</talentlabs>

Lecture 6
Sorting Algorithms - Part 2

</talentlabs>

Agenga

- Recursion Revision
- Merge Sort
- Quick Sort

Recursion Review

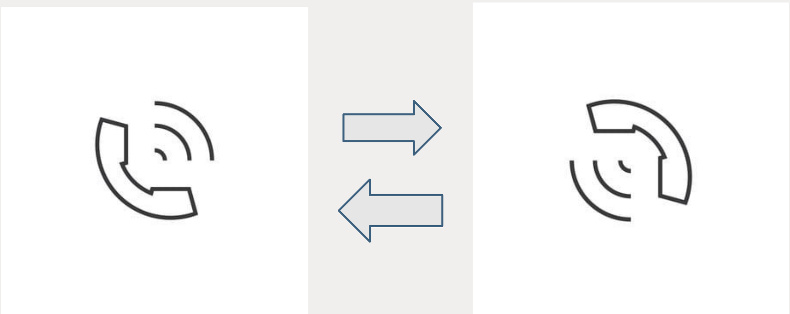
</talentlabs>

What is Recursive algorithm?

The process in which a function calls itself directly or indirectly is called recursion.

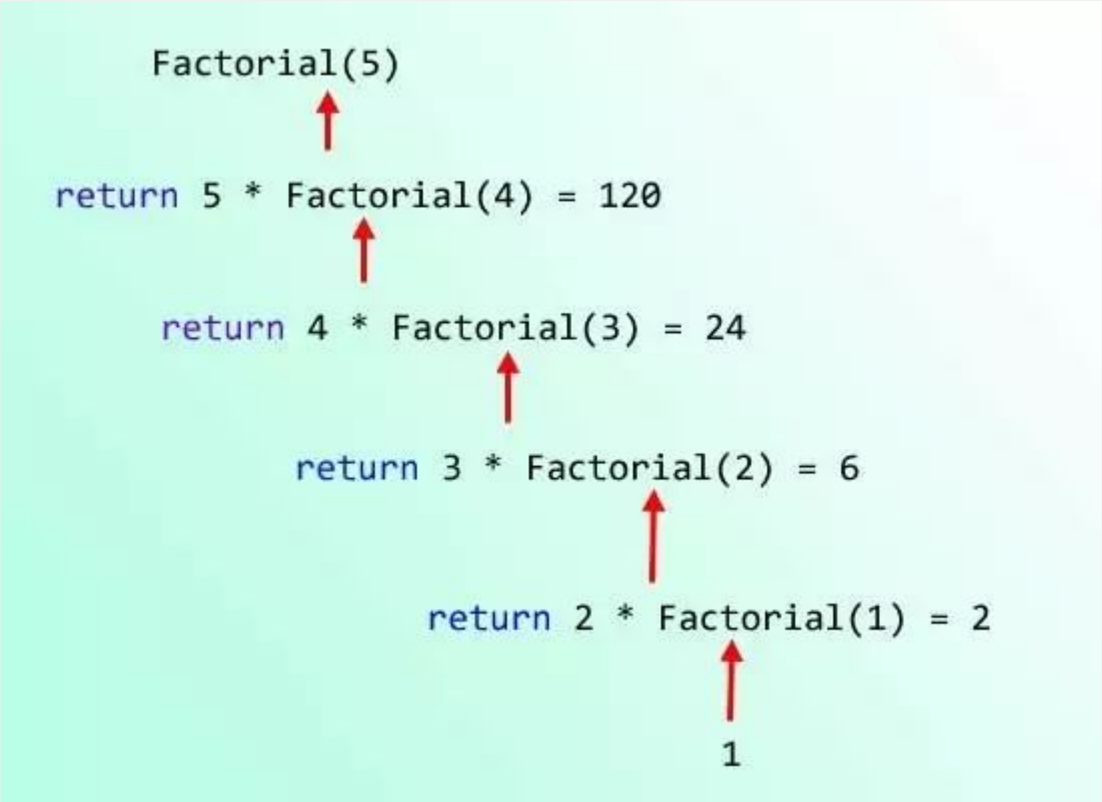
Recursion is useful for problems that can be represented by a simpler version of the same problem.

The smallest example of the same task has a non-recursive solution.



JavaScript Recursive Function Example

```
function factorial(n){
    if(n == 0 || n == 1){
        return 1;
    }else{
        return n * factorial(n-1);
    }
}
```

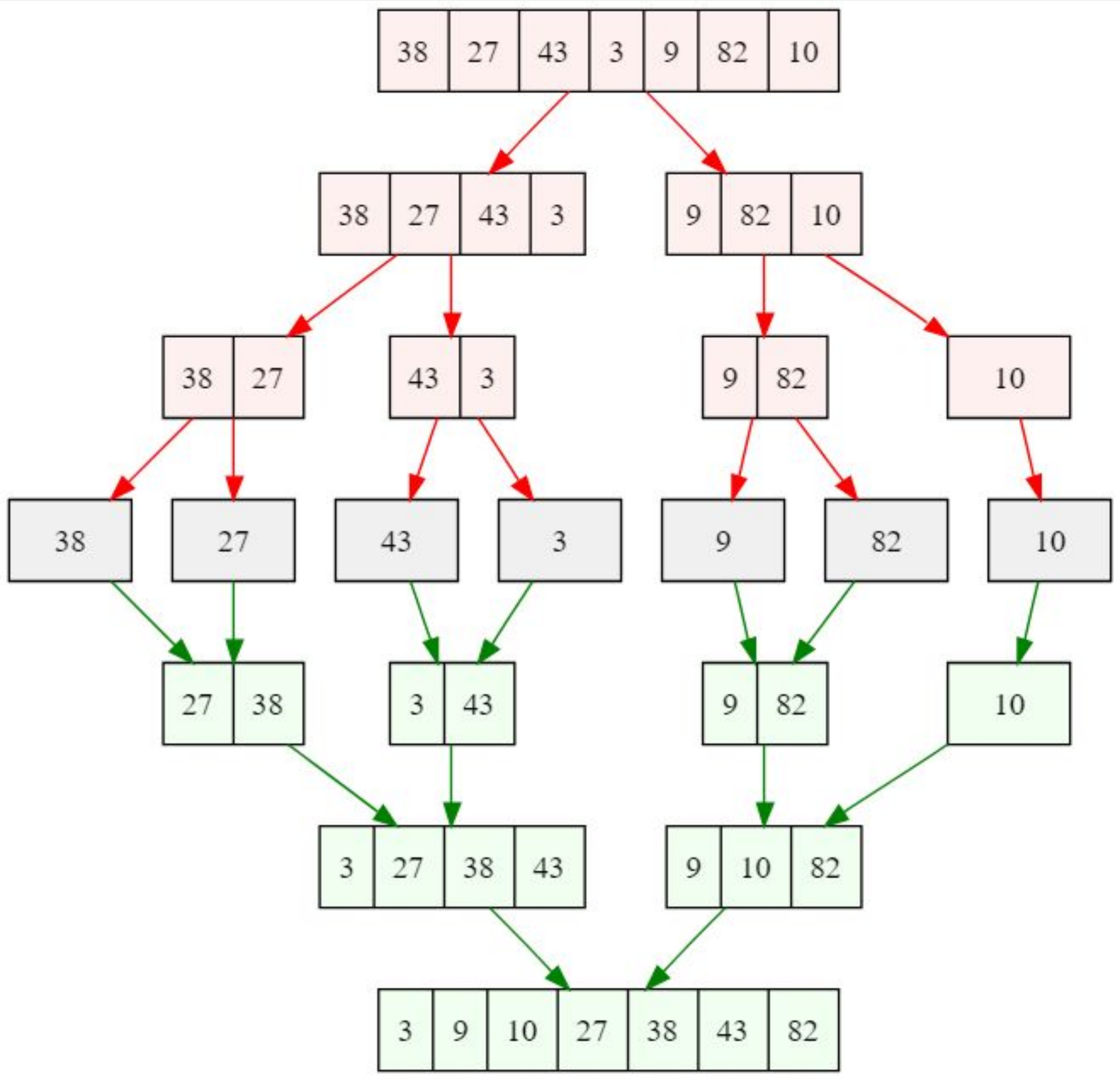


Sorting Algorithm 3 - Merge Sort

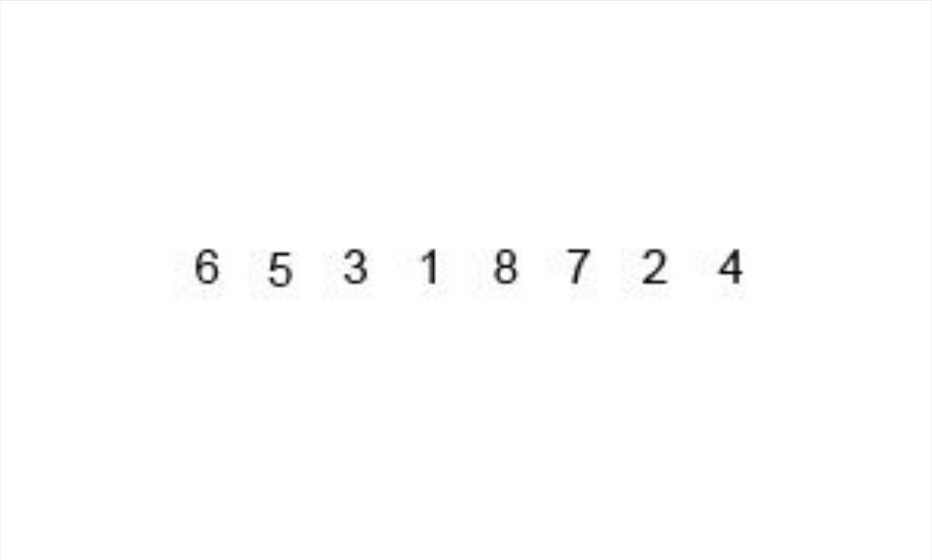
Merge Sort

- Merge Sort divides the input array into two halves, calls itself(the merge sort function) for the two halves, and then merges the two sorted halves.

- This is often referred as “Divide and Conquer” - Break down the problem to smaller problem


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort



Attribution: Swfung8, CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0>, via Wikimedia Commons, https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge-sort-example-300px.gif

Key Study Notes:

1. The exit condition of the algorithm is “there is only 1 element in the half”
2. The most difficult part of this algorithm is not about “divide” but to “merge”
3. When we merge, we always assume that both halves is already sorted, so we only need to compare the first element and see which one is smaller

Merge Sort

Consider using Merge sort to sort this array in increasing order.

6	5	12	10	9	1

Merge Sort

Merge sort first divides the whole array iteratively into equal halves. An array of 6 items is divided into two arrays of size 3.

6	5	12

10	9	1

Merge Sort

We continue to divide these two arrays into halves.

6	5	12

10	9	1

6

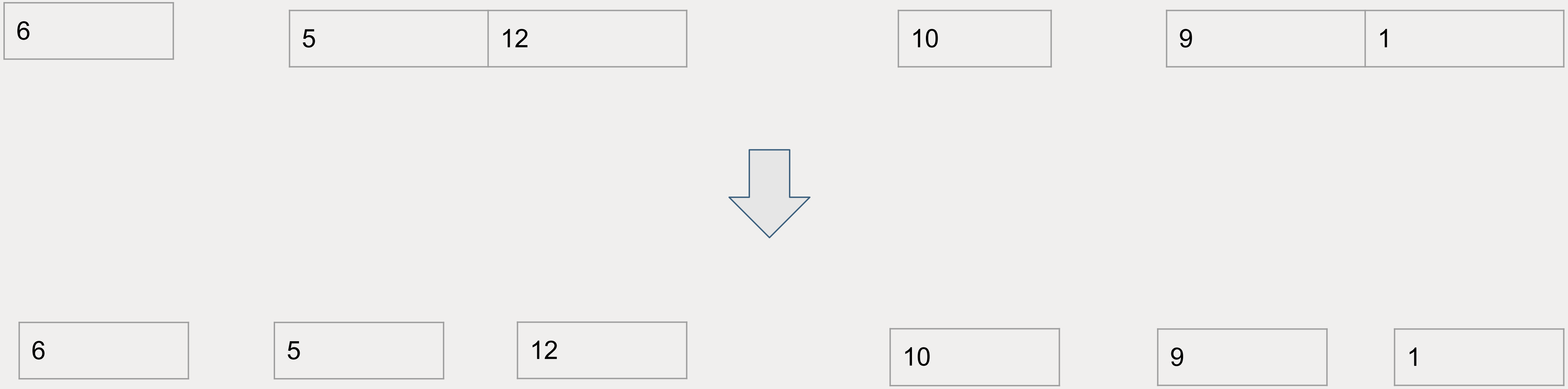
5	12

10

9	1

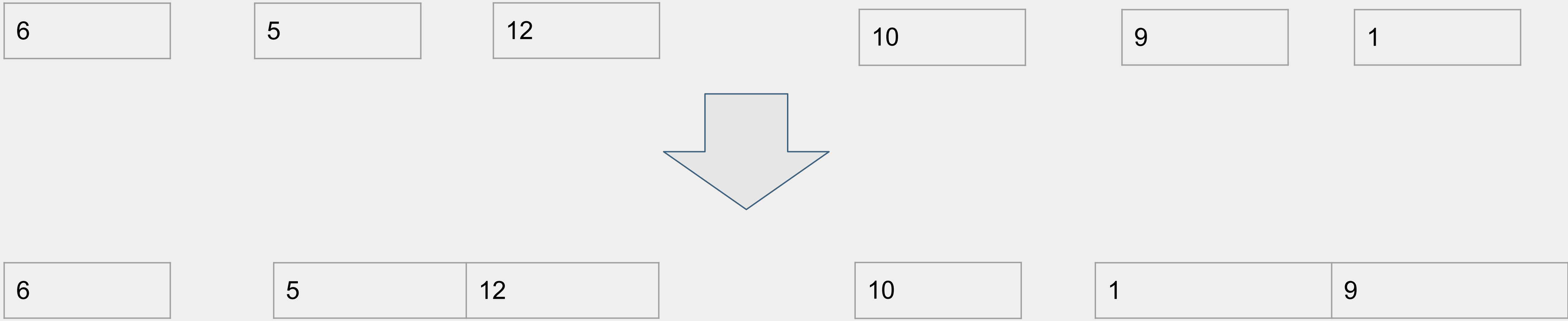
Merge Sort

We further divide these arrays and we achieve atomic value which can no more be divided.



Merge Sort

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 5 and 12 are in sorted positions. We compare 1 and 9 and in the target list of 2 values we put 1 first, followed by 9.



Merge Sort

We compare lists of three data values, and merge them into a list of found data values placing all in a sorted order.

5	6	12

1	9	10

Merge Sort

After the final merging, the list should look like this

1	5	6	9	10	12

Merge Sort Algorithm

For each function call (remember recursion?)

Step 1 − if it is only one element in the input list, then it is already sorted, return.

Step 2 − if it is >1 element, break down the list into 2 halves, and call the mergeSort() with each half

Step 3 − merge the two smaller sorted lists into new list in sorted order.

 </talentlabs>

Merge Sort Algorithm

Main Function

```
function mergeSort(inputList){
    if (inputList.length === 1){
        return inputList
    }
    else {
        const half = inputList.length / 2
        const left =  inputList.splice(0, half)
        const sortedLeft = mergeSort(left)
        const sortedRight = mergeSort(inputList)
        return merge(sortedLeft, sortedRight)
    }
}
```

If the input list is of size 1, then no need sort, just return it

Split the array into 2 halves, using “splice” function

Sort each halves using the “mergeSort()” function

Combine the sorted left and sorted right

Merge Sort Algorithm

Merge Function

```
function merge(leftList, rightList){
    let arr = []
    while(leftList.length > 0 && rightList.length > 0){
        if (leftList[0] < rightList[0]){
            arr.push(leftList.shift())
        }
        else {
            arr.push(rightList.shift())
        }
    }

    return arr.concat(leftList).concat(rightList)
}
```

If both left and right list still have at least 1 element

Pick the smaller first element and push to the result array

Combine the result list with left and right list. This is because as it could be only one list was emptied. (When would this happen?)

Merge Function

```
function merge(leftList, rightList){
    let arr = []
    while(leftList.length > 0 && rightList.length > 0){
        if (leftList[0] < rightList[0]){
            arr.push(leftList.shift())
        }
        else {
            arr.push(rightList.shift())
        }
    }

    return arr.concat(leftList).concat(rightList)
}
```

Sorting Algorithm 4 - Quick Sort

Quick Sort

- Quick Sort picks an element as “pivot” and partitions the given array around the picked “pivot”.

- After “Partition”, all smaller elements (smaller than “pivot”) should be placed before “pivot”, and put all greater elements (greater than “pivot”) should be placed after “pivot”.

- After “Partition”, we will call “quickSort()” method for both the Right List and the Left List to sort each half

- Quick Sort is also a “divide and conquer” algorithm

1	17	3	10	9	6

Pivot

- We need to pick a pivot in each “quick sort” call
- This would impact the efficiency of the sorting algorithm, but as a beginner, we will just always use the right most element as the “pivot”
- There are some other ways of picking pivot, e.g. picking middle element, or the leftmost element, or picking the median element

Partition

- Partition is the most important step in quick sort, the objective of “partition” is to put the numbers smaller than pivot on the left, and those larger than pivot on the right
- We are going to compare each element with the pivot
  - If the element is smaller than the pivot, then we swap it with “the first element that is larger than pivot”
  - If the element is larger than the pivot, then we just leave it at the original position

Partition Process Demonstration

i: track the position where should the next “small” elements be placed
j: track which element we are comparing with “pivot”

At the very beginning, i = -1, j = 0
that means if element 0 is smaller than pivot, then we will i++, and we will swap it with element i (after i++), aka element 0
aka, no need swap => because there are no element larger than pivot on the left

1	17	3	10	9	6

Partition Process Demonstration

i: track the position where should the next “small” elements be placed
j: track which element we are comparing with “pivot”

now, i = 0, j = 1
that means if element 1 is smaller than pivot, then we will swap it with element i+1, aka element 1
aka, no need swap => because there are no element larger than pivot on the left

However, element 1 is larger than the pivot, so no need to bump up i or do swapping

1	17	3	10	9	6

Partition Process Demonstration

i: track the position where should the next “small” elements be placed
j: track which element we are comparing with “pivot”

now, i = 0, j = 2
that means if element 2 is smaller than pivot, then we will i++, and swap element 2 with i (after i++), aka element 1

We need to do swap this time, because 3 is smaller than 6, and there is a element larger than pivot on the left (element 1)

1	17	3	10	9	6

</talentlabs>

Summarizing Partition

- For each of the element
  - If the element is smaller than the pivot, then we swap it with “the first element that is larger than pivot”
  - If there are no “large element” identified yet (“large element < j”), then no need to swap (or swap with itself)
  - If the element is larger than the pivot, then we just leave it at the original position

Summarizing Partition

```
const partition = (arr, minIndex, maxIndex) => {
    // Assuming the pivot is always the rightmost element
    pivot = arr[maxIndex]

    // start with i - 1
    // (as we haven't found any large elements yet)
    i = minIndex - 1

    // Find the right position of pivot
    for (let j = minIndex; j <= maxIndex - 1; j++){
        if (arr[j] < pivot){
            i++
            // swap
            tempI = arr[i]
            arr[i] = arr[j]
            arr[j] = tempI
        }
    }

    tempIPlus1 = arr[i+1]
    arr[i+1] = pivot
    arr[maxIndex] = tempIPlus1

    return i+1
}
```

The range that we are going to do partition, i.e. defining the "sub array" using min and max index

Our pivot is always the rightmost element

We start from i = -1 (or min index -1)

If the element j is smaller than pivot, then we will do i++ and swap element i with element j

After finished all the comparison, we are going to put the pivot at the right position, which is position i+1

1	17	3	10	9	6

Quick Sort

```
const quickSort = (arr, minIndex, maxIndex) => {
    if (maxIndex > minIndex){
        pi = partition(arr, minIndex, maxIndex)
        quickSort(arr, minIndex, pi-1)
        quickSort(arr, pi + 1, maxIndex)
    }
}
```

For the range, we will do the partition

- First cycle, range would be from 0 to end

Partition function should return the position of pivot after partition

Sort the left and right hand side using quick sort

1	17	3	10	9	6

```
const quickSort = (arr, minIndex, maxIndex) => {
    if (maxIndex > minIndex){
        pi = partition(arr, minIndex, maxIndex)
        quickSort(arr, minIndex, pi-1)
        quickSort(arr, pi + 1, maxIndex)
    }
}
```