Quick Sort

- Divide and Conquer
- O(nlogn) - Average case running time
- $O(n^2)$ - Worst case running time
- In-place
- Fastest

High-Level Description of Quick-Sort

The quick-sort algorithm sorts a sequence S using a simple recursive approach. The main idea is to apply the divide-and-conquer technique, whereby we divide S into subsequences, recur to sort each subsequence, and then combine the sorted subsequences by a simple concatenation. In particular, the quick-sort algorithm consists of the following three steps (see Figure 12.8):

1. ***Divide:*** If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element x from S, which is called the ***pivot***. As is common practice, choose the pivot x to be the last element in S. Remove all the elements from S and put them into three sequences:
   - L, storing the elements in S less than x
   - E, storing the elements in S equal to x
   - G, storing the elements in S greater than x

   Of course, if the elements of S are distinct, then E holds just one element—the pivot itself.
2. ***Conquer:*** Recursively sort sequences L and G.
3. ***Combine:*** Put back the elements into S in order by first inserting the elements of L, then those of E, and finally those of G.

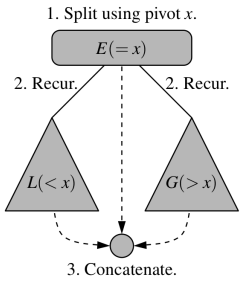
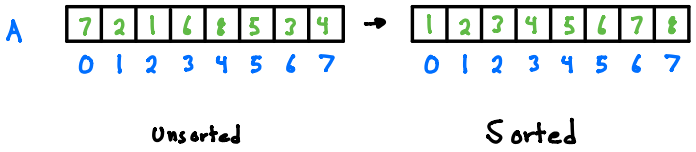
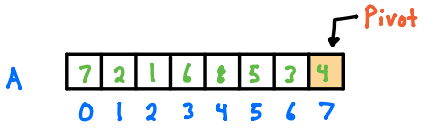


Figure 12.8: A visual schematic of the quick-sort algorithm.

Quick Sort

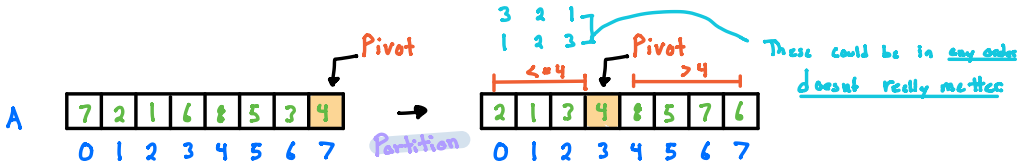


Quick Sort Overview

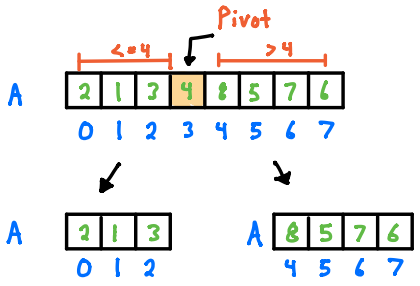


1) We first select one elements in the list (can be any element), we call this selection number, **Pivot.**

2) Now we **rearrange the list** such that **all the elements lesser than** the Pivot are towards the left of it and all the **elements greater** are towards the right of it (**Partitioning of a list**)



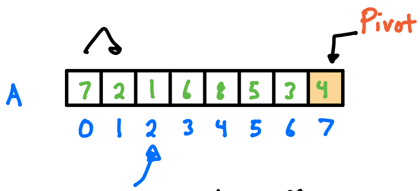
- We can break this problem into **two sub-problems**, those being sorting the array to the **Left of the pivot** and **sorting the array to the right of the pivot**



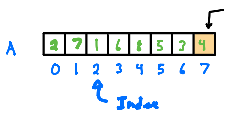
Different than Merge Sort

- Unlike merge sort we do **not need to create auxilery arrays** entirely, We can work on the **same array**. We just have to **keep track** of the **start and end index** of the **segments**

- helper data structure (taking up extra space)



p-index = ~~0~~ = 1



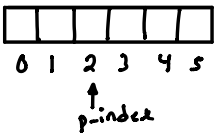

```
for i in range (0, 6)
    if A[i] <= pivot = 4
        [p-index] = A[i]
        A[i] = A[i-1]
        p-index = 1
```

QuickSort Function Sudo Code

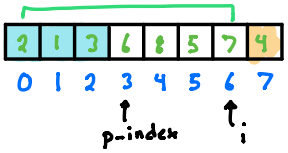
```
def QuickSort( A, start, end):                    # Index

    if start_index < end_index:                   # exit condition of recursive function
        return
    pIndex = partition (A, start, end)

Recursive Calls:
    QuickSort ( A, start, pIndex-1)

    QuickSort ( A, pIndex +1, end)
```

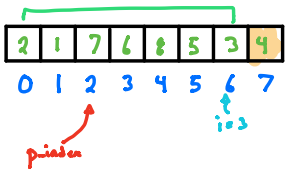
~ i in range (start_index, end_index-1)

```
def Partition (A, start, End):

    pivot = A[end]                                # ~ Selecting the Pivot

    Partition_index = start                       # ~ index
    for i in range ( start, end-1):

1st Swap →
        if A[i] <= pivot:
            A[i] = A[ p_index]
            A[i-1] = A[i]

            p_index = p_index+1

                                                  # ~ Swap
    A[p_index], A[end] = A[end], A[p_index]
    Return Partition_Index
```

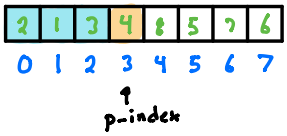


Note: At all times, values to the Left of p_index will be less than the pivot.





- Once we've gone through the range of the array, we will swap at the p_index with our End element, which is our Pivot



Start

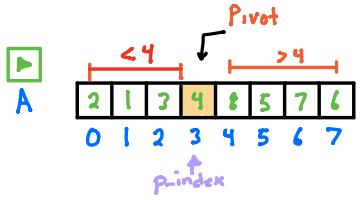
Starting Index Ending Index

① QS(A, 0, 7) A

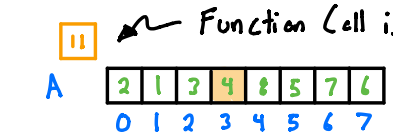
7	2	1	6	8	5	3	4
0	1	2	3	4	5	6	7

← Unsorted Array

① Since this satisfies our condition Start < end (0<7), we will call the Partition function, which will find our Pivot and swap values Left or Right depending on if they're higher or lower then the Pivot point it then will return a p_index, which will be index 3, for this step

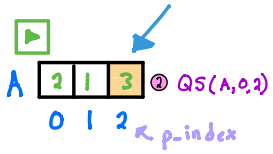


← We have 4 as our Pivot
- The Partition function will rearrange the array to this, it will return the p_index

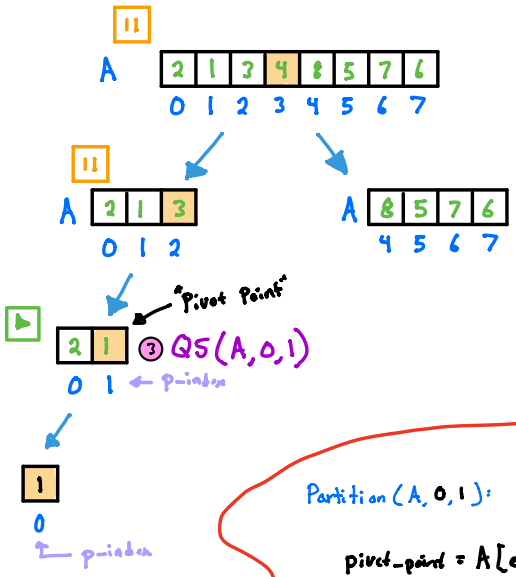


← Function Call is paused.
- Machine says it will come back once im finished

② With our p_index returned we will call the QuickSort(A, Start, pIndex-1), which will move our "End" index (3-1) = 2., With our starting being 0, it will still satisfy our Condition Start < End (0<2), so we call the Partition function it takes in our array, and index values 0 to 2, finds a Pivot point of 1 and reverse things, and return p_index = 2



~ All Values to the Left of the Pivot Point (3), were already less than 3.



③ After the p-index is returned (QuickSort(A, Start, pIndex-1)), is called. Our p-index goes (2-1) = 1. (QS(A,0,1)), This still satisfies the condition Start < End (1<0). So We call the Partition function again.

Partition(A, 0, 1):

pivot_point = A[end] → 1

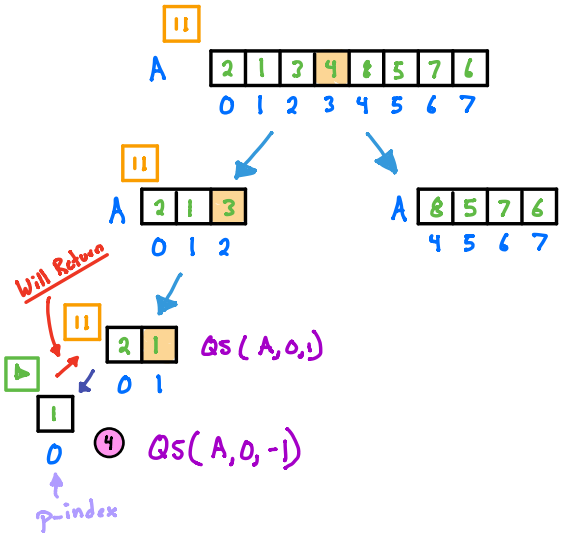
p-index = 0

Index 0

A[0], A[end] = A[end], [p_index]
1 0

So 2 is now in index ①

```
def Partition(A, Start, End):
    pivot = A[end]          ~ Selecting the Pivot
    Partition_index = start  ← index
    for i in range(start, end-1):
        if A[i] <= pivot:
            A[i] = A[p_index]
            A[i-1] = A[i]
            p_index = p_index+1
                                ~ Swap
    A[P_index], A[end] = A[end], A[p_index]
    Return Partition_Index
```



4 We we will call the QuickSort(A, Start, pIndex-1), with QS(A,0,0-1) • QS(A,0,-1), our condition Start > End is false, so we will Return, and Control will return back to QS(A,0,1).

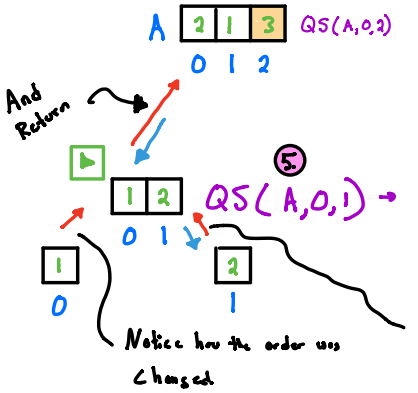
```
def QuickSort(A, start, end):
    if start index < end index :
        return
    pIndex = partition(A, start, end)
Recursion calls
    QuickSort(A, start, pIndex-1)
    QuickSort(A, pIndex+1, end)
```

Index

exit condition of recursive function

QuickSort(A, 1, 1)

Returned Array

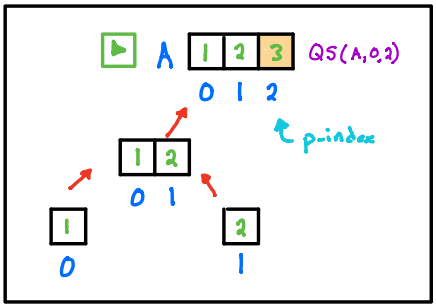


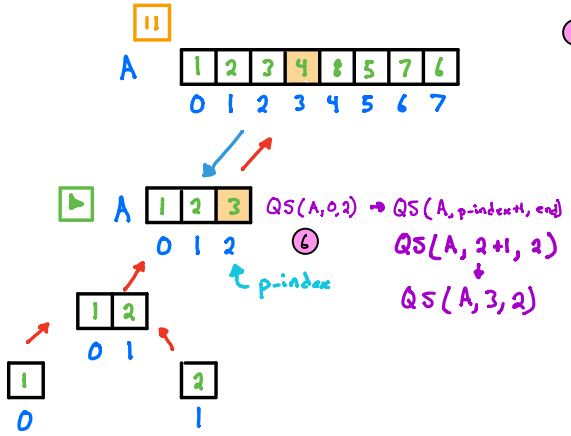
5 With the Control back, QuickSort(A, pIndex+1, end), is called, but since start < end is false, it will Return and return Control to QS(A,0,2).

QuickSort(A, pIndex+1, end)
QS(A, 0+1, 1)
QS(A, 1, 1)

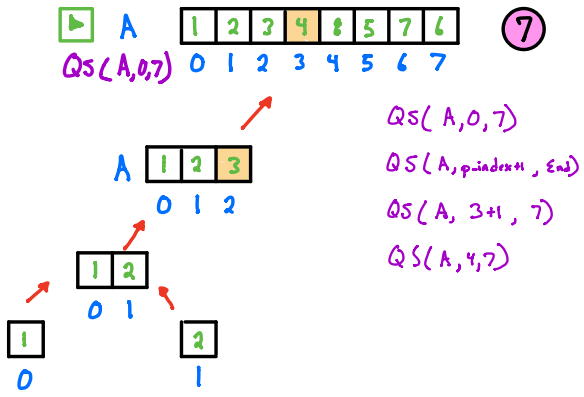
Will not partition and simply return.

The index order for 1 and 2 were swapped earlier

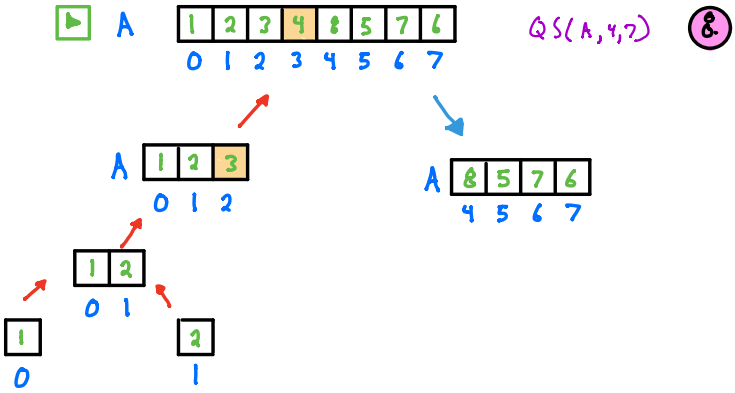




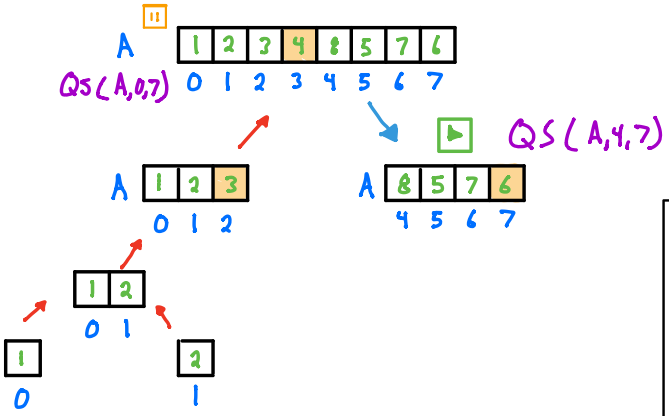
⑥ With Control back here, QuickSort(A, pIndex+1, end) is called and QS(A,3,2) is invalid, so QS will return (↗)



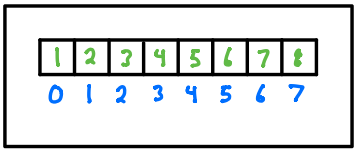
⑦ Our Original array at the top will now resume. QS(A,0,7). Now it will make a call right of the pivot, QuickSort(A, pIndex+1, end), which will be QS(A,4,7).



And here are Starting Index Value is < end index, so We can now partition the other side of the array!



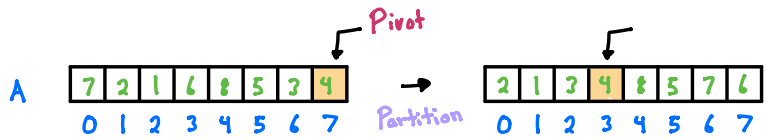

```
def Partition(A, Start, End):
    pivot = A[end]                  ~ Selecting the Pivot
    Partition_index = start         ~ index
    for i in range(start,end-1):
        if A[i] <= pivot:
            A[i] = A[p_index]
            A[i-1] = A[i]
            p_index = p_index+1
                                    ~ Swap
    A[p_index], A[end] = A[end], A[p_index]
    Return Partition_Index
```



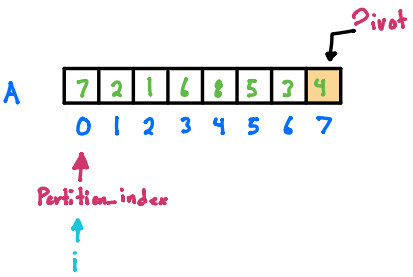
Sorted

```
def QuickSort(A, start, end):

    if start > end:
        return

    p_index = Partition(A, start, end)

    # recursive Calls here:

    QuickSort(A, start, p_index-1)

    QuickSort(A,p_index +1, end)
```

Partition Logic



- This what we want, to select a pivot and to rearrange a list such that all the elements lesser than the pivot are to the Left and all the elements greater than the pivot are to the right of it.




```
def Partition(A ,start, end):

    pivot = A[end]

    p_index = start

    for i in range(start,end):

        if A[i] <= pivot:

            #swap number at index with the P_index!
            A[p_index], A[i] = A[i], A[p_index]

            p_index = p_index +1

    # this is swaping the pivot point between the sorted values

    A[p_index], A[end] = A[end], A[p_index]

    return p_index
```

Running Time of Quicktime

Actual Code

```
def Partition(A ,start, end):

    pivot = A[end]

    p_index = start


    for i in range(start,end):
        print('Index:', i)

        if A[i] <= pivot:


            #swap number at index with the P_index!
            A[p_index], A[i] = A[i], A[p_index]

            p_index = p_index +1


    # this is swaping the pivot point between the sorted values

    A[p_index], A[end] = A[end], A[p_index]


    return p_index


def QuickSort(A, start, end):


    if start > end:
        return

    p_index = Partition(A, start, end)

    # recursive Calls here:

    QuickSort(A, start, p_index-1)

    QuickSort(A,p_index +1, end)


array = [7,2,1,6,8,5,3,4]

n = len(array)

QuickSort(array,0,n-1)

print(array)
```