Sorting

Sorting is the process of placing elements from a collection in some kind of order. There are many, many sorting algorithms that have been developed and analysed. Sorting a large number of items can take a substantial amount of computing resources. The efficiency of a sorting algorithm is related to the number of items being processed. For small collections, a complex sorting method may be more trouble than it is worth. The overhead may be too high.

Before getting into specific algorithms, we should think about the operations that can be used to analyse a sorting process. First, it will be necessary to compare two values to see which is smaller (or larger). To sort a collection, it will be necessary to have some systematic way to compare values to see if they are out of order. The total number of comparisons will be the most common way to measure a sort procedure. Second, when values are not in the correct position with respect to one another, it may be necessary to exchange them. This exchange is a costly operation and the total number of exchanges will also be important for evaluating the overall efficiency of the algorithm.

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. Each item “bubbles” up to the location where it belongs.

Figure SORT 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 $n - 1$ 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.

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

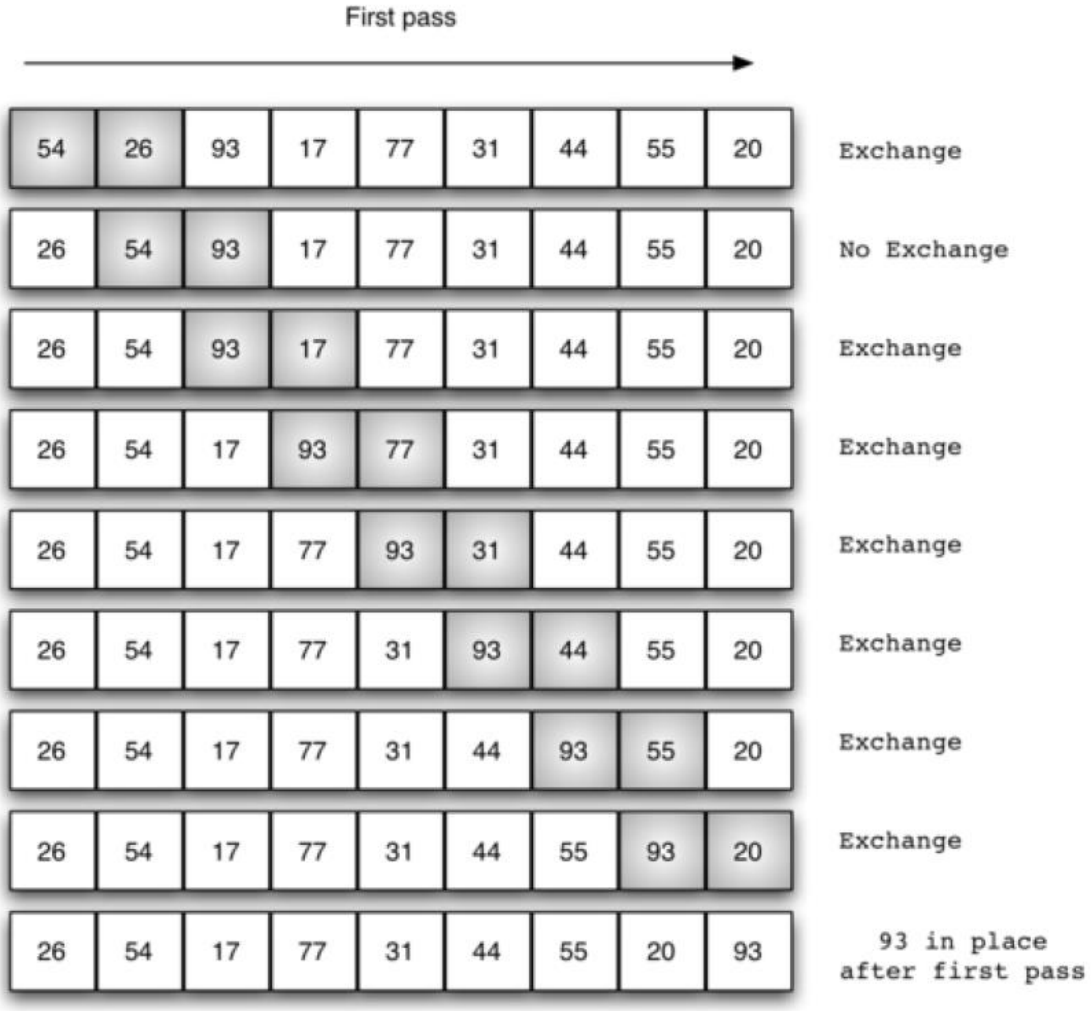

Figure SORT 1 - Bubble sort: The First Pass

```
def bubble_sort(a_list):
    for pass_num in range(len(a_list) - 1, 0, -1):
        for i in range(pass_num):
            if a_list[i] > a_list[i + 1]:
                temp = a_list[i]
                a_list[i] = a_list[i + 1]
                a_list[i + 1] = temp

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
bubble_sort(a_list)
print(a_list)
```

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 = a_list[i]
a_list[i] = a_list[j]
a_list[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 SORT 2). Using simultaneous assignment, the exchange operation can be done in one statement.

Lines 5–7 in the bubble_sort function perform the exchange of the i and (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.

To analyse the bubble sort, we should note that regardless of how the items are arranged in the initial list, $n - 1$ passes will be made to sort a list of size n. Table SORT 1 shows the number of comparisons for each pass.

In the worst case, every comparison will cause an exchange. On average, we exchange half of the time. A bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known. These “wasted” exchange operations are very costly.

However, because the bubble sort makes passes through the entire unsorted portion of the list, it has the capability to do something most sorting algorithms cannot. If during a pass there are no exchanges, then we know that the list must be sorted. A bubble sort can be modified to stop early if it finds that the list has become sorted. This means that for lists that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted list and stop.

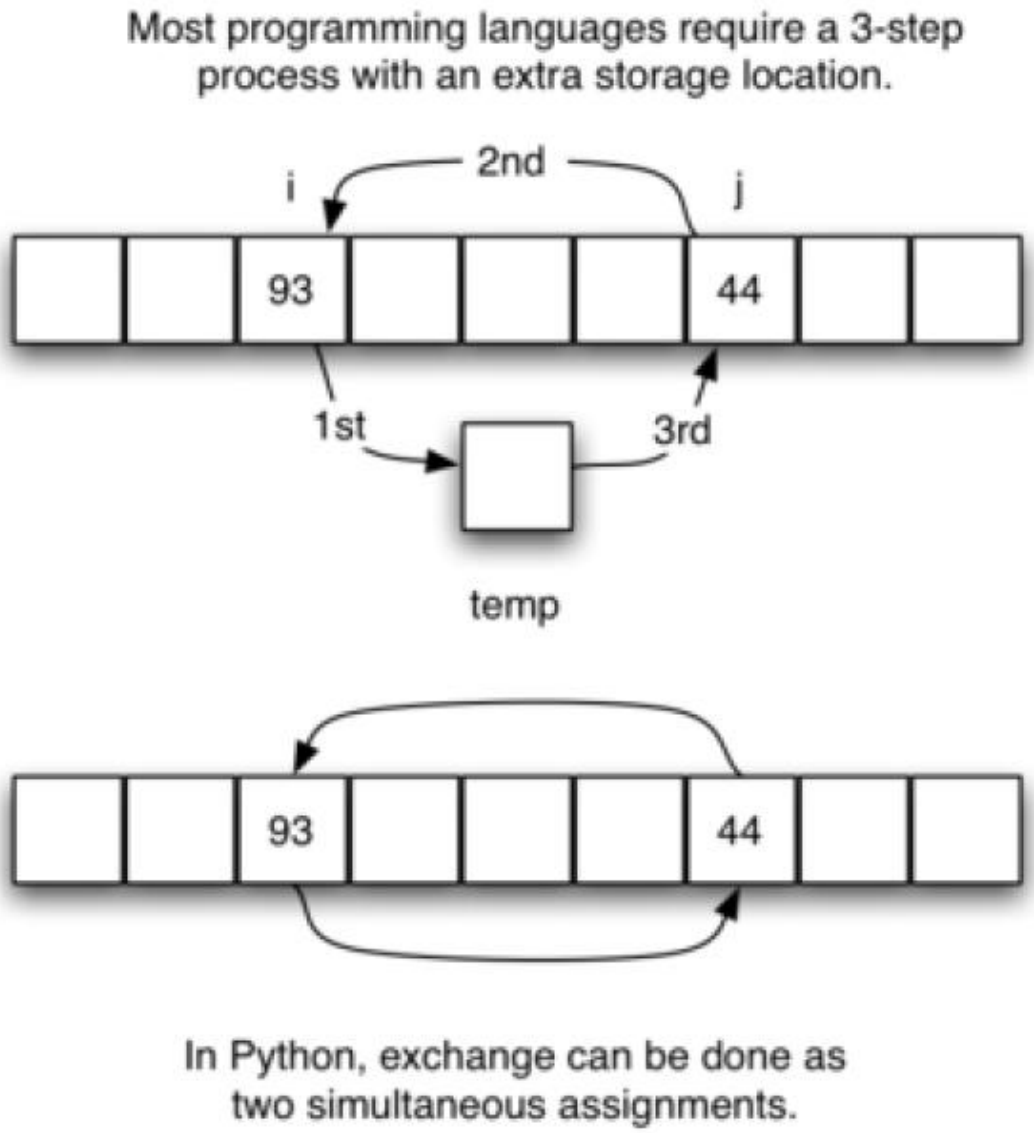

Figure SORT 2 – Exchanging Two Values in Python

Pass	Comparisons
1	$n-1$
2	$n-2$
3	$n-3$
...	...
$n-1$	1

Table SORT 1 - Comparisons for Each Pass of Bubble Sort

The code below shows this modification, which is often referred to as the **short bubble.**

```
def short_bubble_sort(a_list):
    exchanges = True
    pass_num = len(a_list) - 1
    while pass_num > 0 and exchanges:
       exchanges = False
       for i in range(pass_num):
           if a_list[i] > a_list[i + 1]:
               exchanges = True
               temp = a_list[i]
               a_list[i] = a_list[i + 1]
               a_list[i + 1] = temp
     pass_num = pass_num - 1

a_list=[20, 30, 40, 90, 50, 60, 70, 80, 100, 110]
short_bubble_sort(a_list)
print(a_list)
```

Selection Sort

The selection sort improves on the bubble sort by making only one exchange for every pass through the list. To do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires $n - 1$ passes to sort n items, since the final item must be in place after the $(n - 1)$st pass.

Figure SORT 3 shows the entire sorting process. On each pass, the largest remaining item is selected and then placed in its proper location. The first pass places 93, the second pass places 77, the third places 55, and so on. The function is shown below.

```
def selection_sort(a_list):
   for fill_slot in range(len(a_list) - 1, 0, -1):
      pos_of_max = 0
      for location in range(1, fill_slot + 1):
         if a_list[location] > a_list[pos_of_max]:
            pos_of_max = location

      temp = a_list[fill_slot]
      a_list[fill_slot] = a_list[pos_of_max]
      a_list[pos_of_max] = temp

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
selection_sort(a_list)
print(a_list)
```

You may see that the selection sort makes the same number of comparisons as the bubble sort. However, due to the reduction in the number of exchanges, the selection sort typically executes faster in benchmark studies. In fact, for our list, the bubble sort makes 20 exchanges, while the selection sort makes only 8.

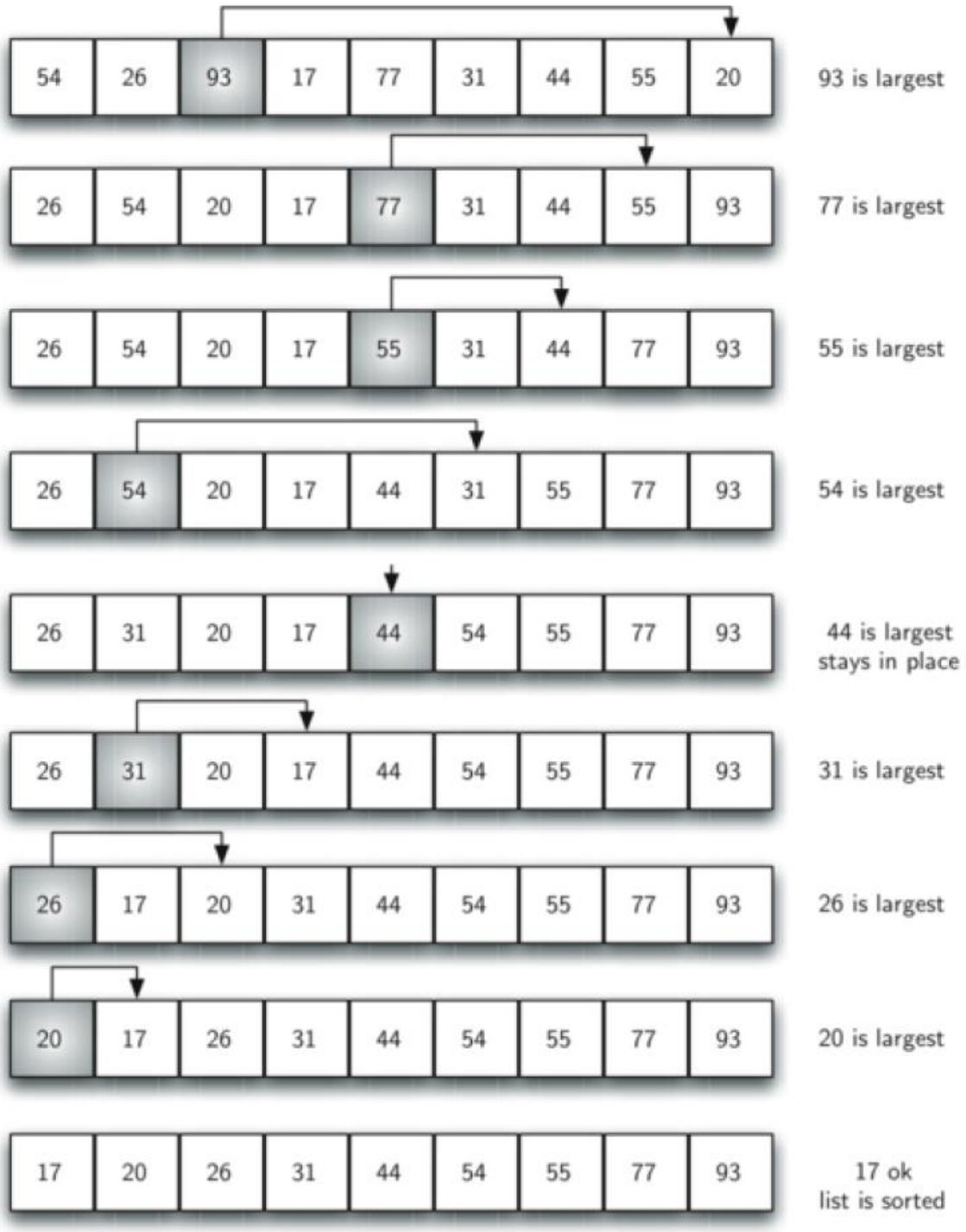

Figure SORT 3 – Selection Sort

The Insertion Sort

The insertion sort works in a slightly different way. It always maintains

a sorted sublist in the lower positions of the list. Each new item is then "inserted" back into the previous sublist such that the sorted sublist is one item larger.

Figure SORT 4 shows the insertion sorting process. The shaded items represent the ordered sublists as the algorithm makes each pass. We begin by assuming that a list with one item (position 0) is already sorted. On each pass, one for each item 1 through $n - 1$, the current item is checked against those in the already sorted sublist. As we look back into the already sorted sublist, we shift those items that are greater to the right. When we reach a smaller item or the end of the sublist, the current item can be inserted.

Figure SORT 5 shows the fifth pass in detail. At this point in the algorithm, a sorted sublist of five items consisting of 17, 26, 54, 77, and 93 exists. We want to insert 31 back into the already sorted items. The first comparison against 93 causes 93 to be shifted to the right. 77 and 54 are also shifted. When the item 26 is encountered, the shifting process stops and 31 is placed in the open position. Now we have a sorted sublist of six items.

The implementation of insertion_sort shows that there are again $n - 1$ passes to sort n items. The iteration starts at position 1 and moves through position $n - 1$, as these are the items that need to be inserted back into the sorted sublists. Line 8 performs the shift operation that moves a value up one position in the list, making room behind it for the insertion. Remember that this is not a complete exchange as was performed in the previous algorithms.

The maximum number of comparisons for an insertion sort is the sum of the first $n-1$ integers. In the best case, only one comparison needs to be done on each pass. This would be the case for an already sorted list.

One note about shifting versus exchanging is also important. In general, a shift operation requires approximately a third of the processing work of an exchange since only one assignment is performed. In benchmark studies, insertion sort will show very good performance.

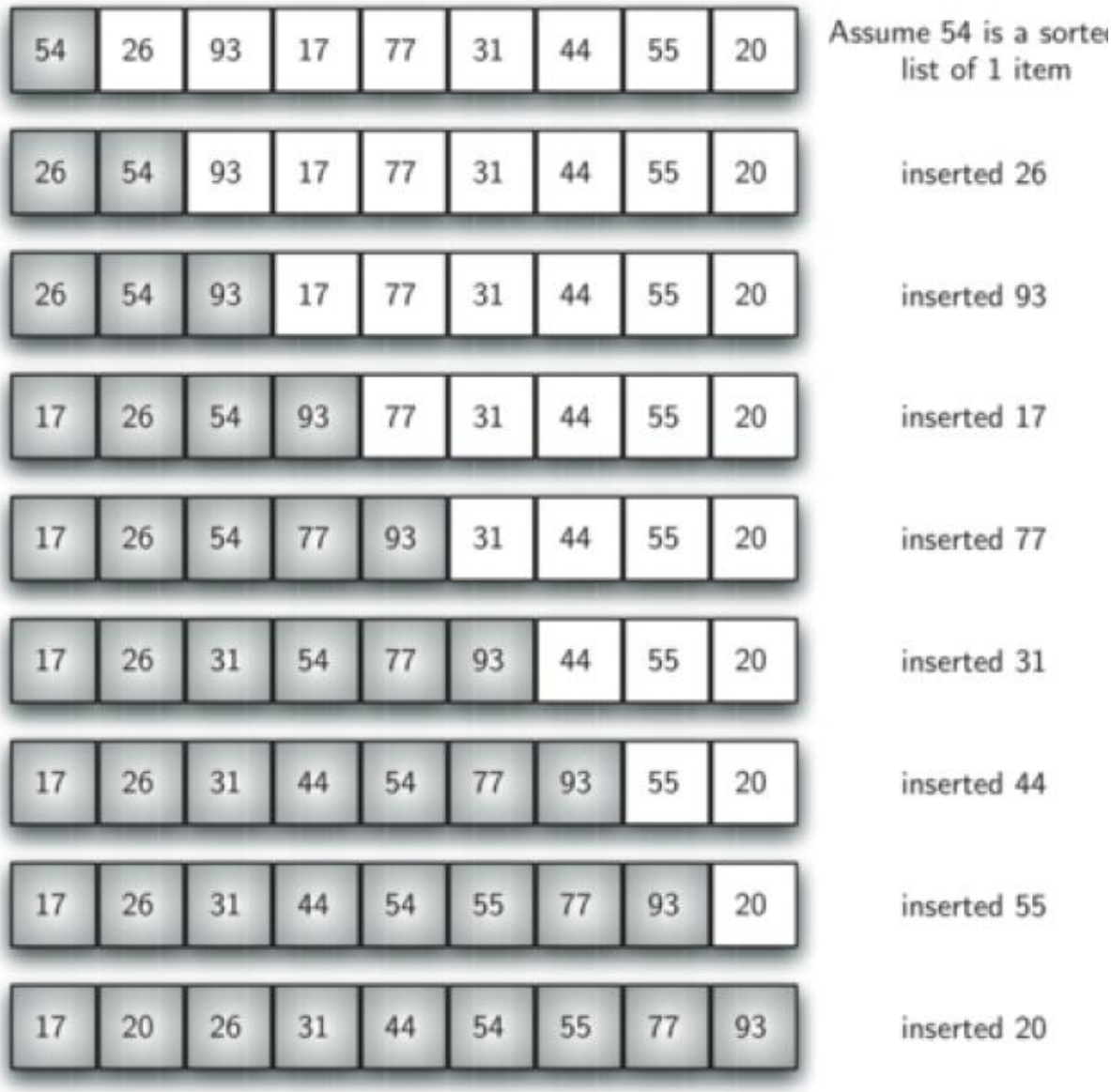

Figure SORT 4 – Insertion Sort

```
def insertion_sort(a_list):
   for index in range(1, len(a_list)):

     current_value = a_list[index]
     position = index

     while position > 0 and a_list[position - 1] > current_value:
        a_list[position] = a_list[position - 1]
        position = position - 1

     a_list[position] = current_value

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
insertion_sort(a_list)
print(a_list)
```

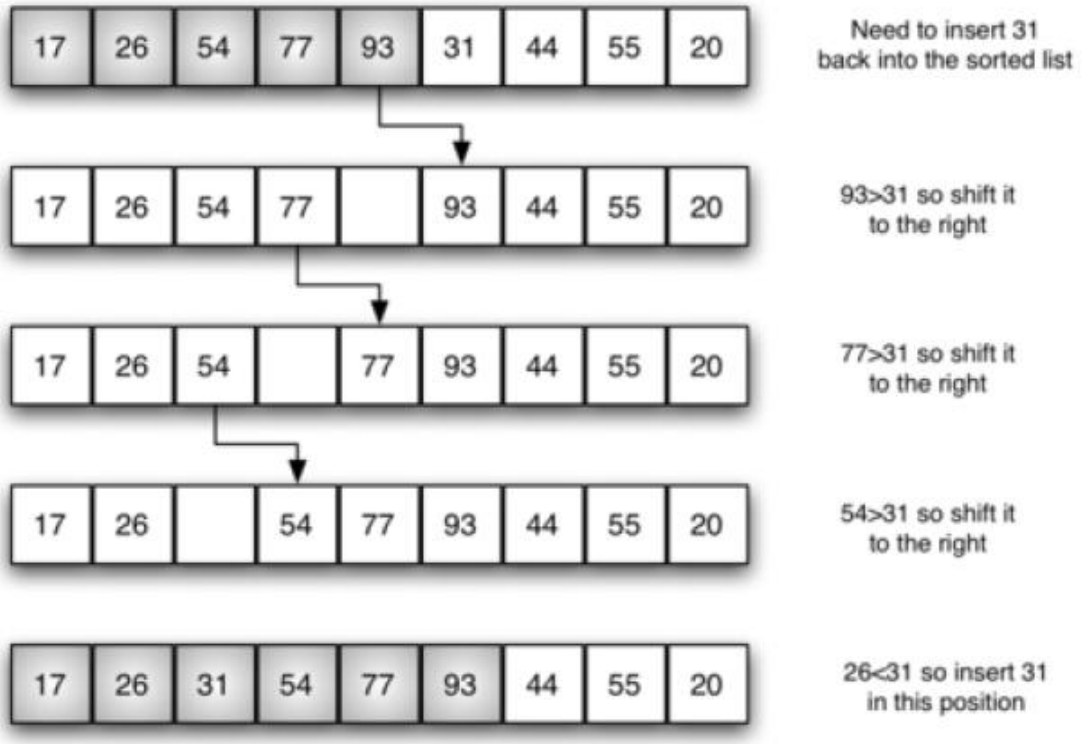

Figure SORT 5 - Insertion Sort: Fifth Pass of the Sort

Shell Sort

The shell sort, sometimes called the “diminishing increment sort,” improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. The unique way that these sublists are chosen is the key to the shell sort.

Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment i, sometimes called the gap, to create a sublist by choosing all items that are i items apart.

This can be seen in Figure SORT 6. This list has nine items. If we use an increment of three, there are three sublists, each of which can be sorted by an insertion sort. After completing these sorts, we get the list shown in Figure SORT 7. Although this list is not completely sorted, something very interesting has happened. By sorting the sublists, we have moved the items closer to where they actually belong.

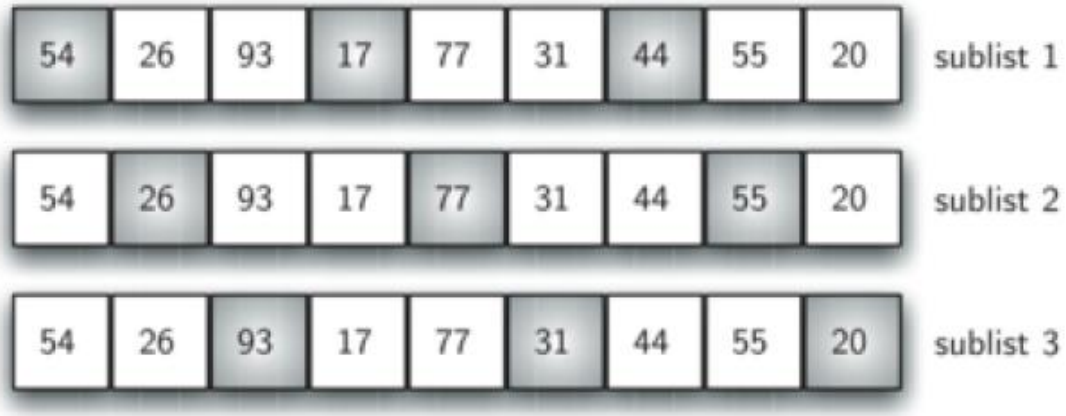

Figure SOR 6 – A Shell Sort with Increments of Three

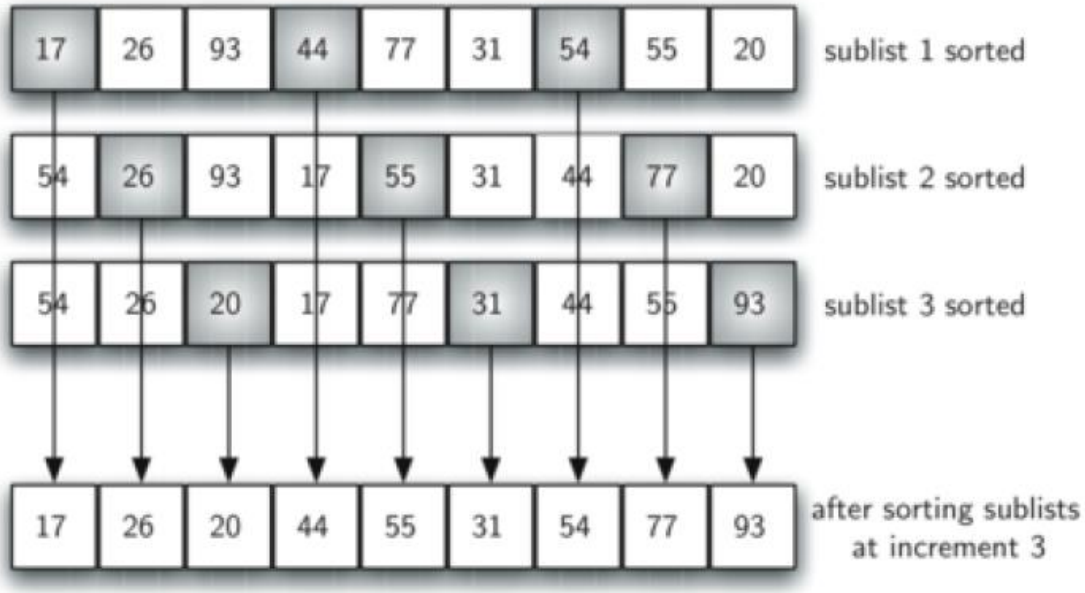

Figure SORT 7 - A Shell Sort after Sorting Each Sublist

Figure SORT 8 shows a final insertion sort using an increment of one; in other words, a standard insertion sort. Note that by performing the earlier sublist sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order. For this case, we need only four more shifts to complete the process.

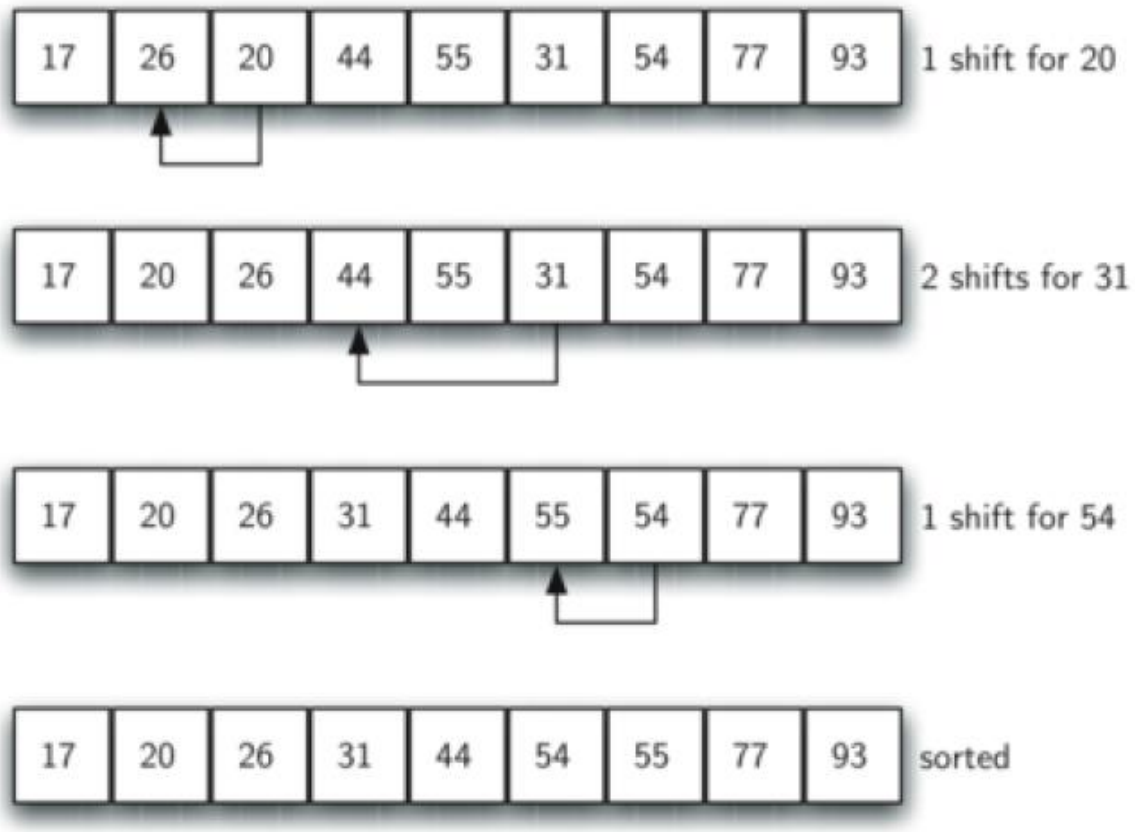

Figure SORT 8 - Shell Sort: A Final Insertion Sort with Increment of 1

We said earlier that the way in which the increments are chosen is the unique feature of the shell sort. The function shell_sort shown below uses a different set of increments. In this case, we begin with $n/2$ sublists. On the next pass, $n/4$ sublists are sorted. Eventually, a single listis sorted with the basic insertion sort. Figure SORT 9 shows the first sublists for our example using this increment.

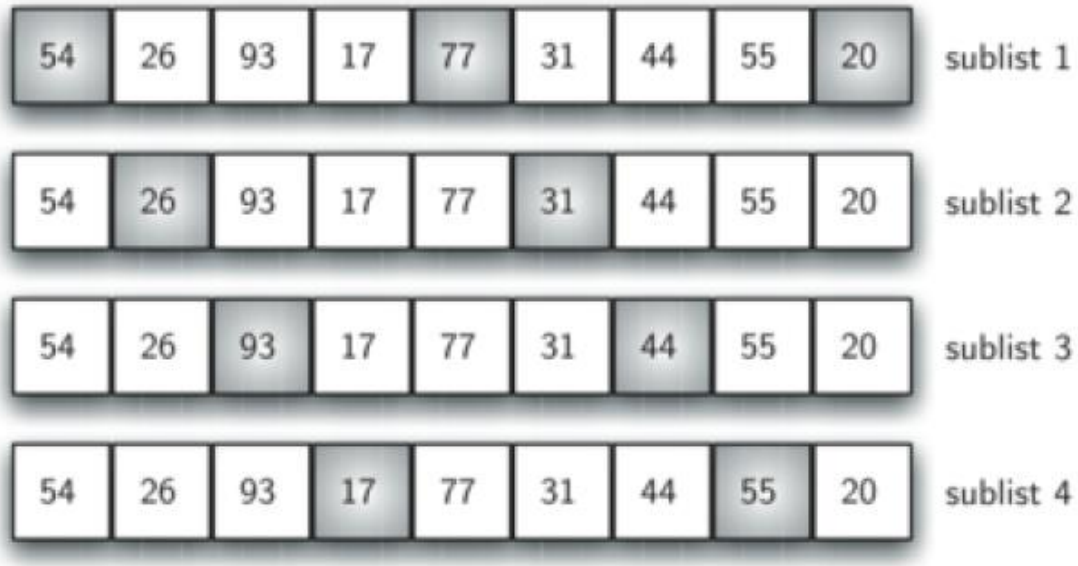

Figure SORT 9 - Initial Sublists for a Shell Sort

The following invocation of the shell_sort function shows the partially sorted lists after each increment, with the final sort being an insertion sort with an increment of one.

```
def shell_sort(a_list):
  sublist_count = len(a_list) // 2
   while sublist_count > 0:
     for start_position in range(sublist_count):
       gap_insertion_sort(a_list, start_position, sublist_count)

     print("After increments of size", sublist_count, "The list is",
        a_list)

     sublist_count = sublist_count // 2

def gap_insertion_sort(a_list, start, gap):
  for i in range(start + gap, len(a_list), gap):
    current_value = a_list[i]
     position = i

       while position >= gap and a_list[position - gap] >
          current_value:
        a_list[position] = a_list[position - gap]
          position = position - gap

       a_list[position] = current_value

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
shell_sort(a_list)
print(a_list)
```

At first glance you may think that a shell sort cannot be better than an insertion sort, since it does a complete insertion sort as the last step. It turns out, however, that this final insertion sort does not need to do very many comparisons (or shifts) since the list has been pre-sorted by earlier incremental insertion sorts, as described above. In other words, each pass produces a list that is “more sorted” than the previous one. This makes the final pass very efficient.

The Merge Sort

We now turn our attention to using a divide and conquer strategy as a way to improve the performance of sorting algorithms. The first algorithm we will study is the merge sort. Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list. Figure SORT 10 shows our familiar example list as it is being split by merge_sort. Figure SORT 11 shows the simple lists, now sorted, as they are merged back together.

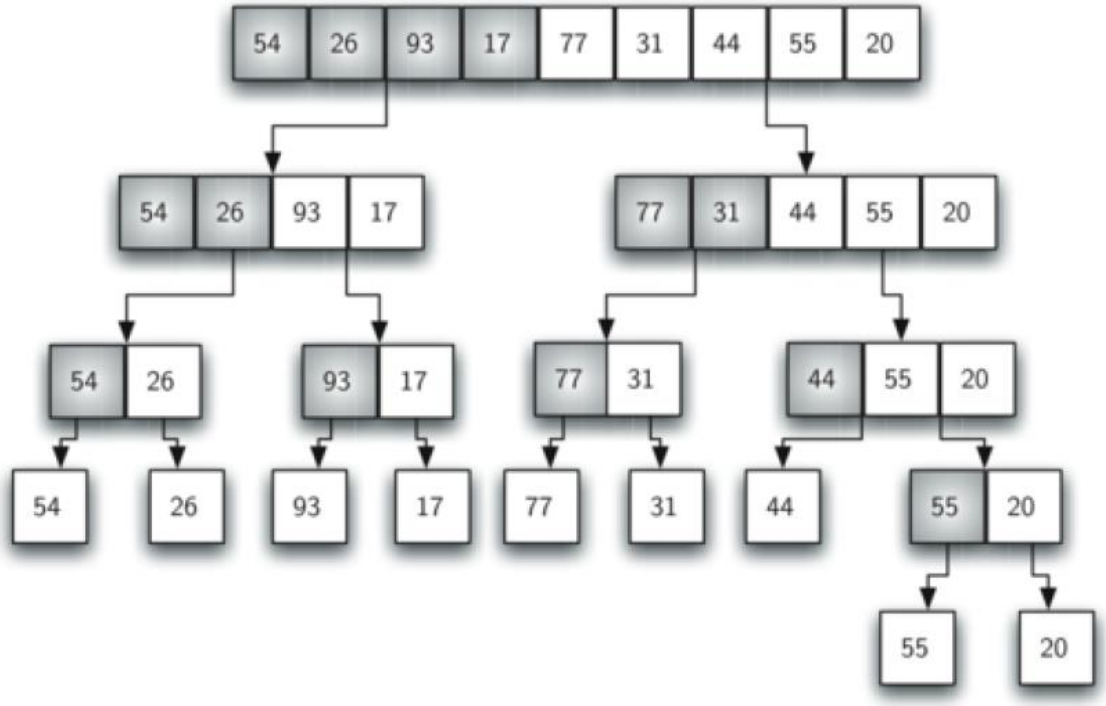

Figure SORT 10 – Splitting the List in a Merge Sort

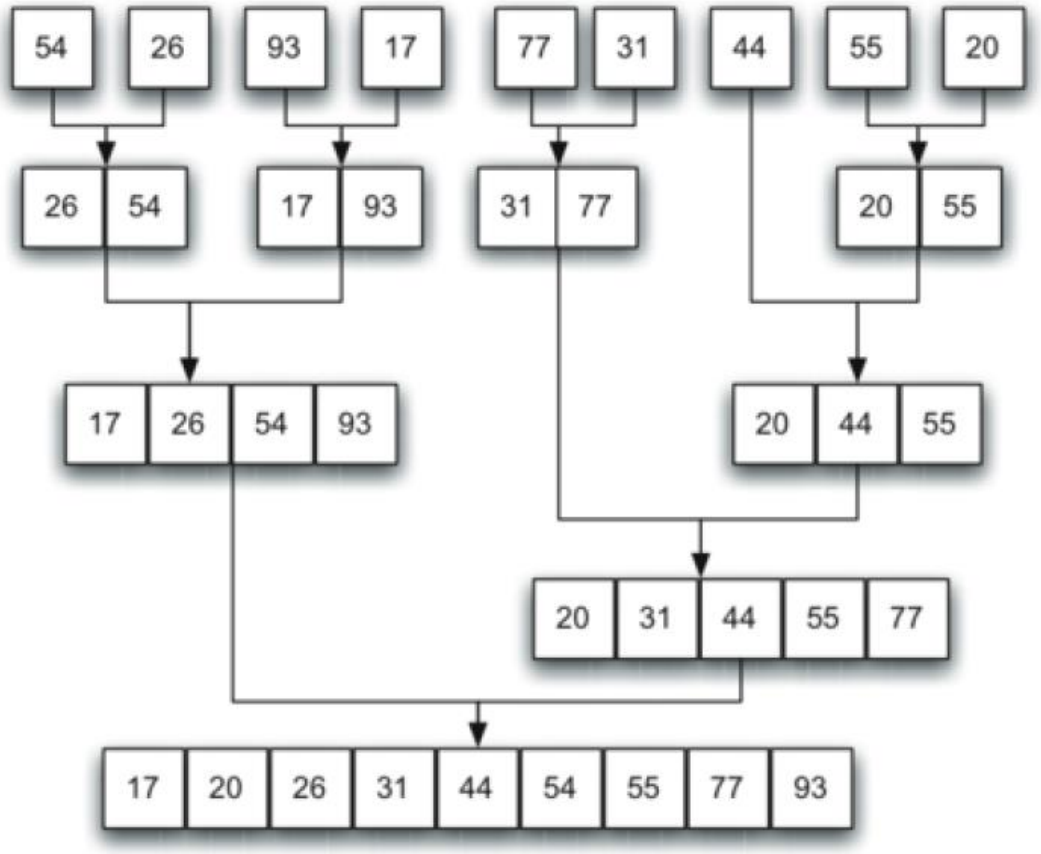

Figure SORT 11 - Lists as They Are Merged Together

The merge_sort function shown below begins by asking the base case question. If the length of the list is less than or equal to one, then we already have a sorted list and no more processing is necessary. If, on the other hand, the length is greater than one, then we use the Python slice operation to extract the left and right halves. It is important to note that the list may not have an even number of items. That does not matter, as the lengths will differ by at most one.

Once the merge_sort function is invoked on the left half and the right half (lines 8–9), it is assumed they are sorted. The rest of the function (lines 11–31) is responsible for merging the two smaller sorted lists into a larger sorted list. Notice that the merge operation places the items back into the original list (a_list) one at a time by repeatedly taking the smallest item from the sorted lists.

```
def merge_sort(a_list):
    print("Splitting ", a_list)
    if len(a_list) > 1:
        mid = len(a_list) // 2
        left_half = a_list[:mid]
        right_half = a_list[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = 0
        j = 0
        k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                a_list[k] = left_half[i]
                i = i + 1
            else:
                a_list[k] = right_half[j]
                j = j + 1
            k = k + 1

        while i < len(left_half):
            a_list[k] = left_half[i]
            i = i + 1
            k = k + 1

        while j < len(right_half):
            a_list[k] = right_half[j]
            j = j + 1
            k = k + 1
    print("Merging ", a_list)

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
merge_sort(a_list)
print(a_list)
```

The merge_sort function has been augmented with a print statement (line 2) to show the contents of the list being sorted at the start of each invocation. There is also a print statement (line 32) to show the merging process. The transcript shows the result of executing the function on our example list. Note that the list with 44, 55, and 20 will not divide evenly. The first split gives [44] and the second gives [55, 20]. It is easy to see how the splitting process eventually yields a list that can be immediately merged with other sorted lists.

In order to analyse the merge_sort function, we need to consider the two distinct processes that make up its implementation. First, the list is split into halves. We already computed (in a binary search) that we can divide a list in half log n times where n is the length of the list. The second process is the merge. Each item in the list will eventually be processed and placed on the sorted list. So, the merge operation which results in a list of size n requires n operations.

It is important to notice that the merge_sort function requires extra space to hold the two halves as they are extracted with the slicing operations. This additional space can be a critical factor if the list is large and can make this sort problematic when working on large data sets.

The Quick Sort

The quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the pivot value. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort.

Figure SORT 12 shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The partition process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.

Figure SORT 12 – The First Pivot Value for a Quick Sort

Partitioning begins by locating two position markers – let's call them left_mark and right_mark – at the beginning and end of the remaining items in the list (positions 1 and 8 in Figure SORT 13). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. Figure SORT 13 shows this process as we locate the position of 54.

We begin by incrementing left_mark until we locate a value that is greater than the pivot value. We then decrement right_mark until we find a value that is less than the pivot value.

At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.

At the point where right_mark becomes less than left_mark, we stop. The position of right_mark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place (Figure SORT 14). In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.

The quick_sort function shown below invokes a recursive function, quick_sort_helper. quick_sort_helper begins with the same base case as the merge sort. If the length of the list is less than or equal to one, it is already sorted. If it is greater, then it can be partitioned and recursively sorted. The partition function implements the process described earlier.

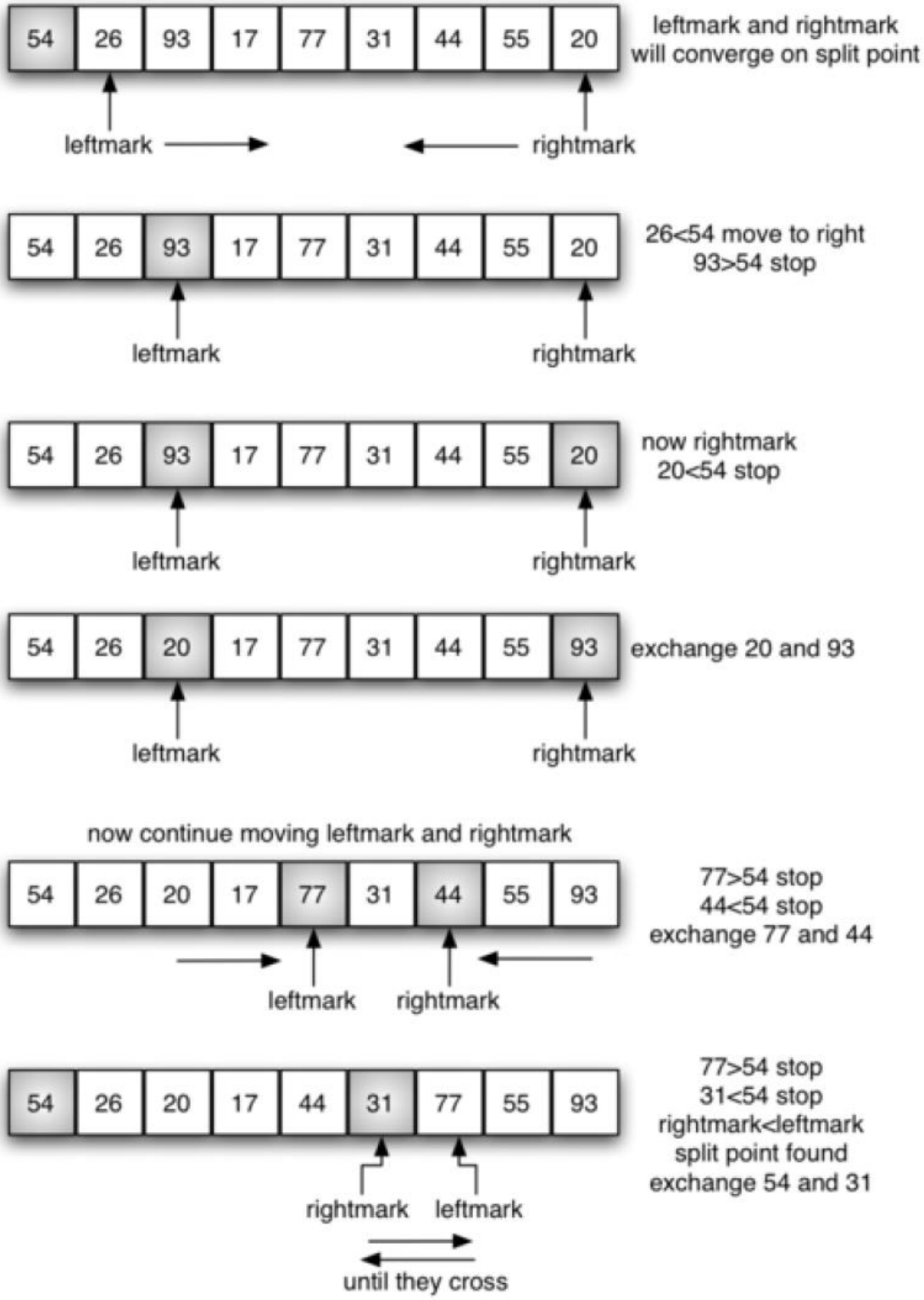

Figure SORT 13 - Finding the Split Point for 54

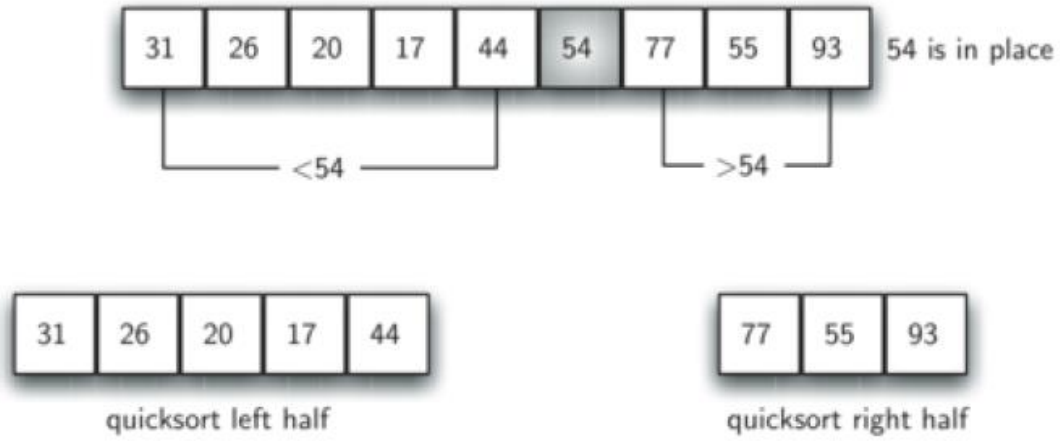

Figure SORT 14 - Completing the Partition Process to Find the Split Point for 54

```
def quick_sort(a_list):
  quick_sort_helper(a_list, 0, len(a_list) - 1)

def quick_sort_helper(a_list, first, last):
  if first < last:

     split_point = partition(a_list, first, last)

     quick_sort_helper(a_list, first, split_point - 1)
     quick_sort_helper(a_list, split_point + 1, last)


def partition(a_list, first, last):
  pivot_value = a_list[first]

  left_mark = first + 1
  right_mark = last

  done = False
  while not done:

     while left_mark <= right_mark and \
           a_list[left_mark] <= pivot_value:
        left_mark = left_mark + 1

     while a_list[right_mark] >= pivot_value and \
           right_mark >= left_mark:
```

```
            right_mark = right_mark - 1

        if right_mark < left_mark:
            done = True
        else:
            temp = a_list[left_mark]
            a_list[left_mark] = a_list[right_mark]
            a_list[right_mark] = temp

    temp = a_list[first]
    a_list[first] = a_list[right_mark]
    a_list[right_mark] = temp


    return right_mark

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
quick_sort(a_list)
print(a_list)
```