Sorting Algorithms - Note Set 1

Introduction

Sorting Algorithms are crucial for all Software Developers. On surface level, these algorithms find and evaluate ways to sort arrays or lists of data in the most efficient manner and are often examined in interviews.

However, they also hit at the core of Software Development which is **Problem Solving** and finding **effective** solutions.

Consider the *Bubble Sort* Algorithm to come, it is a valid solution that can reliably sort a range of values in order. If we already have an intuitive means to do this, then why do we need more algorithms?

In Software Development, just because a solution solves a problem does not imply by any means that the solution is good. Solving a problem is the absolute, bear minimum requirement of a solution. For the solution to be considered 'good' it must meet a range of criteria such as:

- Efficient and Reliable
- Simple and non-repetitive
- Reusable and Scalable
- Well Designed and Thought Out
- Tested and Easy to Understand (with good documentation)

Sorting Algorithms and their study are excellent for developers to find ways to sort ranges of values in a manner that is efficient and reliable, but they also offer insight and practice in learning and remembering to write effective solutions for any problem.

1. Bubble Sort

The Bubble Sort Algorithm is generally one of the most intuitive, yet one of the least efficient ways to sort a list or range of values.

The Algorithm works by:

Keep count of iterations (run throughs of array or list):

- Starting with the first list position, compare it to the value in the next position. If first > second, swop them
- Move to the next position and compare to the next value, swop if first > second
- Keep doing this until you reach the end of the list or array

This counts only as the first iteration (count of 1)

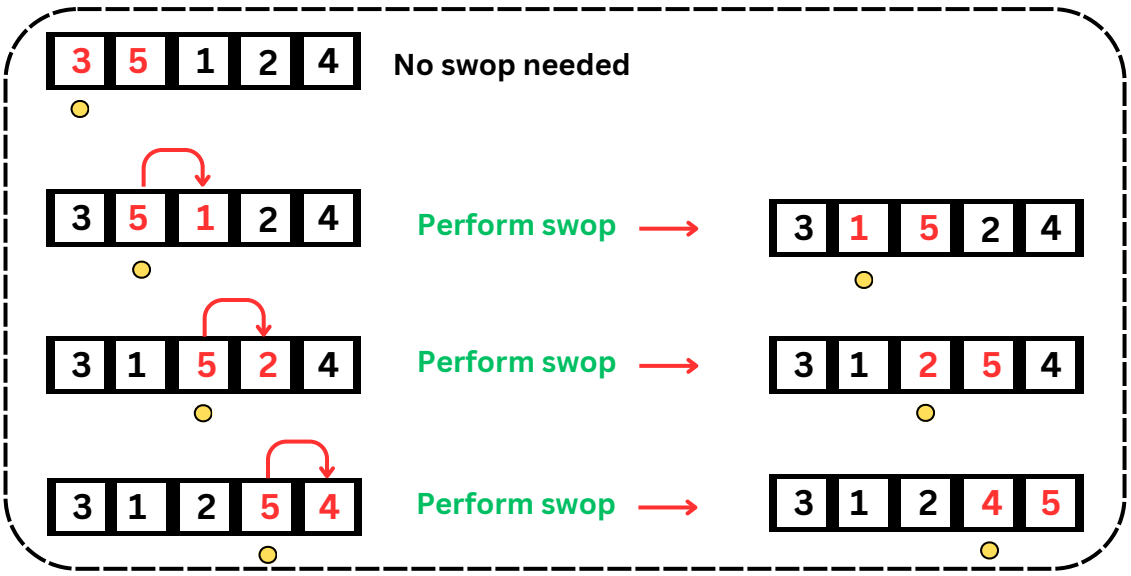
Repeat the entire process above until count equals the number of list items

The idea is to check every value and find its correct place in the list by checking which values it is greater than

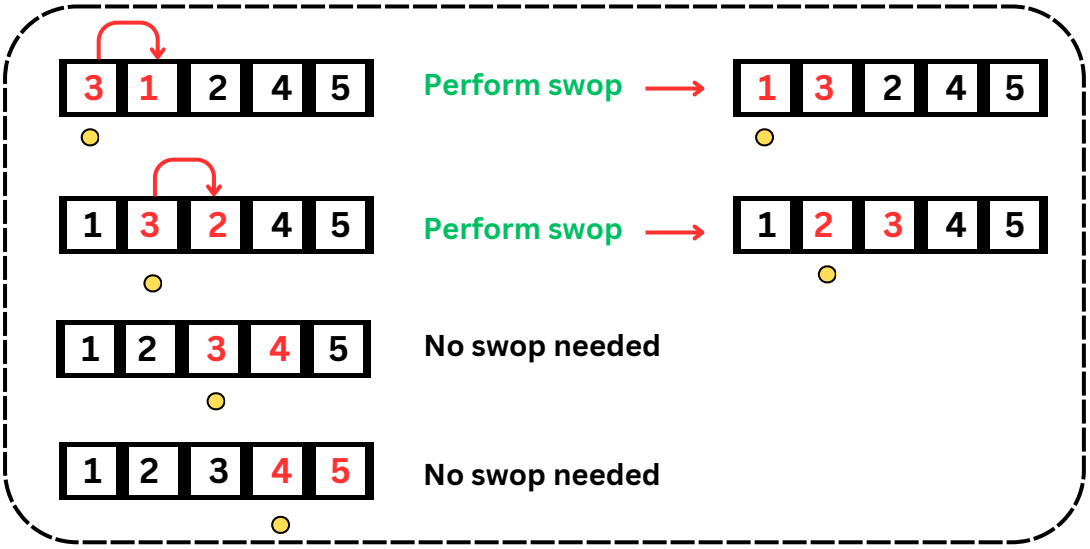
Consider: 3 5 1 2 4

There are five values, so we need to iterate five times (count goes from 0 - 4)

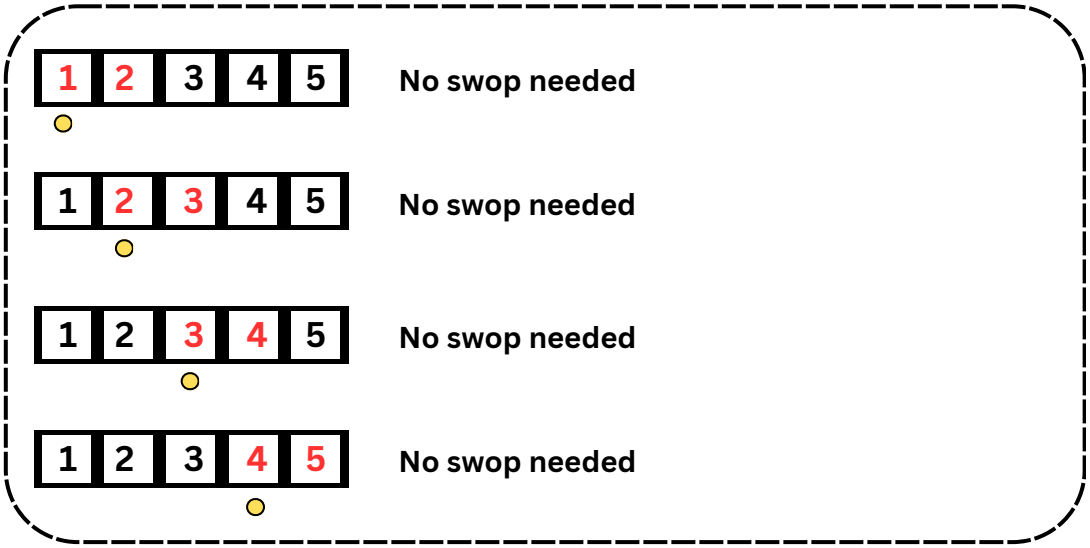
Iteration: 0



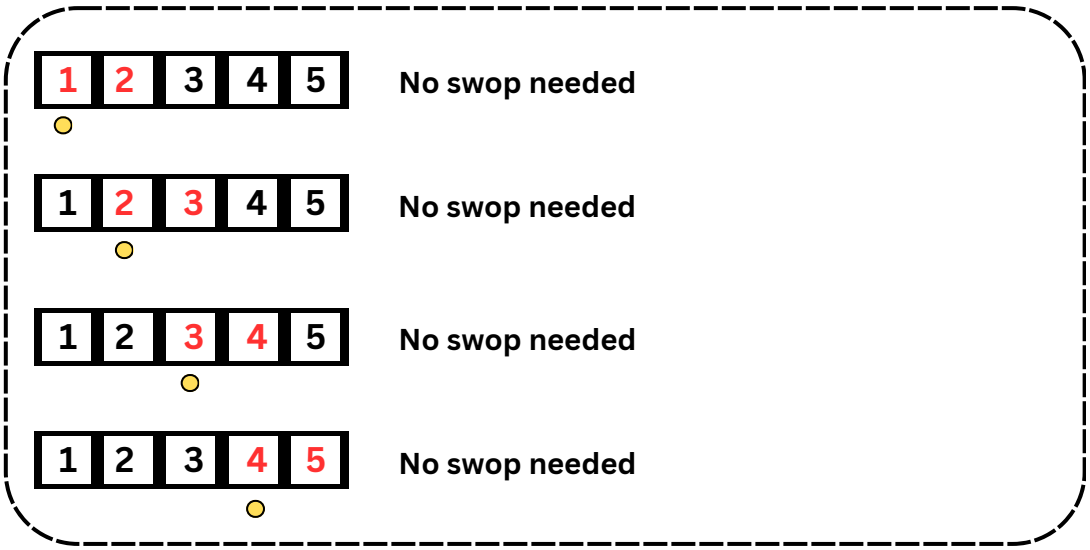
Iteration: 1



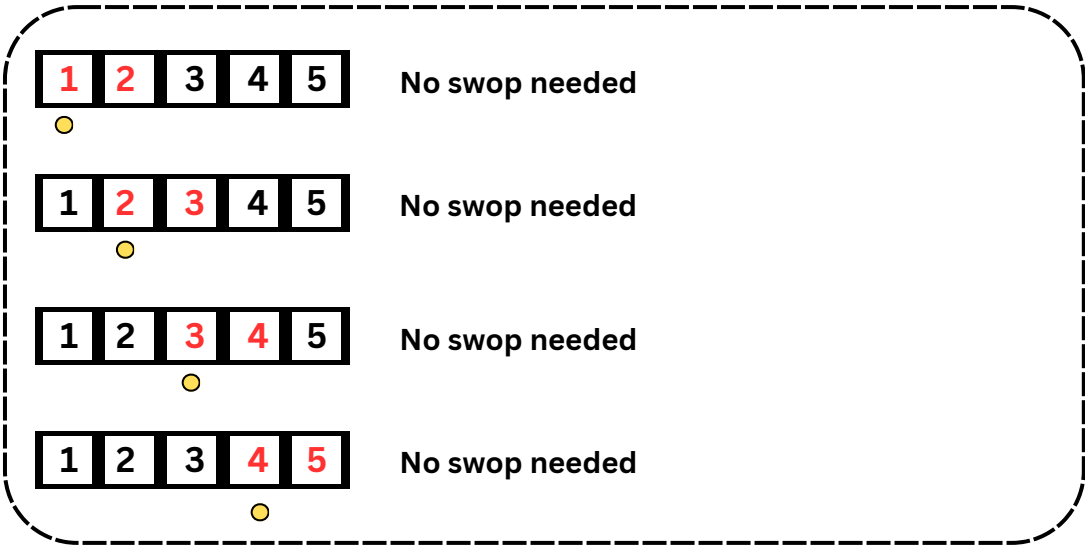
Iteration: 2



Iteration: 3



Iteration: 4



Did you notice that the list was sorted at the end of the second iteration, but we continued to perform three more?

This may not always occur, but this algorithm requires us to check every value against every other value (often repeatedly)

If we made the list even longer, we would increase the number of outer and inner iterations needed

IN CODE

Implementing this algorithm in code requires the use of nested for loops. For reference purposes:

- **i** represents the iteration counts in the diagrams above
- **j** represents the position index during these iterations (seen with the yellow dot) in the diagrams above

The solution below also implements the use of exception handling, but this is one of a few different ways to implement a bubble sort algorithm

Algorithm - Bubble Sort

```python
class BubbleSort():

    def sort(self, nums):
        # outer iterations
        for i in range(0, len(nums)):
            # inner iterations
            for j in range(0, len(nums)):

                # check if next value is greater than current
                try:
                    if nums[j] > nums[j + 1]:
                        # perform swop
                        nums[j], nums[j + 1] = nums[j + 1], nums[j]
                except IndexError:
                    continue

        # return sorted list
        return nums
```

Testing

```python
bubble_sort = BubbleSort()

# Test 1
first_num_list = [3, 5, 1, 2, 4]
print(f"Unsorted: {first_num_list} - Sorted: {bubble_sort.sort(first_num_list)}")

# Test 2
second_num_list = [0, 10, 8, 4, 5, 3, 4, 7, -8, 10]
print(f"Unsorted: {second_num_list} - Sorted: {bubble_sort.sort(second_num_list)}")

# Test 3
third_num_list = [2, 1]
print(f"Unsorted: {third_num_list} - Sorted: {bubble_sort.sort(third_num_list)}")
```

Results

```
Unsorted: [3, 5, 1, 2, 4] - Sorted: [1, 2, 3, 4, 5]
Unsorted: [0, 10, 8, 4, 5, 3, 4, 7, -8, 10] - Sorted: [-8, 0, 3, 4, 4, 5, 7, 8, 10, 10]
Unsorted: [2, 1] - Sorted: [1, 2]
```

2. Insertion Sort

The Insertion Sort Algorithm works by using a **partition** (a tracking position) from which we compare and sort values to the left of it. The goal is to sort values to the left, then increment the partition and then sort the left side of it again

The Algorithm works by:

Keep count of current index (partition) position starting from second value:

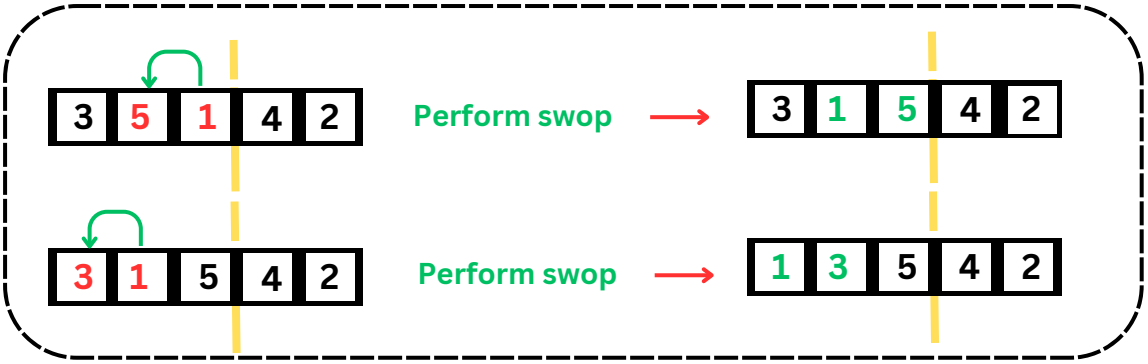
- Starting with second value, compare against the first and if second < first, perform a swap
- Move to the next (third) position and compare to the next value, swop if third < second**, then** perform comparison of second value with first and if second < first, perform a swop
- Keep doing this until the partition reaches the end of the array

Consider: 

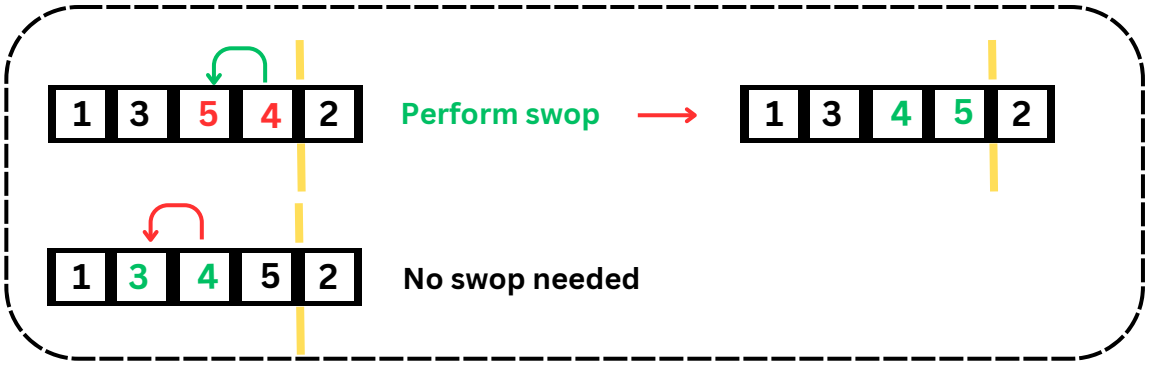
There are five values, so we need to move the partition four times

Partition 1

Partition 2

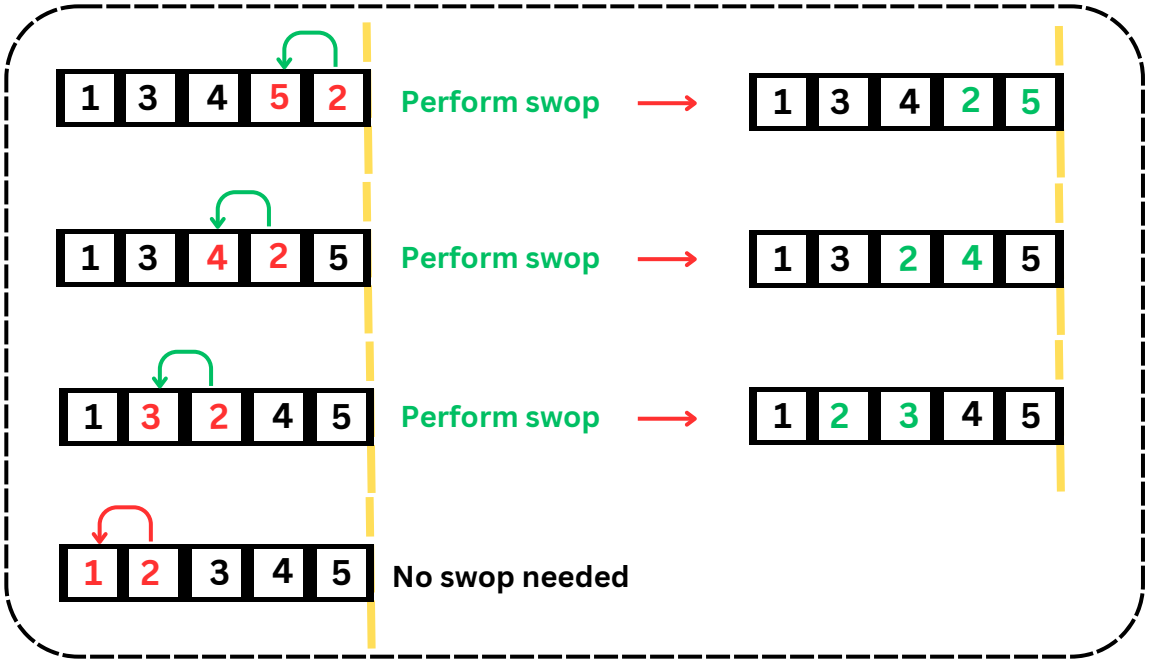


Partition 3



Note: we do not need to perform any check to the left of position 1 (value = 3) since we know this portion is already sorted
If 4 is not less than 3, then it will not be less than anything to the left of 3 either by the nature of the Insertion Sort Algorithm

Partition 4



IN CODE

Implementing this algorithm in code requires the use of nested for loops. For reference purposes:

- **i** represents the partition position starting from position 2 (index 1), seen with the yellow line
- **j** represents the inner comparisons from the left of **i**

Algorithm - Insertion Sort

With Optimization

```python
class InsertionSort():

    def sort(self, nums):
        # partition - from second item
        for i in range(1, len(nums)):

            # optimization - check if two swops are done
            # first is initial swop but does not mean new value is sorted to the left
            swop_count = 0

            # inner track - go backwards from current partition
            for j in list(range(0, i))[::-1]:

                # Swop already completed, terminate inner loop
                if swop_count == 2:
                    break

                if nums[j] > nums[j+1]:
                    # perform swop
                    nums[j], nums[j+1] = nums[j+1], nums[j]
                    swop_count += 1

        return nums
```

Testing

```python
insertion_sort = InsertionSort()

# Test 1
first_num_list = [3, 5, 1, 4, 2]
print(f"Unsorted: {first_num_list} - Sorted: {insertion_sort.sort(first_num_list)}")

# Test 2
second_num_list = [0, 10, 8, 4, 5, 3, 4, 7, -8, 10]
print(f"Unsorted: {second_num_list} - Sorted: {insertion_sort.sort(second_num_list)}")

# Test 3
third_num_list = [2, 1]
print(f"Unsorted: {third_num_list} - Sorted: {insertion_sort.sort(third_num_list)}")

# Test 4
fourth_num_list = [2]
print(f"Unsorted: {fourth_num_list} - Sorted: {insertion_sort.sort(fourth_num_list)}")
```

Results

```
Unsorted: [3, 5, 1, 2, 4] - Sorted: [1, 2, 3, 4, 5]
Unsorted: [0, 10, 8, 4, 5, 3, 4, 7, -8, 10] - Sorted: [0, 4, 5, 3, -8, 4, 7, 8, 10, 10]
Unsorted: [2, 1] - Sorted: [1, 2]
Unsorted: [2] - Sorted: [2]
```

3. Selection Sort

The Selection Sort Algorithm (like the Insertion Sort) uses a partition. To the left of the partition we have a sorted array, but selection sort works by repetitively looking for a minimum value in the unsorted portion of the array and then placing it in the sorted portion

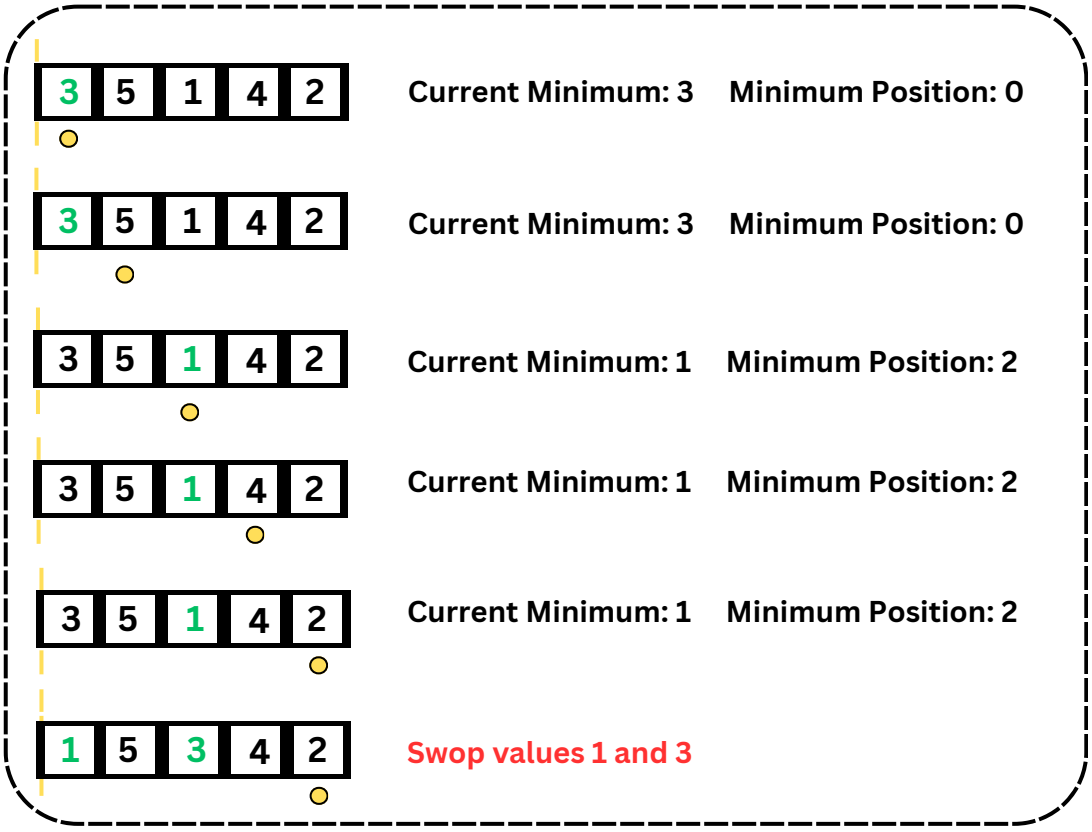
The Algorithm works by:

- Keep count of current index (partition) position starting from first value and keep track of current minimum:
- Iterate through right portion of array (unsorted) to find minimum value
- Swop minimum value with value at partition position
- Increment partition and repeat process

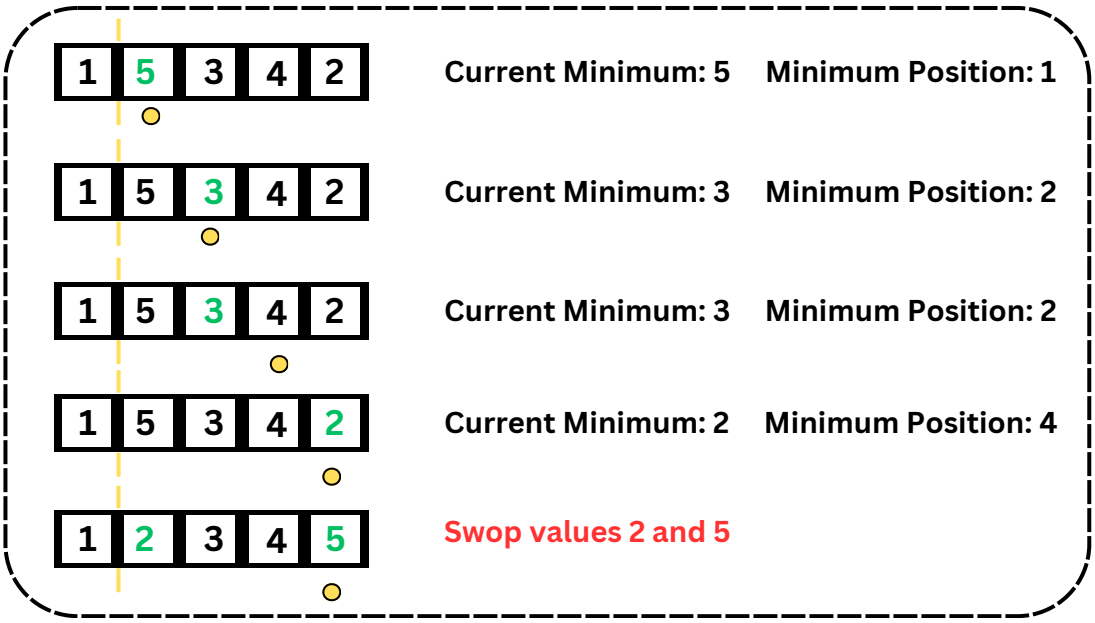
Consider: | 3 | 5 | 1 | 4 | 2 |

There are five values, so we need to set the partition four times starting from position 1 (index 0)

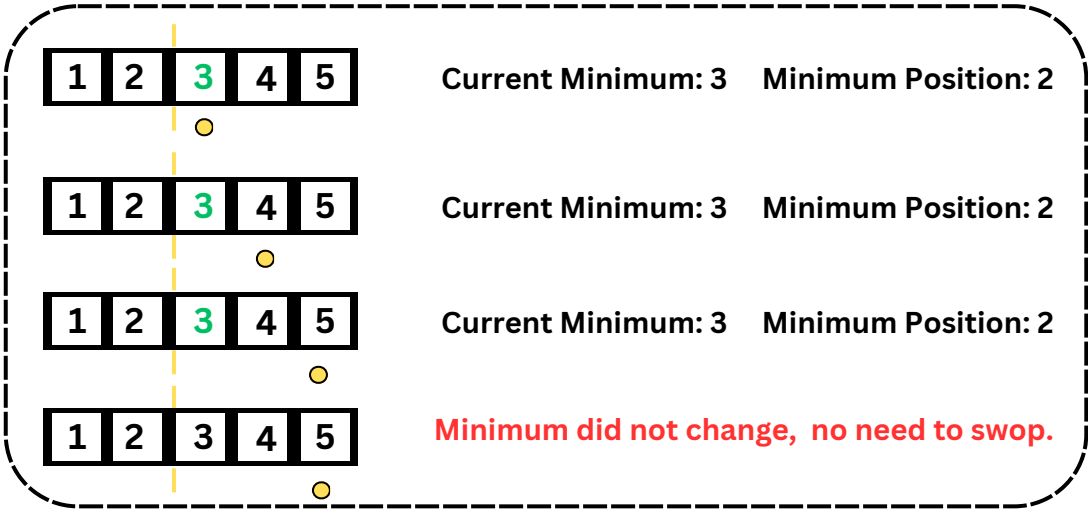
Partition 1



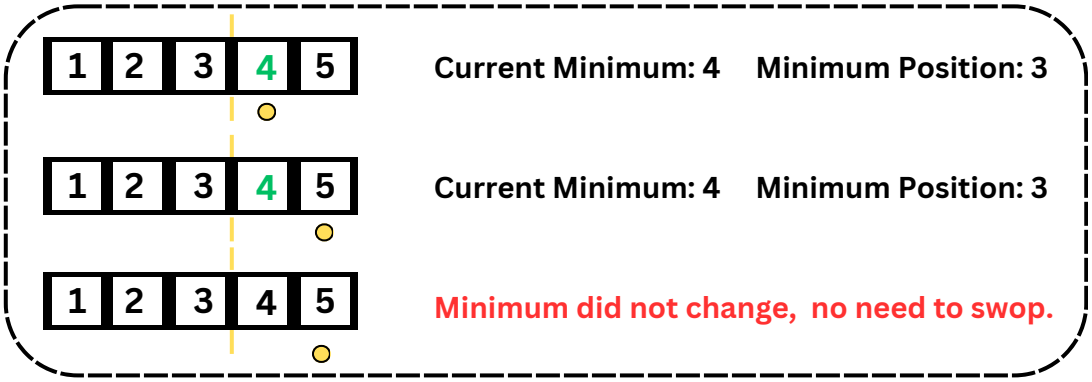
Partition 2



Partition 3



Partition 4



The last value would be sorted, and does not require the partition to move

IN CODE

Implementing this algorithm in code requires the use of nested for loops:

- **i** represents the partition position from index 1, seen with the yellow line in the diagrams
- **j** represents minimum tracker (yellow dot in diagrams)

Algorithm -Selection Sort

```
class SelectionSort():

    def sort(self, nums):

        # i tracks partition
        for i in range(0, len(nums)):

            # track index and value of current minimum
            min_index = i
            min_value = nums[i]

            # determine if swop is needed
            perform_swop = False

            # j tracks the current item in unsorted portion - right of partition
            for j in range(i + 1, len(nums)):

                if nums[j] < min_value:
                    min_value = nums[j]
                    min_index = j
                    perform_swop = True

            if perform_swop:
                nums[i], nums[min_index] = nums[min_index], nums[i]

        return nums
```

Testing

```
selection_sort = SelectionSort()

# Test 1
first_num_list = [3, 5, 1, 4, 2]
print(f"Unsorted: {first_num_list} - Sorted: {selection_sort.sort(first_num_list)}")

# Test 2
second_num_list = [0, 10, 8, 4, 5, 3, 4, 7, -8, 10]
print(f"Unsorted: {second_num_list} - Sorted: {selection_sort.sort(second_num_list)}")

# Test 3
third_num_list = [2, 1]
print(f"Unsorted: {third_num_list} - Sorted: {selection_sort.sort(third_num_list)}")

# Test 4
fourth_num_list = [2]
print(f"Unsorted: {fourth_num_list} - Sorted: {selection_sort.sort(fourth_num_list)}")
```

Results

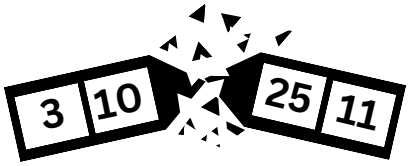
```
Unsorted: [3, 5, 1, 2, 4] - Sorted: [1, 2, 3, 4, 5]
Unsorted: [0, 10, 8, 4, 5, 3, 4, 7, -8, 10] - Sorted: [-8, 0, 3, 4, 4, 5, 7, 8, 10, 10]
Unsorted: [2, 1] - Sorted: [1, 2]
Unsorted: [2] - Sorted: [2]
```

Discussion

We notice that the Insertion and Quick Sort Algorithms both implement the use of a partition; the idea of separating sorted and unsorted portions of the list.

Whilst they are not incredibly efficient Algorithms in terms of speed, they do generally perform better than the Bubble Sort. We can see this in their diagrams noticing reductions in the number of internal iterations required.

Divide and Conquer



In the above algorithms, we continuously worked with the entire array or list, but to create algorithms that sort more efficiently, the next step is to stop considering the entire collection in every step, instead we want to break the collection of values down into smaller pieces and then rebuild

This is the basic idea of **divide and conquer**. We do note that not all divide and conquer algorithms will actually break the list apart (create new sub lists), but the concept of splitting the original list up into smaller components is still applied

4. Quick Sort

The Quick Sort Algorithm is a recursive, divide and conquer algorithm. This means that it will repetitively call itself (recursive) until a base condition is met (the size of the sub-array is 1). This algorithm is an in-place algorithm meaning it does not create any new subarrays or lists

The idea with this algorithm is to recursively use a pivot (a designated value) against which others values are compared and divided so that values to the left are smaller than the pivot, and values to the right are greater than or equal to it. A common implementation is to set the last value as the pivot.

Note: Prior knowledge of recursive functions is needed and is discussed in my 'Functions' Note Set available from my LinkedIn and GitHub Profiles.

Select last value in the array or list as the pivot against which all other values are compared

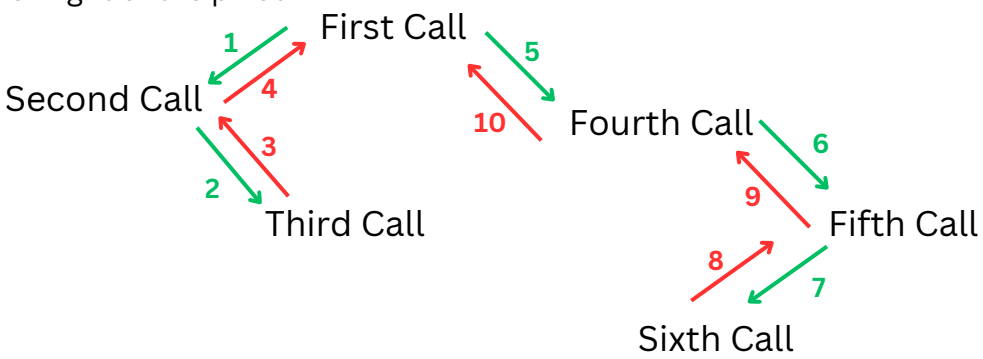
- Set a 'split variable' (called **i**) to indicate the separation. Those to the left of **i** are less than the pivot, those to the right are greater than or equal to the pivot
- Set a 'tracking variable' (called **j**) that will iterate through the array or list
- The logic for swopping the values is best seen in the diagrams below, but essentially we compare the pivot value against the value in the list at position j.
    - If the value is less than j, increment i and the swop the values at i and j
    - If the value is greater than or equal to j then just increment j and do not swop anything
    - When j is at the position of the pivot, increment i and then swop the values at i and the pivot index, this will put the pivot into it's correct position
- Call quick sort function again to check sub lists to the left and right of the correctly placed pivot and repeat above process until the starting index is equal to the pivot (this indicates the array or list cannot be broken down further)

Consider:

8	3	2	5	1	6	7	4

The complete process of this alogorithm, combined with its recursive nature does make its implementation more complex to visualize. The diagrams below show **every** step this algorithm would follow to sort the array.

This summary diagram is meant to show a **very high level** overview of the logic order for the recursive calls for this list with consideration for sorting done to the left or right of the pivot



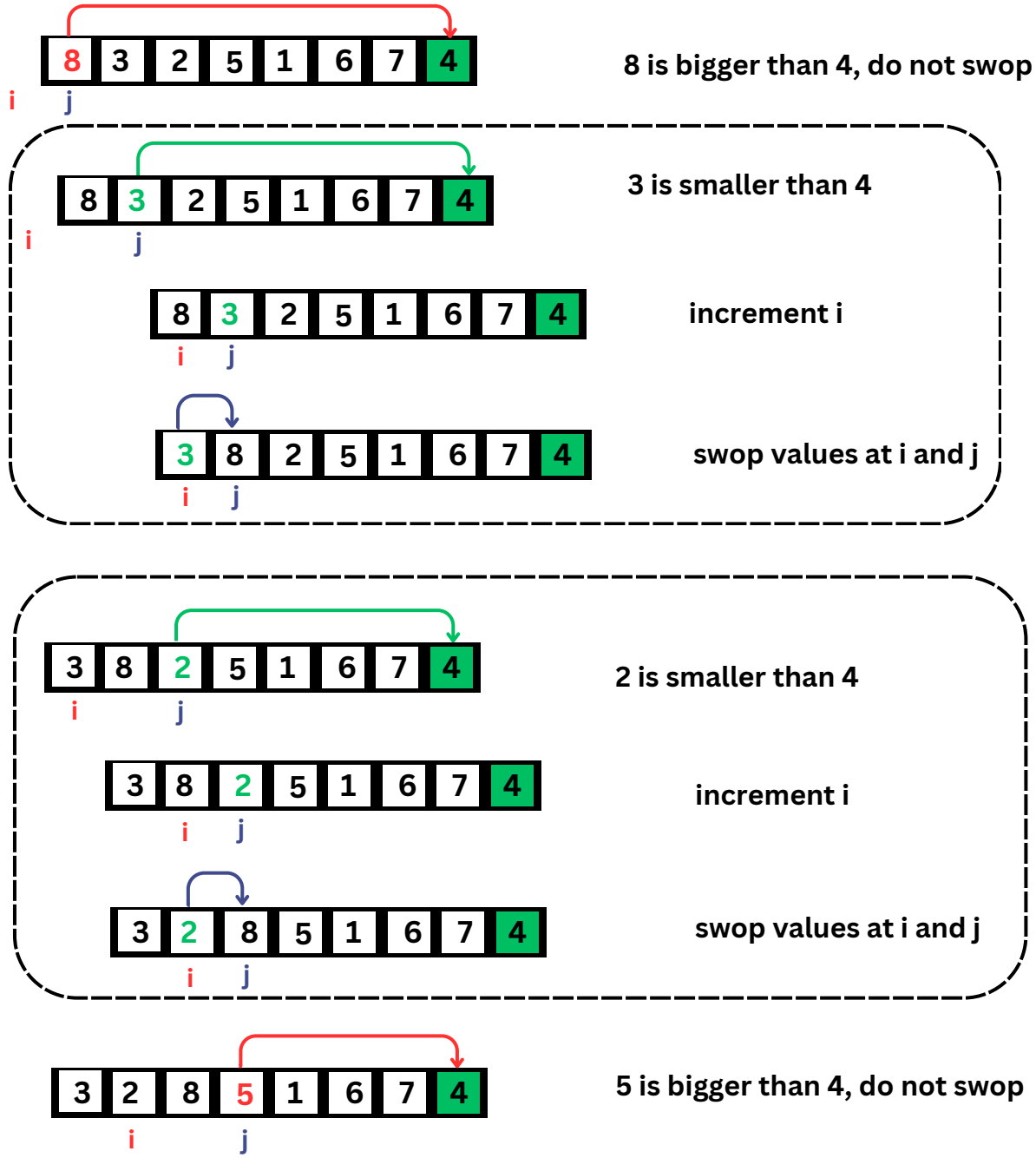
i = pivot placement track j = iterator p = pivot original index

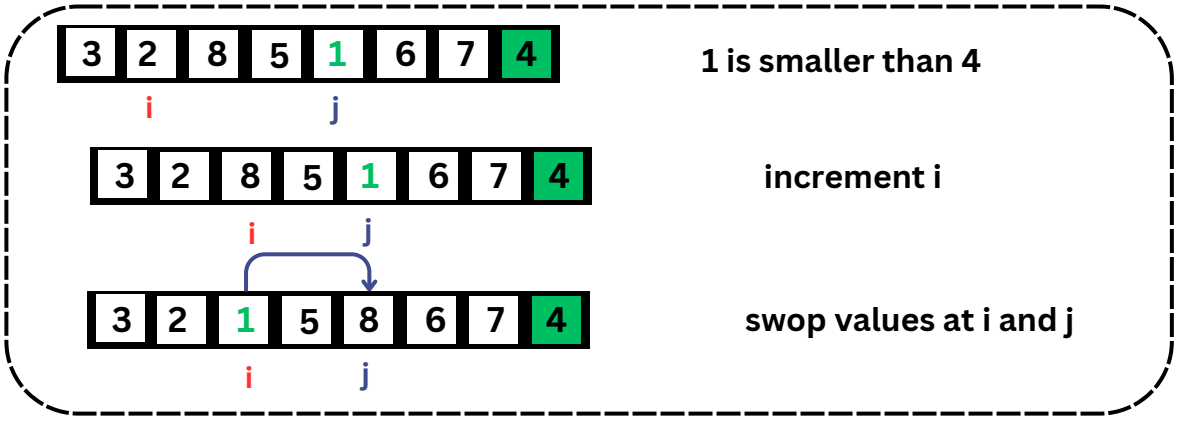
NOTE: 'j' is always incremented for each step and therefore not stated in every step below

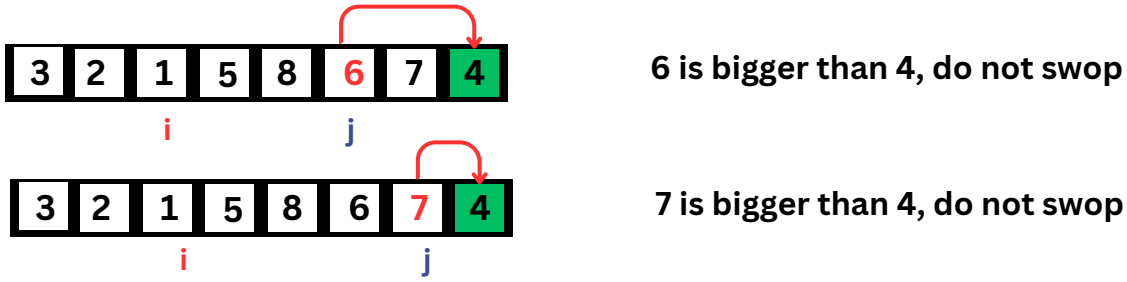
Pivot value = 4 Pivot index = 7 start_index = 0

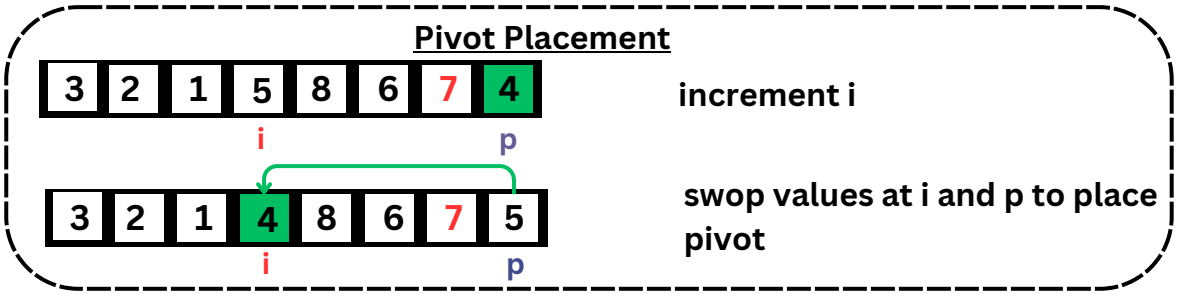
start index not equal to pivot index, proceed to sort

First Call

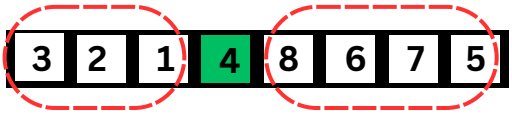








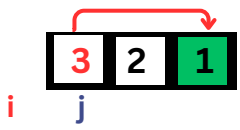
Go down left side first

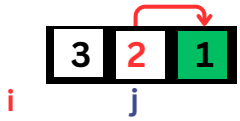


Pivot value = 1 **Pivot index = 2**
start_index = 0

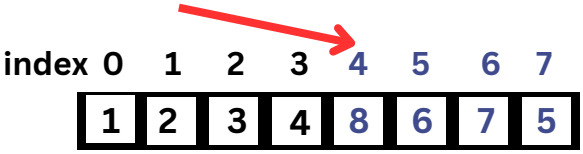
start index not equal to pivot index, proceed to sort

Second Call



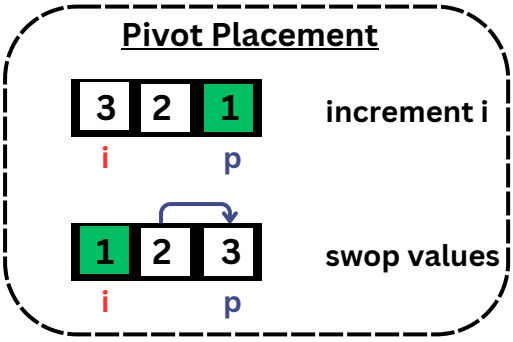


Note: This implementation performs these divisions using index positions of the original array or list. For sorting of the right side of the array, this means that index positions are indexed from the right (start index will **not** be zero)

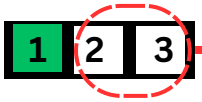


Pivot value = 5 **Pivot index = 7**
start_index = 4

start index not equal to pivot index, proceed to sort



Notice that there is no longer a portion to the left of the pivot, only the right side needs to be checked



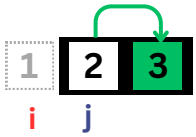
Pivot value = 3 Pivot index = 2
start_index = 1

start index not equal to pivot index, proceed to sort

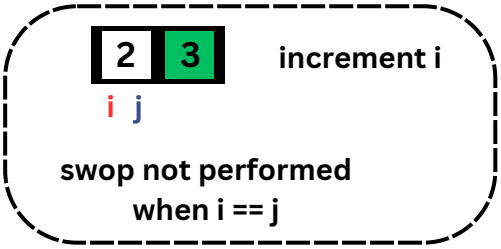
Third Call

In the diagrams below, you will see that evaluations are done, but there is no change to the order

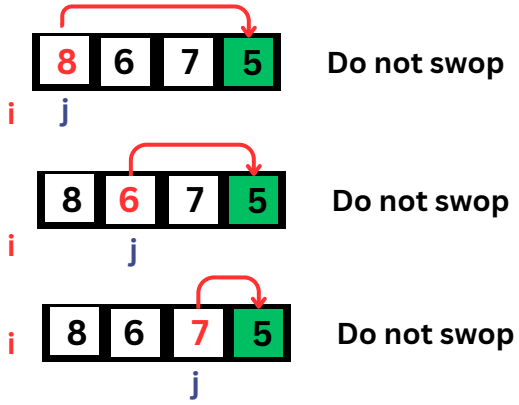
Whilst it may appear pointless, we must remember that this is a **recursive function** meaning that all calls up to the base condition follow the same logic

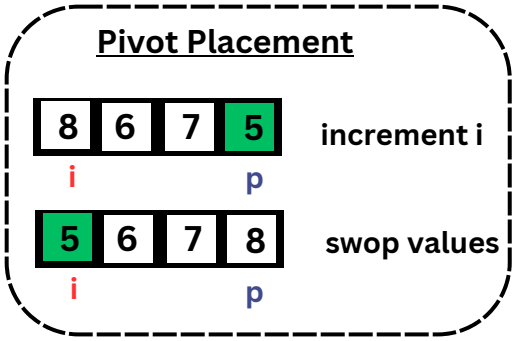


2 < 3

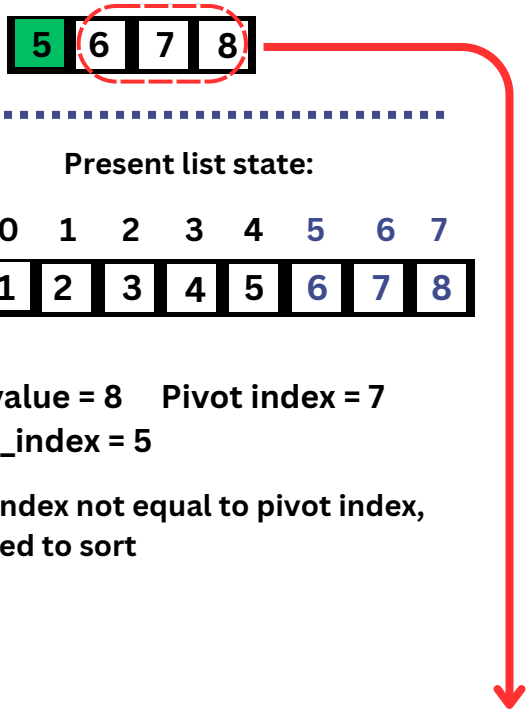


Fourth Call





Notice that there is not a portion to the left of the pivot, only the right side needs to be checked



Pivot value = 8 Pivot index = 7
start_index = 5

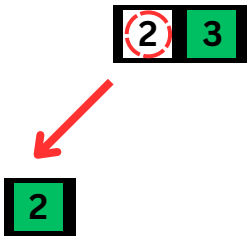
start index not equal to pivot index, proceed to sort

Pivot Placement

increment i

swop is not performed when i == p (seen in code)

Notice that there is no longer a portion to the right of the pivot, only the left side needs to be checked

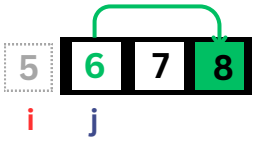


Pivot value = 2 Pivot index = 0
start_index = 0 end_index = 0

start index = pivot index

return back (through Recursive Calls 3 and 2) to First Call

Fifth Call



6 < 8

increment i

swop not performed when i == j

7 < 8

increment i

swop not performed when i == j

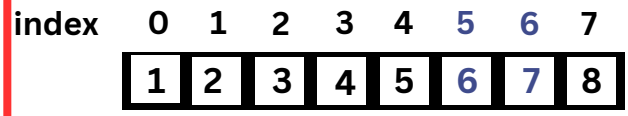
Pivot Placement

increment i

swop not performed when i == p

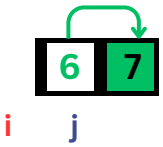
Notice that there is no longer a portion to the right of the pivot, only the left side needs to be checked

Present list state:



Pivot value = 7 Pivot index = 6
start_index = 5

Sixth Call



6 < 7

increment i

i j

swop not performed
when i == j

Pivot Placement

increment i

i p

swop not performed
when i == p

Notice that there is not a portion to the right of the pivot, only the left side needs to be checked

Present list state:

index	0	1	2	3	4	5	6	7
	1	2	3	4	5	6	7	8

Pivot value = 6 Pivot index = 5
start_index = 5

start index = pivot index

return back (through Recursive Calls 6, 5 and 4) back to First Call

```python
class QuickSort():

    def __init__(self):
        self.nums = None

    def sort(self, nums):
        self.nums = nums

        # call quick_sort (recursive function) for first time
        self.quick_sort(0, len(self.nums) - 1)
        return self.nums

    def quick_sort(self, start_index, pivot_index):

        # end condition
        if start_index == pivot_index:
            return

        else:

            # track pivot value
            pivot_val =  self.nums[pivot_index]

            # i must always be one to the left of the start
            i = start_index - 1
            for j in range(start_index, pivot_index):
                if self.nums[j] < pivot_val:
                    # i must be incremented before swop
                    i += 1
                    if i != j:
                        # perform swop
                        self.nums[i], self.nums[j] = self.nums[j], self.nums[i]

            # place pivot in correct position
            i += 1
            if i != pivot_index:
                self.nums[i], self.nums[pivot_index] = self.nums[pivot_index], self.nums[i]

            # attempt sort of left sub_array
            if i != start_index:
                self.quick_sort(start_index, i - 1)

            # # attempt sort of right sub_array
            if i + 1 <= pivot_index:
                self.quick_sort(i + 1 , pivot_index)
```

```
# Test 1
first_num_list = [3, 5, 1, 2, 4]
print(f"Unsorted: {first_num_list} - Sorted: {quick_sort.sort(first_num_list)}")

# Test 2
second_num_list = [0, 10, 8, 4, 5, 3, 4, 7, -8, 10]
print(f"Unsorted: {second_num_list} - Sorted: {quick_sort.sort(second_num_list)}")

# Test 3
third_num_list = [2, 1]
print(f"Unsorted: {third_num_list} - Sorted: {quick_sort.sort(third_num_list)}")

# Test 4
fourth_num_list = [2]
print(f"Unsorted: {fourth_num_list} - Sorted: {quick_sort.sort(fourth_num_list)}")

# Test 5
fifth_num_list = [8, 3, 2, 5, 1, 6, 7, 4]
print(f"Unsorted: {fifth_num_list} - Sorted: {quick_sort.sort(fifth_num_list)}")

# Test 6
sixth_num_list = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
print(f"Unsorted: {sixth_num_list} - Sorted: {quick_sort.sort(sixth_num_list)}")

# Test 7 - Given sorted array
seventh_num_list = [-5, -4, -3, -2, -1, 0]
print(f"Unsorted: {seventh_num_list} - Sorted: {quick_sort.sort(seventh_num_list)}")
```

Results

```
Unsorted: [3, 5, 1, 2, 4] - Sorted: [1, 2, 3, 4, 5]
Unsorted: [0, 10, 8, 4, 5, 3, 4, 7, -8, 10] - Sorted: [-8, 0, 3, 4, 4, 5, 7, 8, 10, 10]
Unsorted: [2, 1] - Sorted: [1, 2]
Unsorted: [2] - Sorted: [2]
Unsorted: [8, 3, 2, 5, 1, 6, 7, 4] - Sorted: [1, 2, 3, 4, 5, 6, 7, 8]
Unsorted: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] - Sorted: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Unsorted: [-5, -4, -3, -2, -1, 0] - Sorted: [-5, -4, -3, -2, -1, 0]
```