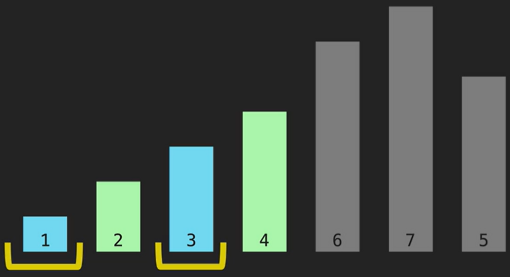
Now we are done with the left side.

Swap the pivot (2) with the last item that is lower than it which is 1.

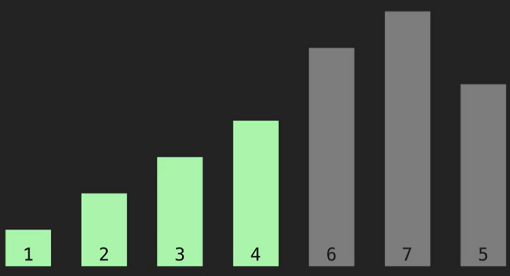


Now the 2 is sorted.

Now we perform the same operation on the left and right side of the 2.

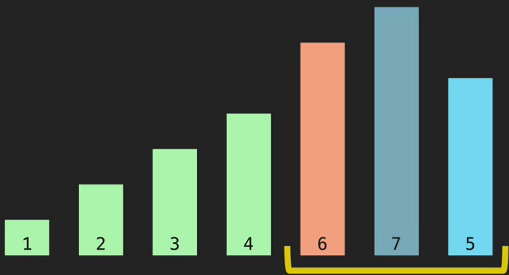


Since there are only 1 item in left and right side, they are already sorted.

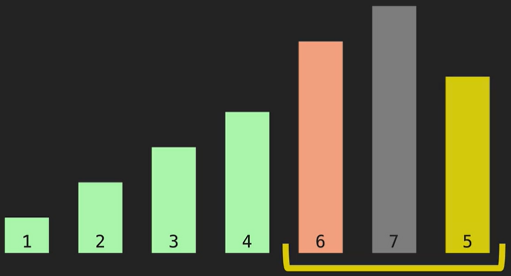


Now we work on the right side.

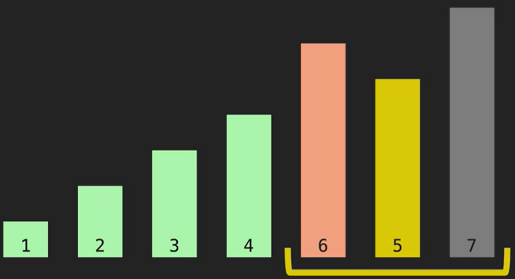
6 is our pivot.



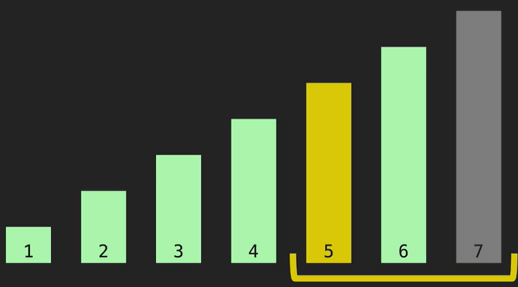
7 is greater than the pivot (6) and 5 is less than the pivot (6).



Swap the 5 with the first lowest item which is 7.

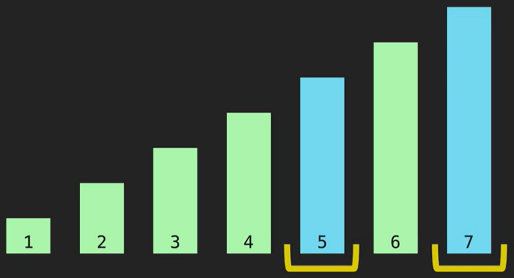


Swap the pivot (6) with the last lowest item which is 5.

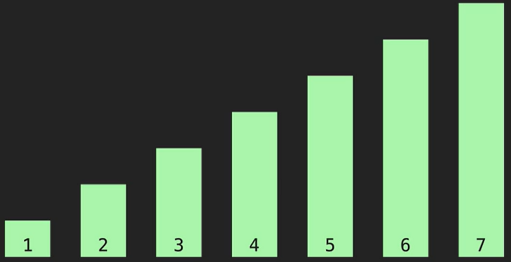


Now the 6 is sorted and where it should be in the array.

Now we repeat the steps for the left and right side of the 6.



Since there is only 1 item on the left and right side of the 6, we know that they are sorted as well.

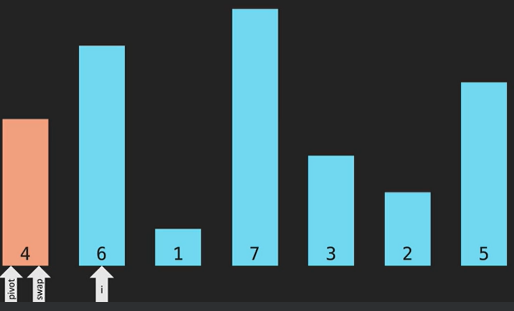


Now we are done.

**Pivot**

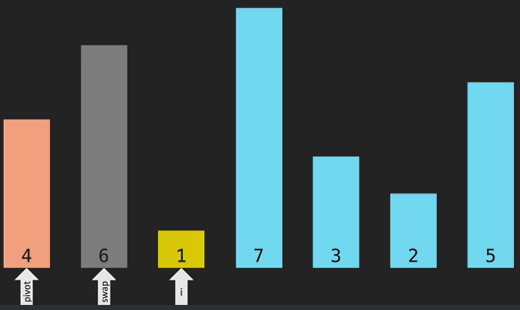
Set the pivot and swap variables to the first item in the array.

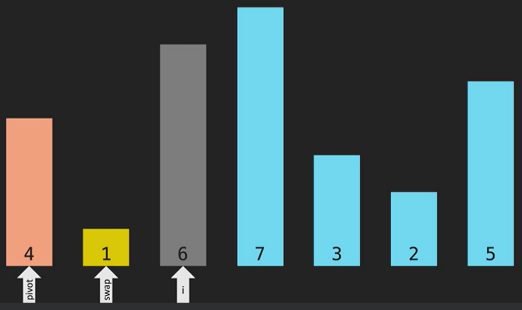
Set the variable i to the second item in the array.



Iterate through the array using the variable i.

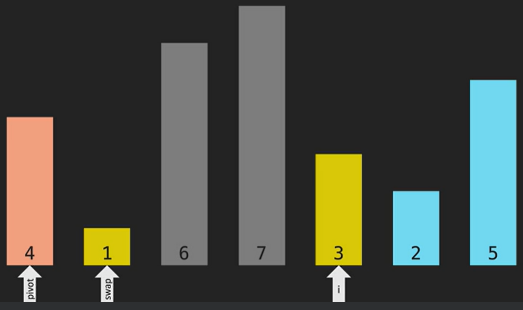
Whenever there is an item that is less than the pivot, we move swap up by one and swap the values of swap and i.

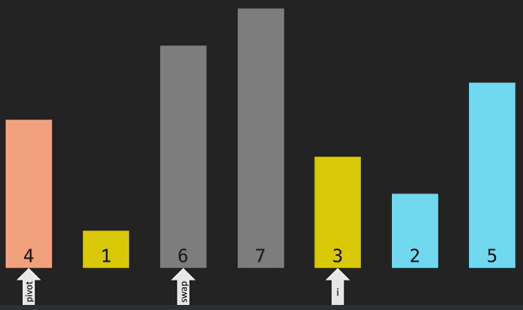


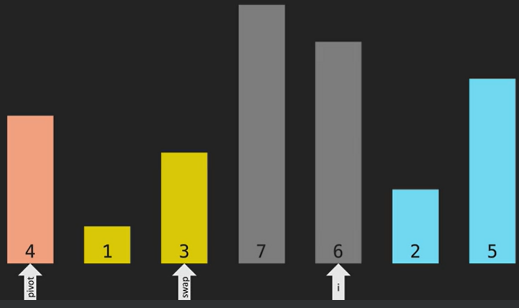


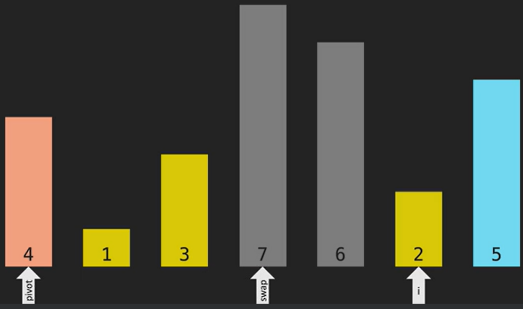
Continue to iterate until you reach the next item that is less than the pivot.

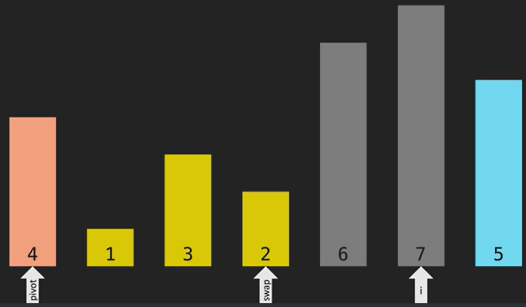
Move swap up by one and swap that item with i.



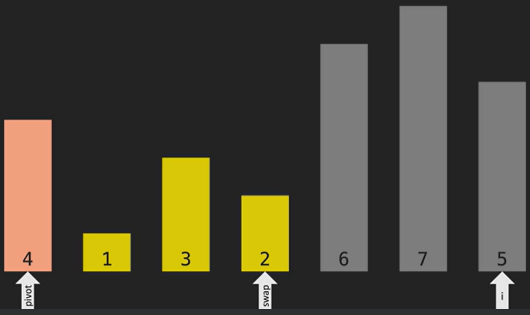




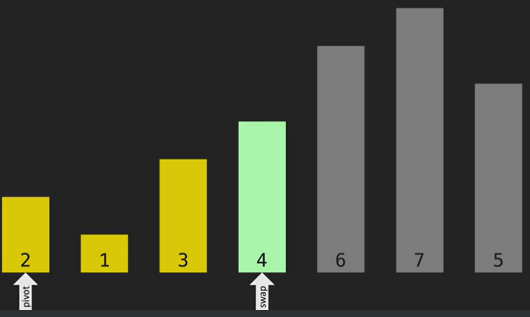




Now we are done with the first iteration.



Now we swap the pivot (4) with the last item that is less than the pivot which is 2.



Once we are done, we will use that swap variable to iterate through the left and right side.

left side = beginning of array to swap – 1

right side = swap + 1 to end of array

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*