**Week 6: Lab pre-read**

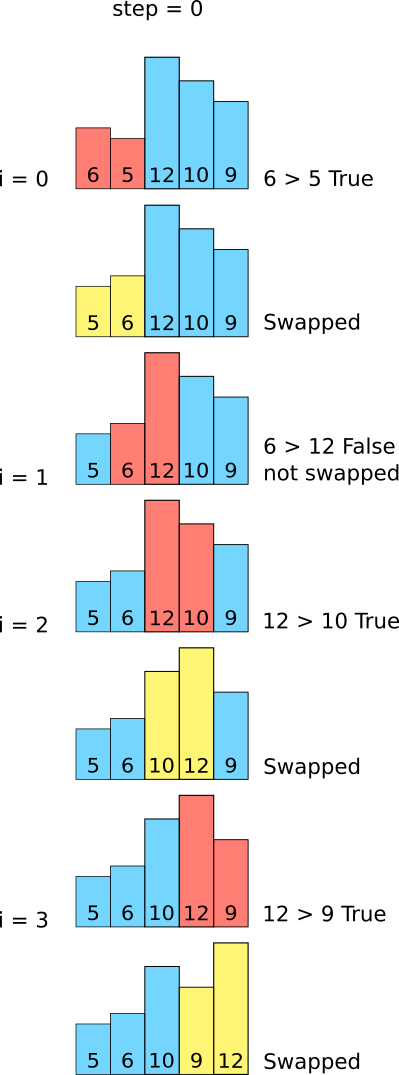
# **1. Bubble sort**

The idea of the bubble sort algorithm is to move higher valued elements generally towards the right, and lower valued elements generally towards the left.

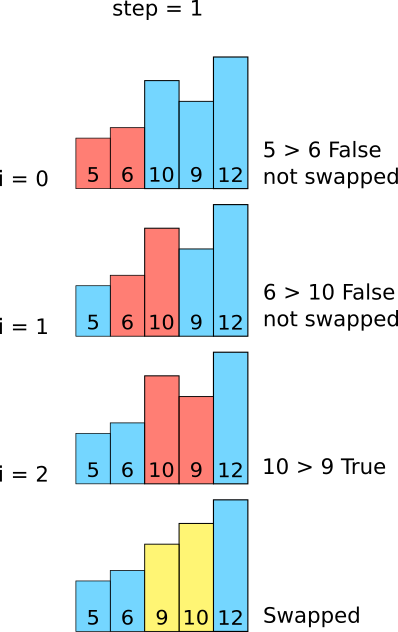
Let's implement bubble sort using the below data.

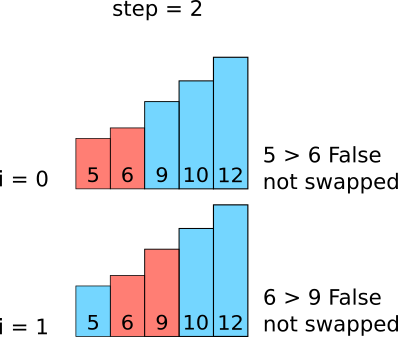
**bubbleSort([6, 5, 12, 10, 9, 1])**

1. **First Iteration (Compare and Swap)**
   * Starting from the first index, compare the first and the second elements.
   * If the first element is greater than the second element, swap the elements.
   * Now, compare the second and the third elements. Swap them if they are not in order as mentioned above.
   * The process goes on until the last element, at which point the heaviest/largest element in the array would be sedimented at the end of the array.

****

1. **Subsequent iterations**
   * The same process goes on for the remaining iterations.
   * After each iteration the largest element among the unsorted elements is placed at the end.





**Analysis**

* Notably, each iteration processes one less element (the one deposited at the end of previous iteration is ignored since it is already in the correct place - the largest at the end) which makes the number of elements processed as:

O(n) + O(n-1) + O(n-2) +. ..... + O(2) ~ O(n2).

# 

# **2. Merge sort**

In **Merge sort**, the idea of the algorithm is to sort smaller arrays and then combine those arrays together (merge them) in sorted order.

Merge Sort is based on the principle of Divide and Conquer Algorithm where an array is divided into multiple sub-arrays and each sub-array is sorted individually. Finally, sub-arrays are combined to form the final solution.

**In pseudocode:** Using the Divide and Conquer technique, we divide the array into sub-arrays. After splitting the arrays,

* Sort the left half of the array (assuming n > 1)
* Sort the right half of the array(assuming n> 1)

Then merge the two halves together

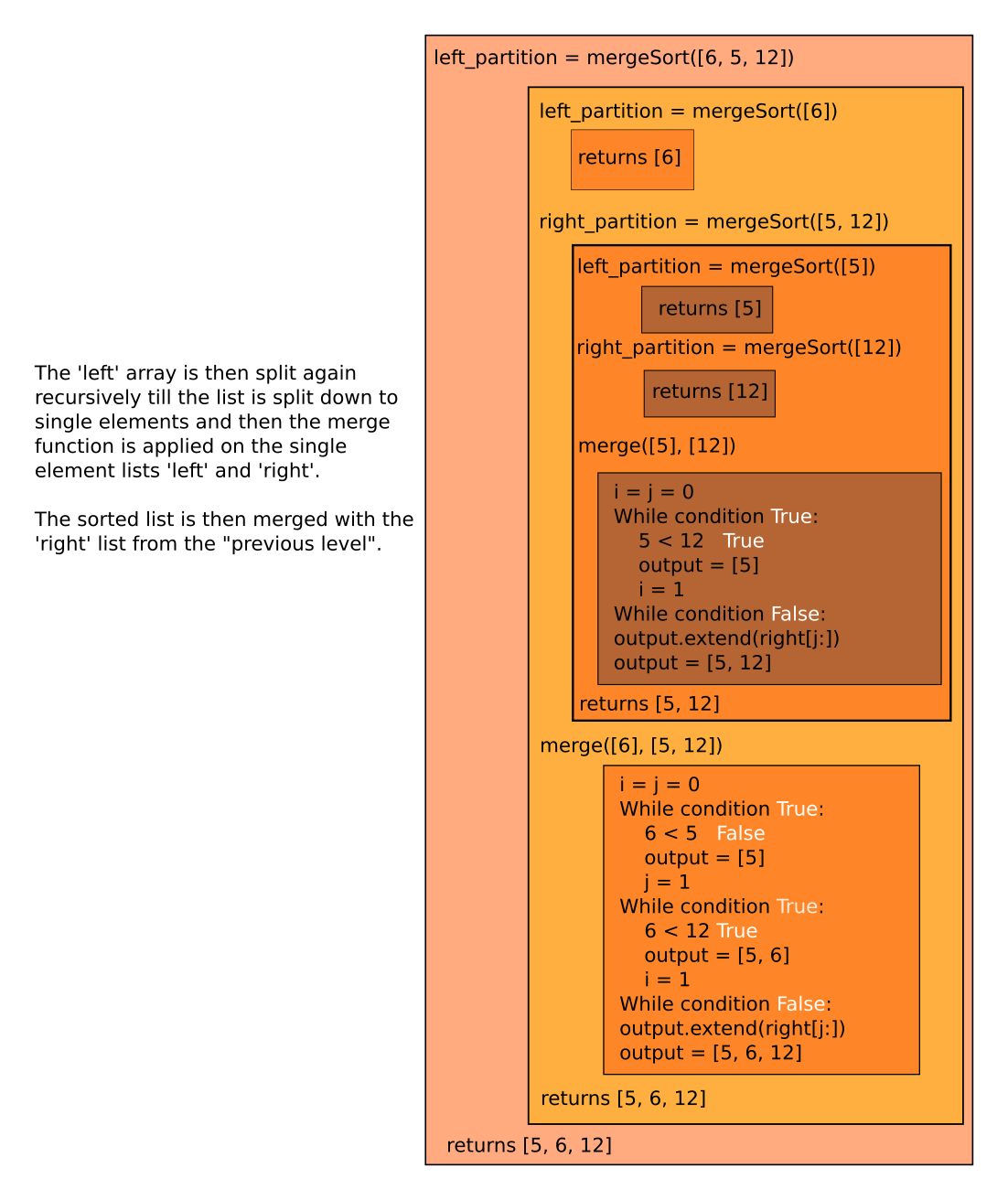
Notably, at each level of the binary tree, merge sort needs to process O(n) elements. Since there are log(n) levels in a binary tree of n elements, the total work done in merge sort ~ O(nlog(n)).

**Working example:**

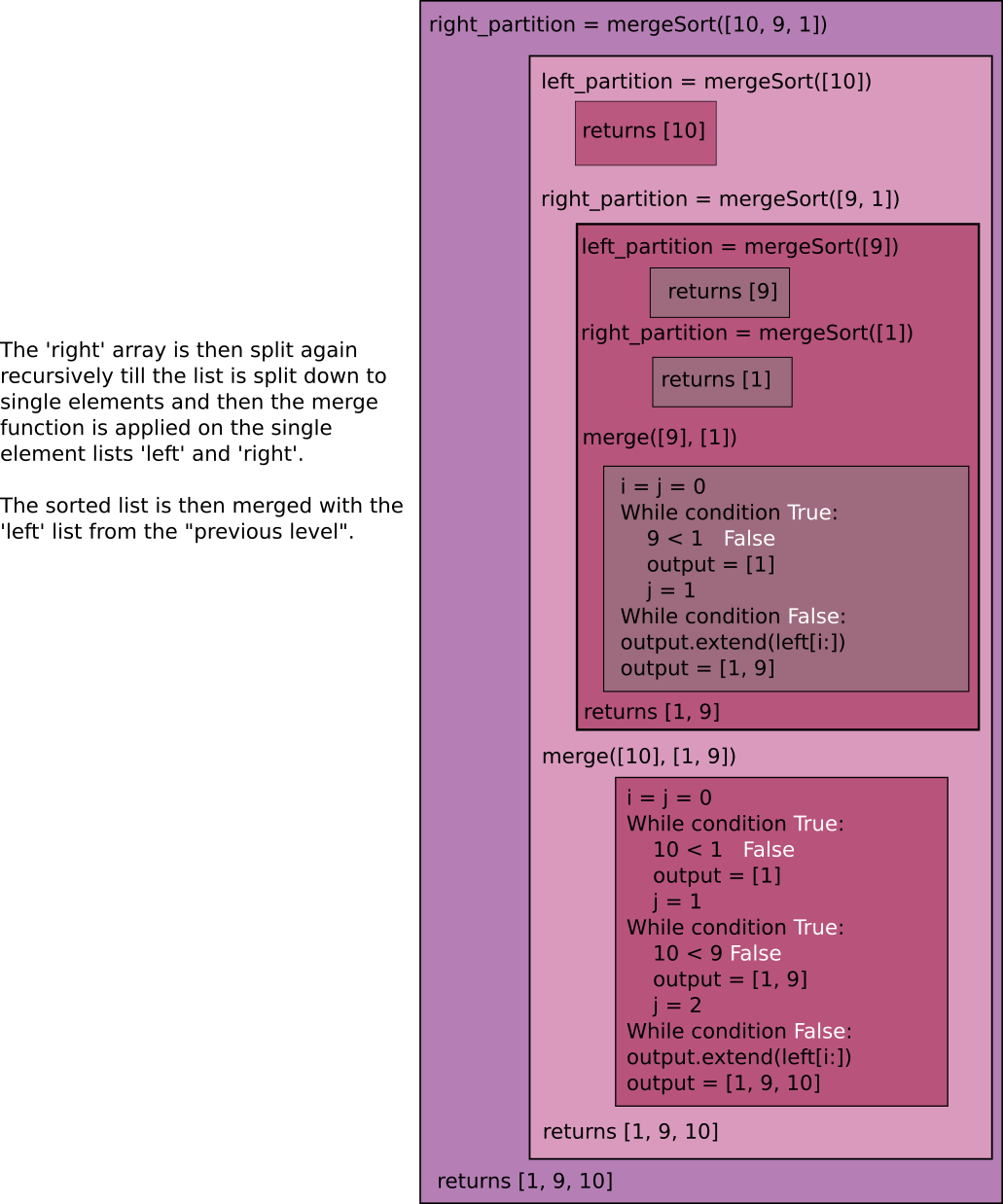
1. **mergeSort([6, 5, 12, 10, 9, 1])**

Splits the original array into two halves 'left' and 'right' which are recursively split again by using the mergeSort function until they are down to single elements.

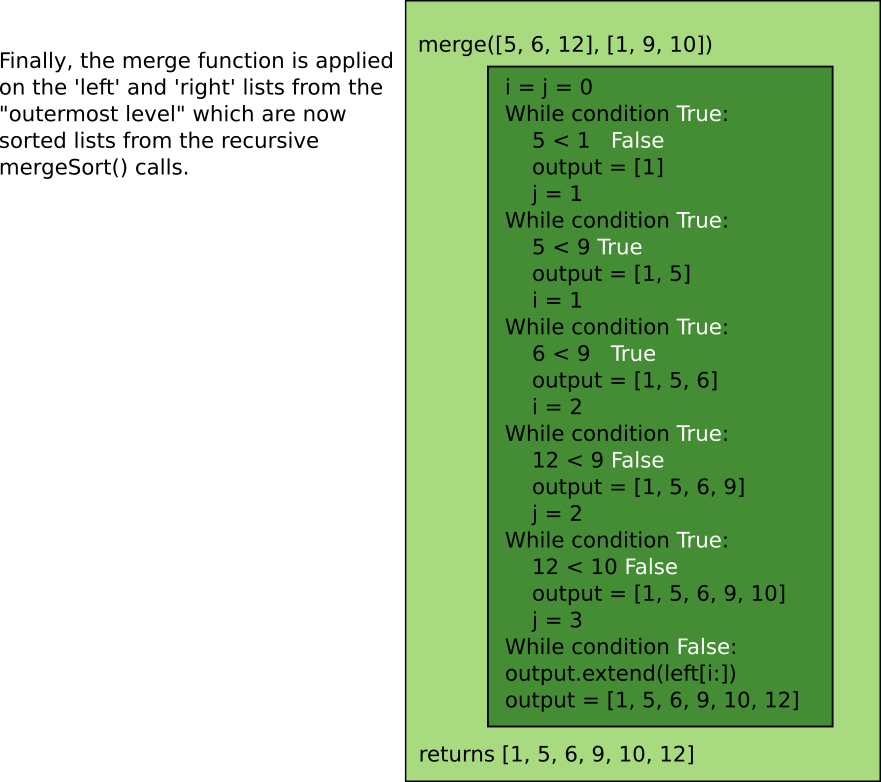
1. **The mergeSort function is recursively called on the ‘left’ list again:**

****

1. **The mergeSort function is recursively called on the ‘right’ list again:**

****

1. **The merge function is called on the outputs:**

****

The full svg image can be found [here](https://drive.google.com/file/d/1DbKVSwPlfed3lQ3tqtozItKrKBinDoD8/view?usp=sharing).

# 

# 

# 

# 

# **3. Quicksort**

The name **Quicksort** stems from the fact that it can sort a list of data elements substantially faster than any other sorting method. It is one of the most efficient sorting algorithms and is based on dividing an array into smaller ones and swapping based on a comparison with the selected ‘pivot’ element.

Partitioning is the most important process in quickSort (). Given an array and a pivot element ‘p’, the goal of partitioning is to place ‘p’ in the right position of the sorted array, with all smaller items (less than p) placed before p and all larger elements (greater than p) placed after p. This partitioning step is repeated over and over again until all elements are in their right places, which corresponds to the array being sorted.

**In pseudocode:** Quicksort follows the below steps

**Step 1** − Selecting a pivot element. (There are variations to selecting the pivot element. For this exercise we will be selecting the last element as the pivot element)

**Step 2** − Calling for partitioning of the array on the basis of pivot.

After selecting a pivot element, re-arrange all the elements in such a way that elements lesser than pivot are on the left side of the pivot and higher valued elements are on the right side of the pivot.

To do this re-arrangement of the list, we use two pointers(1st pointer initially positioned at 1st index and 2nd pointer initially positioned at (last index -1)) for iterating over the elements.

* Pointers are made to traverse towards each other, by incrementing the first pointer index if the element at the first pointer index is lesser or equal to the element at the pivot index and by decrementing the second pointer index if the element at the second pointer index is greater or equal to the element at the pivot index.
* When the 1st pointer encounters value greater than pivot and 2nd pointer encounters value lesser than pivot, the value at the 1st pointer is swapped with the value of the second pointer and vice versa.
* When the 1st pointer and 2nd pointer index cross each other i.e i > j, stop traversing. Swap the elements at the pivot variable and the 1st pointer index(i).

**Step 3** − After re-arrangement, split the list is divided into two parts:

* Sublist before pivot element
* Sublist after pivot element

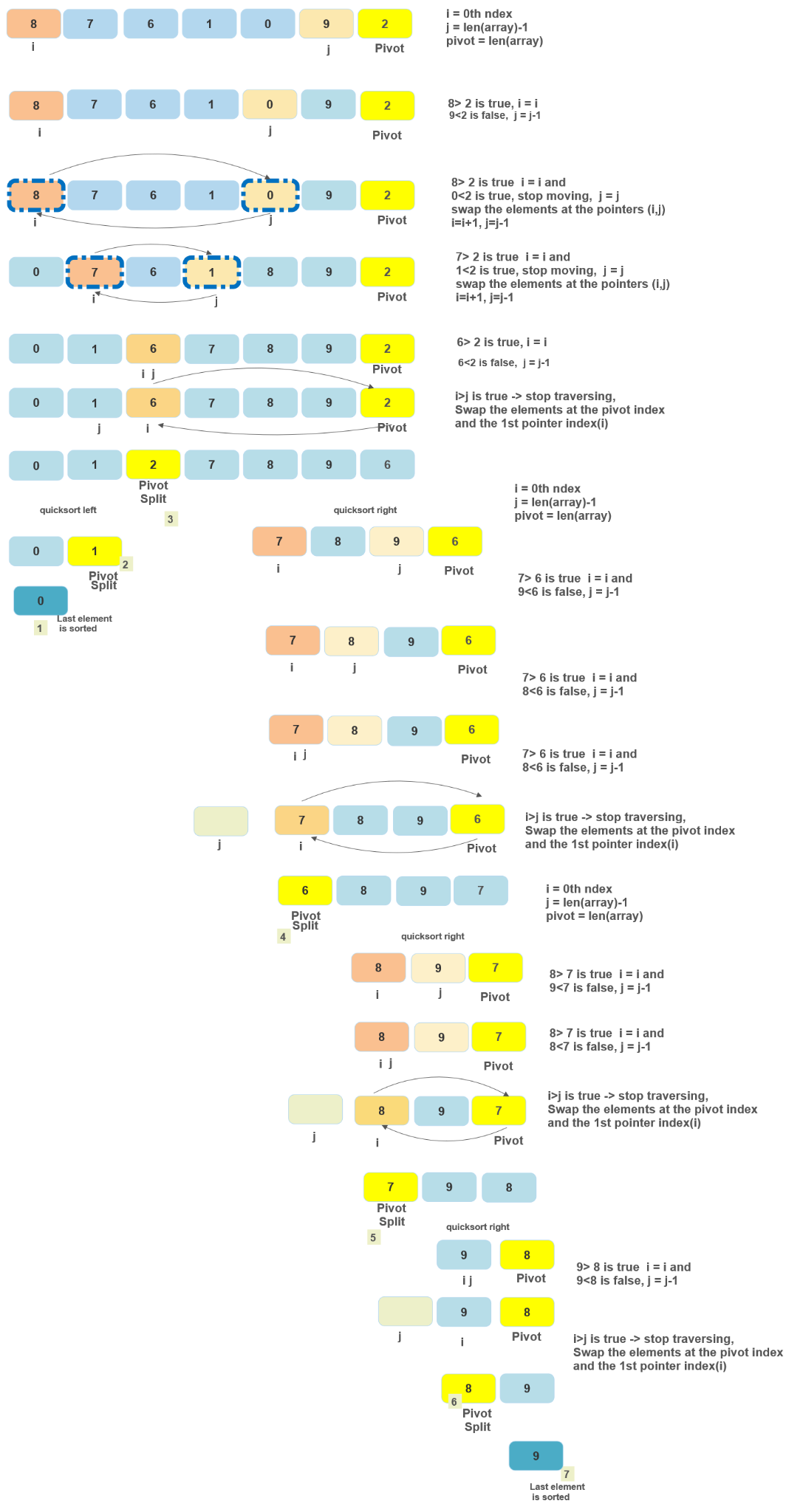
**Step 4** − Apply quicksort on the left partition (sublist before the pivot element) by recursively partitioning until the number of elements in the sub-partition is less than or equal 1.

**Step 5** − Apply quick sort on the right partition (sublist after the pivot element) recursively partitioning until the number of elements in the sub-partition is less than or equal to 1.

The resulting array, after all the partitioning steps have placed their corresponding pivots into their right places, thus becomes sorted.

**Working example:**

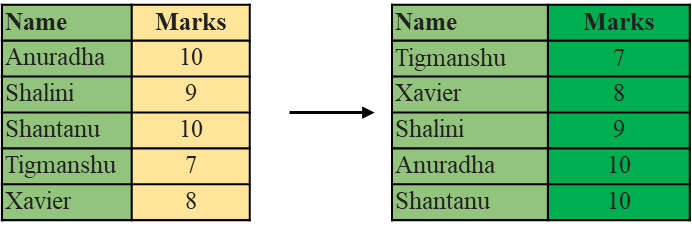
Example of quick sort with data = [8, 7, 6, 1, 0, 9, 2]



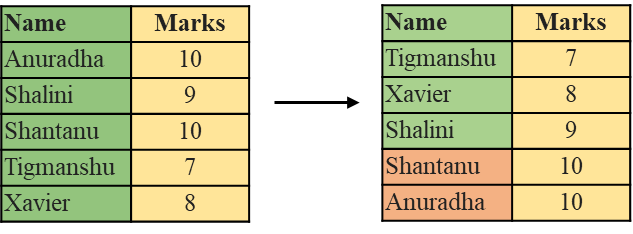
# **4. Stability of a sorting algorithm**

# A sorting algorithm is said to be stable if it preserves the relative ordering of equal value elements, as they appear in the unsorted input array, at output of the sorting process.

# For example, let’s say the table below is first sorted in the alphabetical order of names (column 1). We would then like to sort with respect to the marks (column 2). If we apply a sorting algorithm on column 2, a stable sorting algorithm would be the one that maintains the two ‘10’s in the same order i.e. Anuradha with marks 10 would appear before Shantanu with the same marks.



From amongst the sorting algorithms we have seen, **bubble sort and merge sort** produce the sorted result displayed above. Thus, they are **stable sorting algorithms.** However, if we apply quicksort on the values in column 2, we get the following result (Confirm this by running the quicksort implementation in the jupyter notebook on this list of marks.). Since the relative ordering of ‘Shantanu’ and ‘Anuradha’ is compromised, **quicksort** is an **unstable sorting algorithm.**



This results in the circular problem, where sorting by a certain column A and then by another column B disrupts the sorting done on column A and vice-versa.

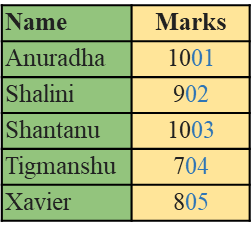
**Why does this happen?**

Bubble sort and Merge sort only ever swap adjacent elements, and in the case that one has a value that is different (smaller or larger) than the other. This bypasses the possibility of two equal value elements swapping positions.

On the other hand, quicksort is able to swap non-adjacent elements, when they have a difference of value in the desired direction. This allows ‘Tigmanshu - 7’ to be swapped with ‘Anuradha - 10’, causing the relative ordering of ‘Anuradha - 10’ and ‘Shantanu - 10’ to be disrupted, by putting the former after the latter.

**How can this be avoided?**

One way to make a sorting algorithm stable is by decorating the elements to be sorted in a way so as to break ties. In the example above, we may decorate the right column by concatenating the row indices to the values, as follows.



Note that this comes at an additional processing cost. The space complexity increases by 𝚯(n) - the space needed for the suffixes, while the sorting time can go up by twice as much.

# 

# 

# 

# 

# 

# 

# 

# **5. Algorithmic complexity**

Time complexity plot for merge vs. bubble vs. quick sort

