**Assignment-3**

1. **Bubble sort:**

The **bubble sort** makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item “bubbles” up to the location where it belongs.

Figure 1 shows the first pass of a bubble sort. The shaded items are being compared to see if they are out of order. If there are *n* items in the list, then there are 𝑛−1n−1 pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.



Figure 1: bubbleSort: The First Pass

At the start of the second pass, the largest value is now in place. There are 𝑛−1n−1 items left to sort, meaning that there will be 𝑛−2n−2 pairs. Since each pass places the next largest value in place, the total number of passes necessary will be 𝑛−1n−1. After completing the 𝑛−1n−1 passes, the smallest item must be in the correct position with no further processing required. ActiveCode 1 shows the complete bubbleSort function. It takes the list as a parameter, and modifies it by exchanging items as necessary.

The exchange operation, sometimes called a “swap,” is slightly different in Python than in most other programming languages. Typically, swapping two elements in a list requires a temporary storage location (an additional memory location). A code fragment such as

temp = alist[i]

alist[i] = alist[j]

alist[j] = temp

will exchange the ith and jth items in the list. Without the temporary storage, one of the values would be overwritten.

In Python, it is possible to perform simultaneous assignment. The statement a,b=b,a will result in two assignment statements being done at the same time (see Figure 2). Using simultaneous assignment, the exchange operation can be done in one statement.

Lines 5-7 in ActiveCode 1 perform the exchange of the 𝑖i and (𝑖+1)𝑡ℎ(i+1)th items using the three–step procedure described earlier. Note that we could also have used the simultaneous assignment to swap the items.

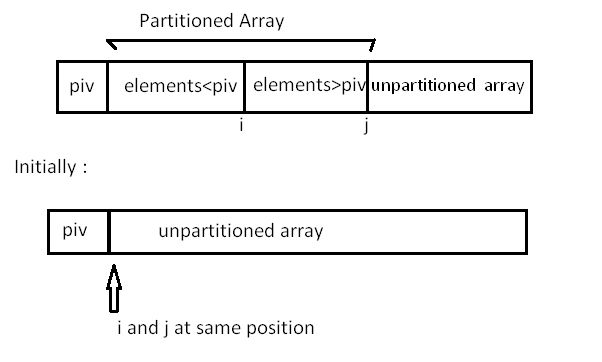
1. **Quick Sort:**

Quick sort is based on the divide-and-conquer approach based on the idea of choosing one element as a pivot element and partitioning the array around it such that: Left side of pivot contains all the elements that are less than the pivot element Right side contains all elements greater than the pivot

It reduces the space complexity and removes the use of the auxiliary array that is used in merge sort. Selecting a random pivot in an array results in an improved time complexity in most of the cases.

**Implementation** :

Select the first element of array as the pivot element First, we will see how the partition of the array takes place around the pivot.

   
In the implementation below, the following components have been used: Here, A[] = array whose elements are to be sorted

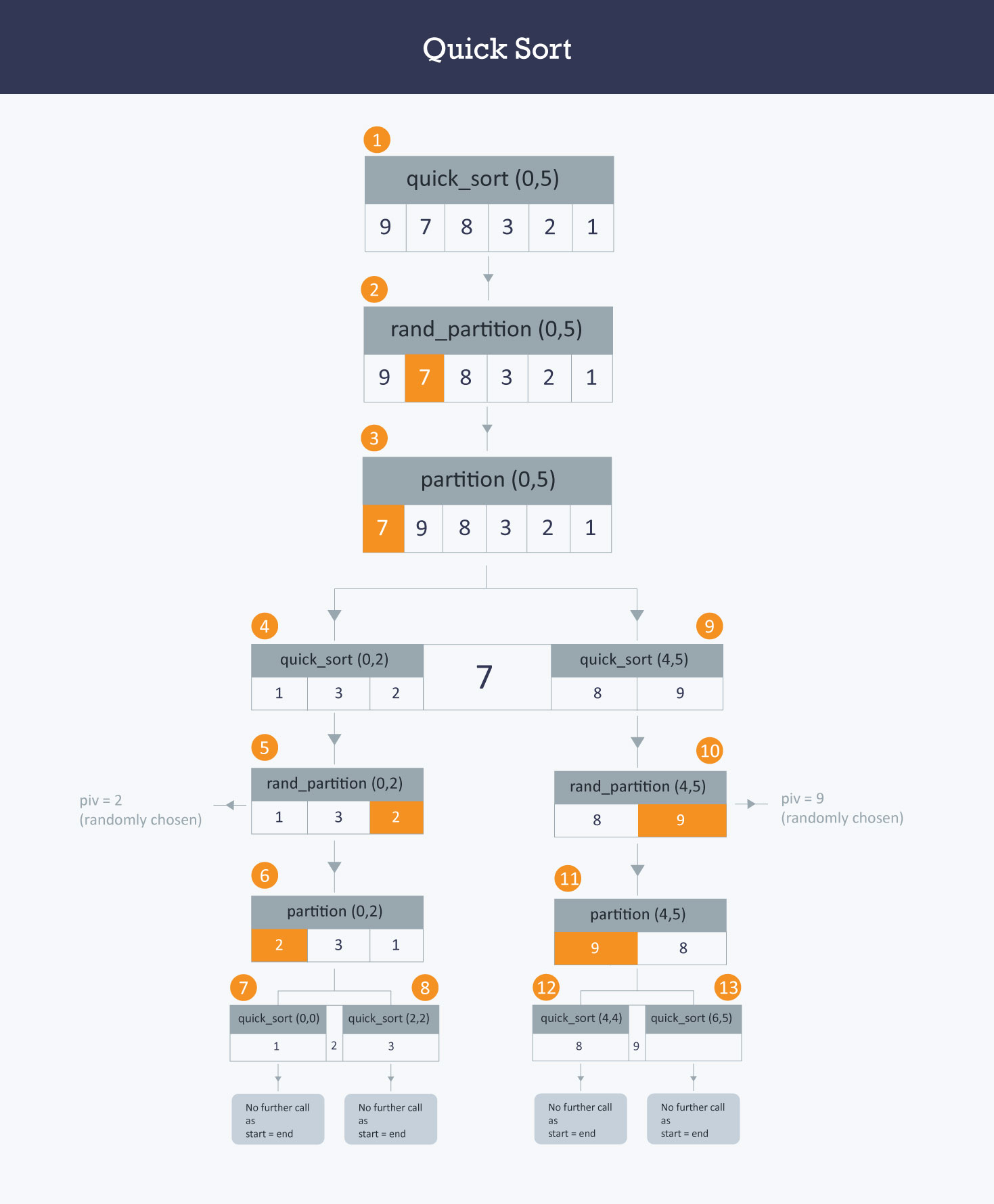
start: Leftmost position of the array

end: Rightmost position of the array

i : Boundary between the elements that are less than pivot and those greater than pivot

j : Boundary between the partitioned and unpartitioned part of array

piv: Pivot element



Here we find the proper position of the pivot element by rearranging the array using partition function. Then we divide the array into two halves left side of the pivot (elements less than pivot element) and right side of the pivot (elements greater than pivot element) and apply the same step recursively.

Example: You have an array A=[9,7,8,3,2,1] Observe in the diagram below, that the randpartition() function chooses pivot randomly as 7 and then swaps it with the first element of the array and then the partition() function call takes place, which divides the array into two halves. The first half has elements less than 7 and the other half has elements greater than 7.  
For elements less than 7, in 5th call, randpartition() function chooses 2 as pivot element randomly and then swap it with first element and call to the partition() function takes place. After the 7th and 8th call, no further calls can take place as only one element left in both the calls. Similarly, you can observe the order of calls for the elements greater than 7.

1. **Merge Sort:**

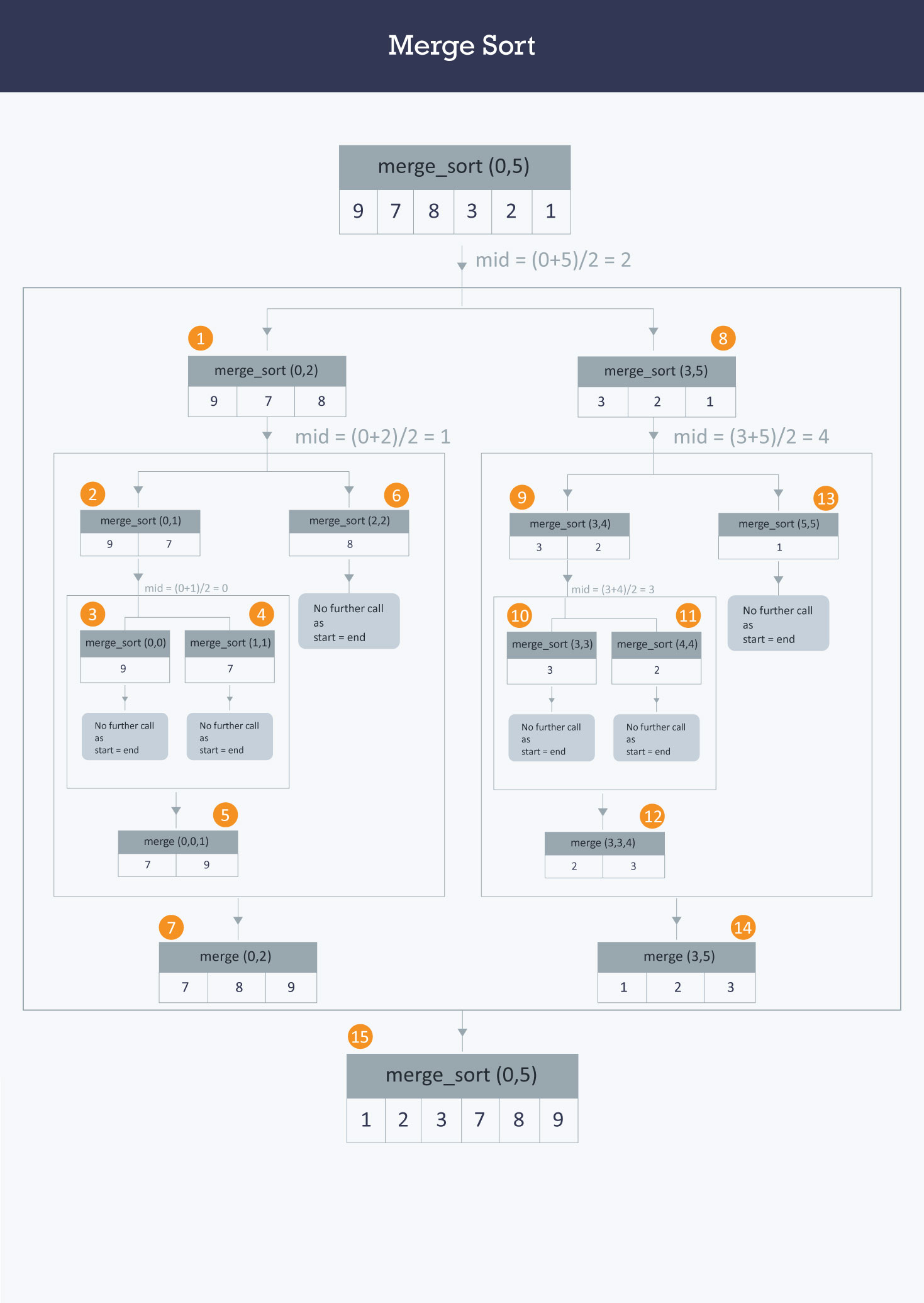
Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

**Idea:**

* Divide the unsorted list into N sublists, each containing 1 element.
* Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
* Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

**Let’s consider the following image**



As one may understand from the image above, at each step a list of size M is being divided into 2 sublists of size M/2, until no further division can be done. To understand better, consider a smaller array A containing the elements (9,7,8).

At the first step this list of size 3 is divided into 2 sublists the first consisting of elements (9,7) and the second one being (8). Now, the first list consisting of elements (9,7) is further divided into 2 sublists consisting of elements (9) and (7) respectively.

As no further breakdown of this list can be done, as each sublist consists of a maximum of 1 element, we now start to merge these lists. The 2 sub-lists formed in the last step are then merged together in sorted order using the procedure mentioned above leading to a new list (7,9). Backtracking further, we then need to merge the list consisting of element (8) too with this list, leading to the new sorted list (7,8,9).

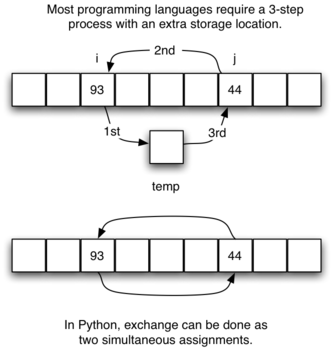


Figure 2: Exchanging Two Values in Python

1. **Selection Sort**

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

Assume that the array A=[7,5,4,2] needs to be sorted in ascending order.

The minimum element in the array i.e. 2 is searched for and then swapped with the element that is currently located at the first position, i.e. 7. Now the minimum element in the remaining unsorted array is searched for and put in the second position, and so on.

At ith iteration, elements from position 0 to i−1 will be sorted.



**Time Complexity:**

To find the minimum element from the array of N elements, N−1 comparisons are required. After putting the minimum element in its proper position, the size of an unsorted array reduces to N−1 and then N−2 comparisons are required to find the minimum in the unsorted array.

Therefore (N−1) + (N−2) + ....... + 1 = (N⋅(N−1))/2 comparisons and N swaps result in the overall complexity of O(N2).