The Art of Data Structures

Sorting

Alan Beadle

CSC 162: The Art of Data Structures

UNIVERSITY of ROCHESTER

Agenda

- To be able to explain and implement various sorting algorithms
  - Bubble
  - Selection
  - Insertion
  - Shell
  - Merge
  - Quick

Sorting

Sorting

- This is the process of organizing data in some particular order
  - Numbers, increasing order
  - Words, alphabetically
  - etc
- Some algorithms benefit with pre-sorted data, e.g. binary search

Sorting

- Sorting is an important area of computer science
- Many sorting algorithms have been developed, and analyzed
- Sorting can take significant time, and is related to the number of items to process

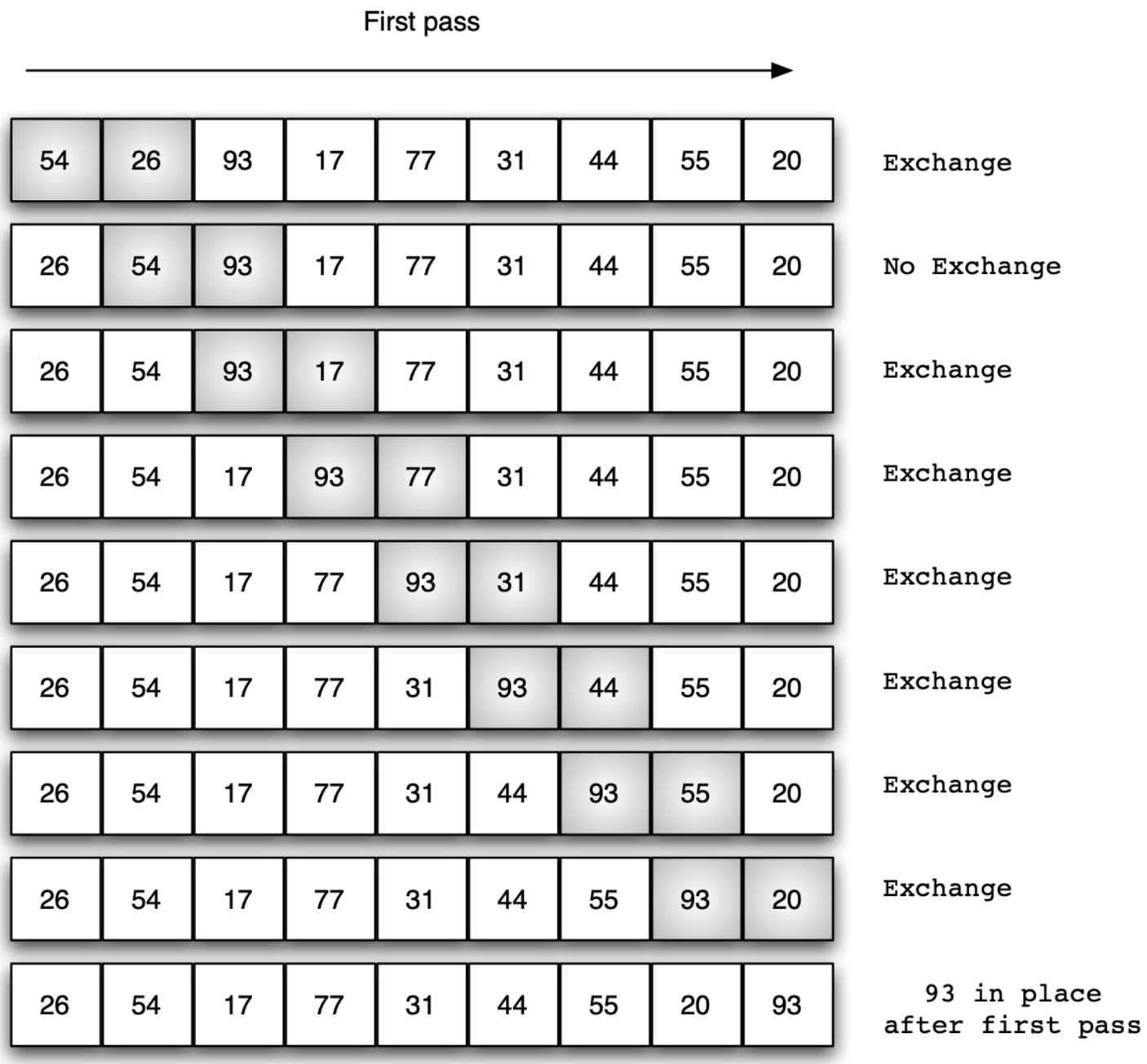
Sorting

- Sorting requires two main operations:
  - **Comparisons** of items to see if they are out of order; comparisons will be an important metric
  - **Exchange** of items can be a costly operation, and also an important metric

Bubble Sort

Bubble Sort

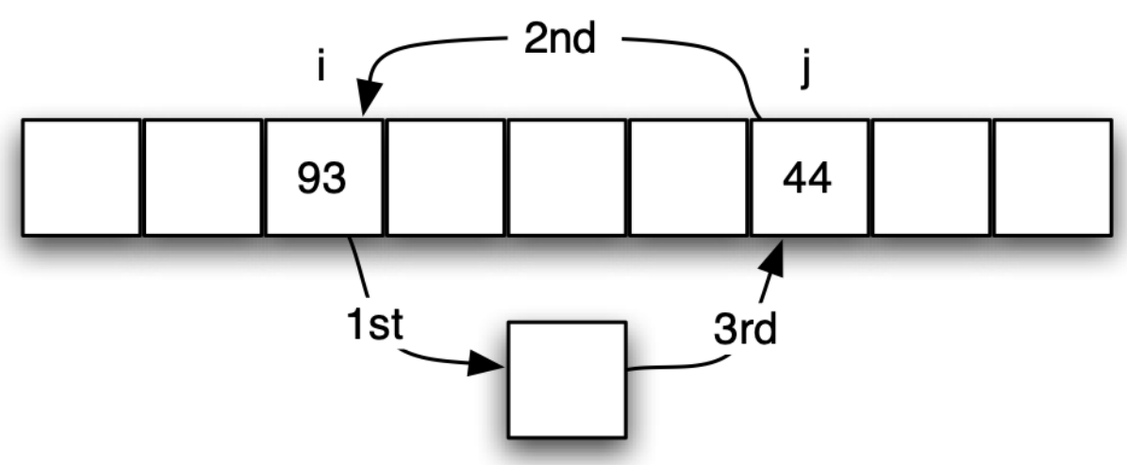
`bubble_sort`: *The First Pass*

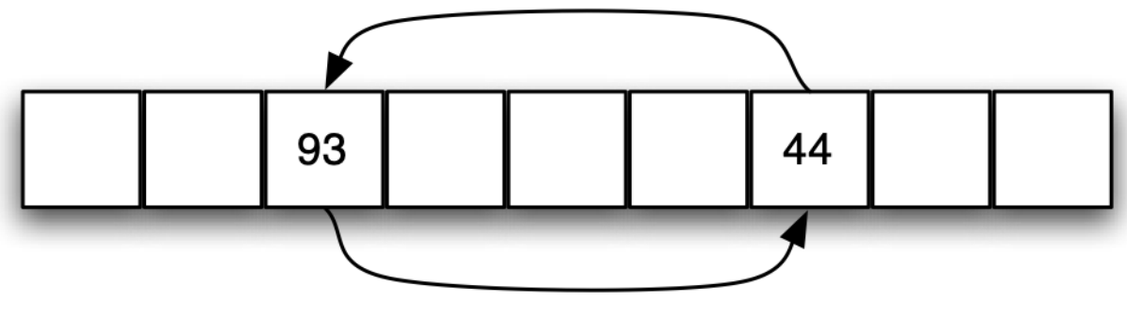


Bubble Sort

Exchanging Two Values in Python

Most programming languages require a 3-step process with an extra storage location.





In Python, exchange can be done as two simultaneous assignments.

Bubble Sort

Implementation

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

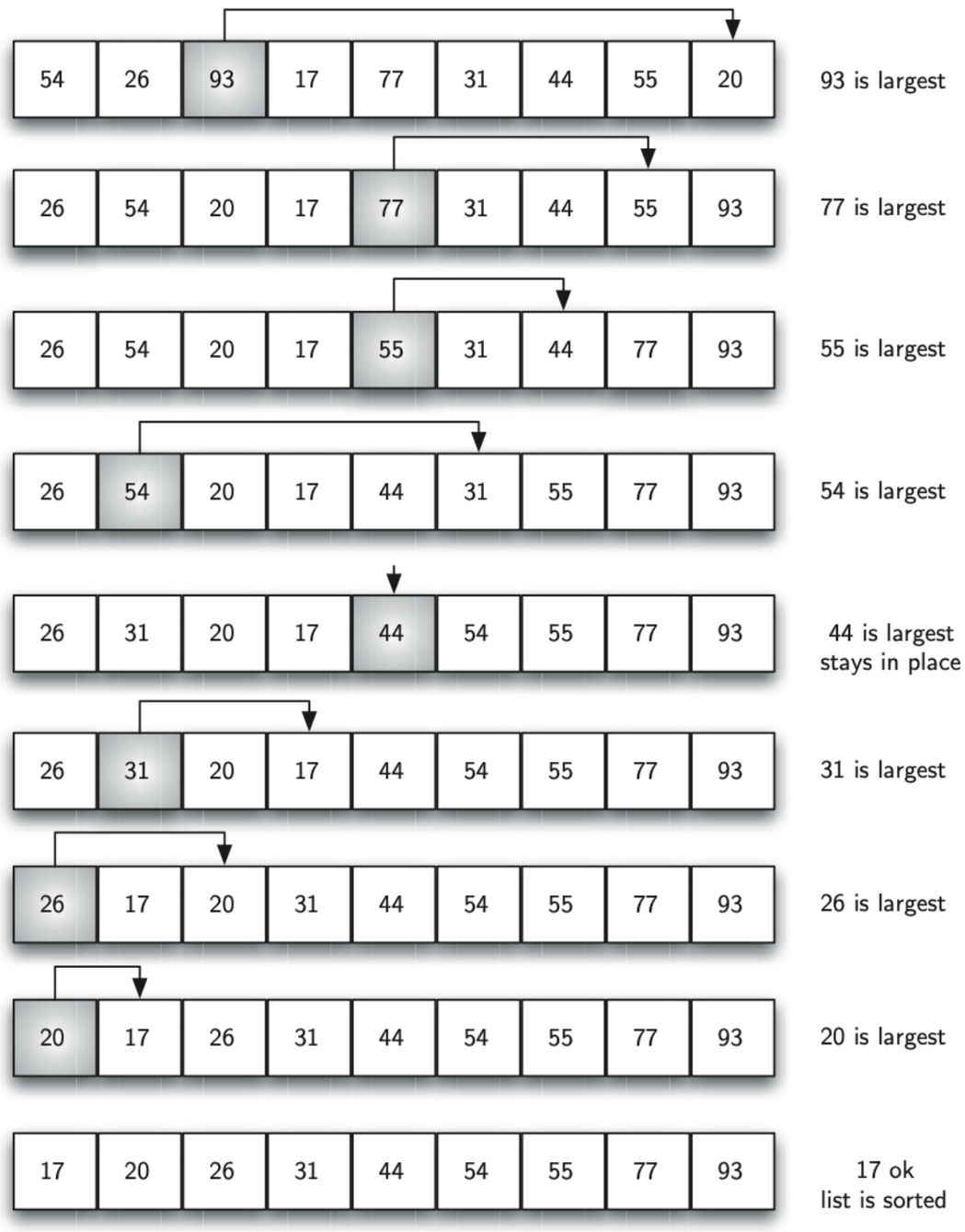
Bubble Sort

Modified Implementation (cont.)

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

Selection Sort

Selection Sort



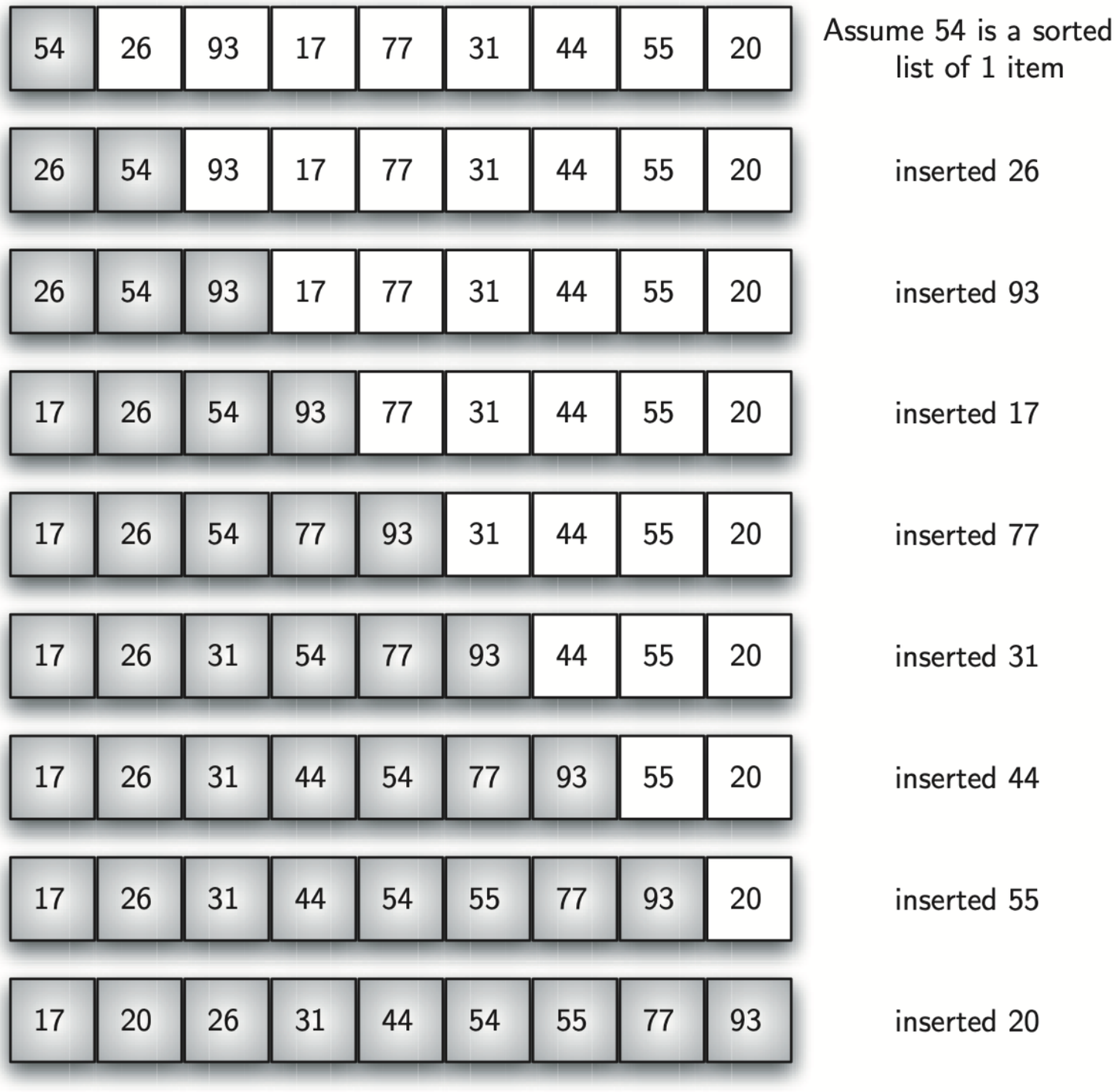
Selection Sort

Implementation

```
def selection_sort(alist):
    for fillslot in range(len(alist)-1, 0, -1):
        position_of_max = 0
        for location in range(1, fillslot+1):
            if alist[location] > alist[position_of_max]:
                position_of_max = location

        temp = alist[fillslot]
        alist[fillslot] = alist[position_of_max]
        alist[position_of_max] = temp
```

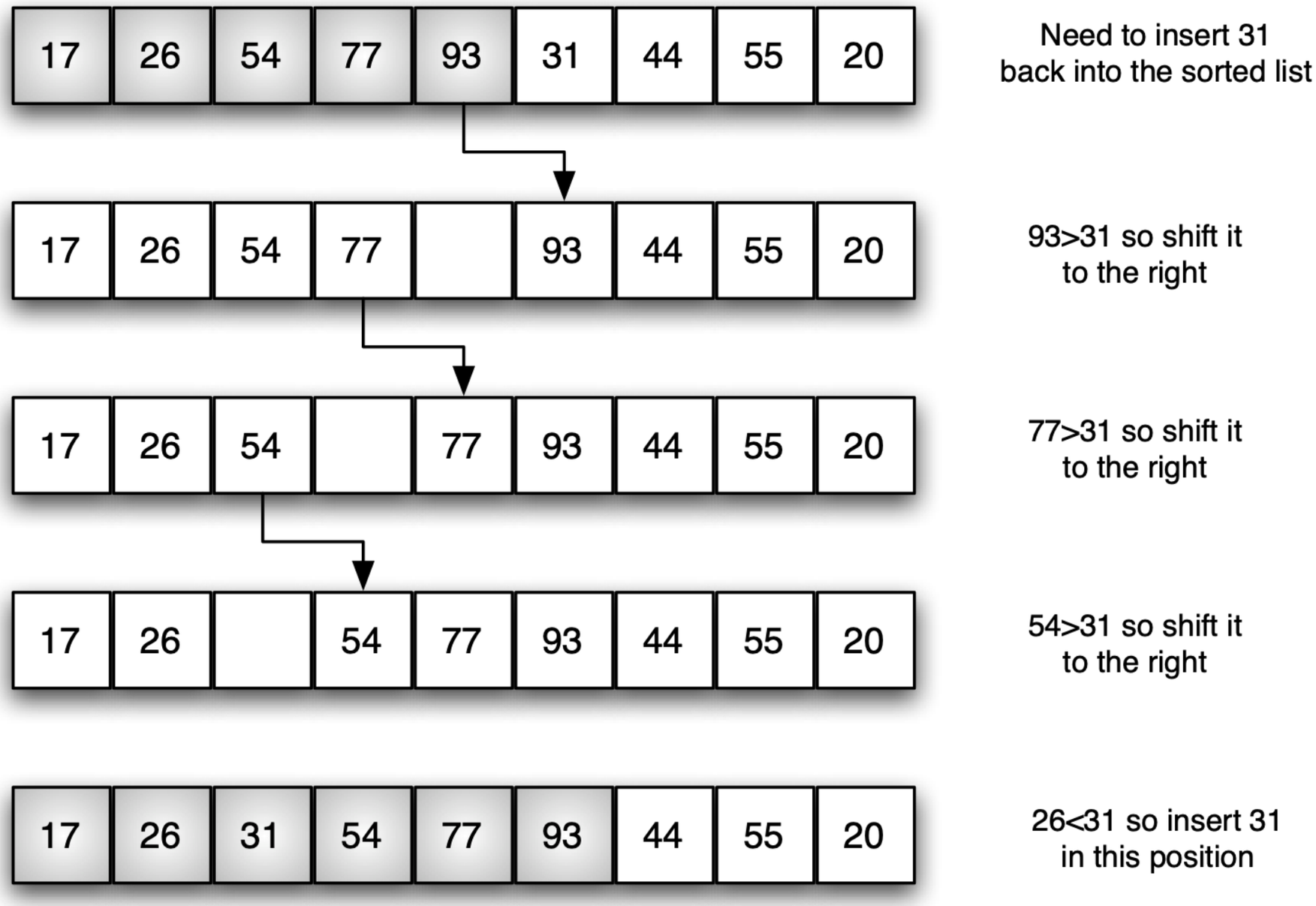
Insertion Sort

Insertion Sort



Insertion Sort

`insertion_sort`: *The Fifth Pass*



Insertion Sort

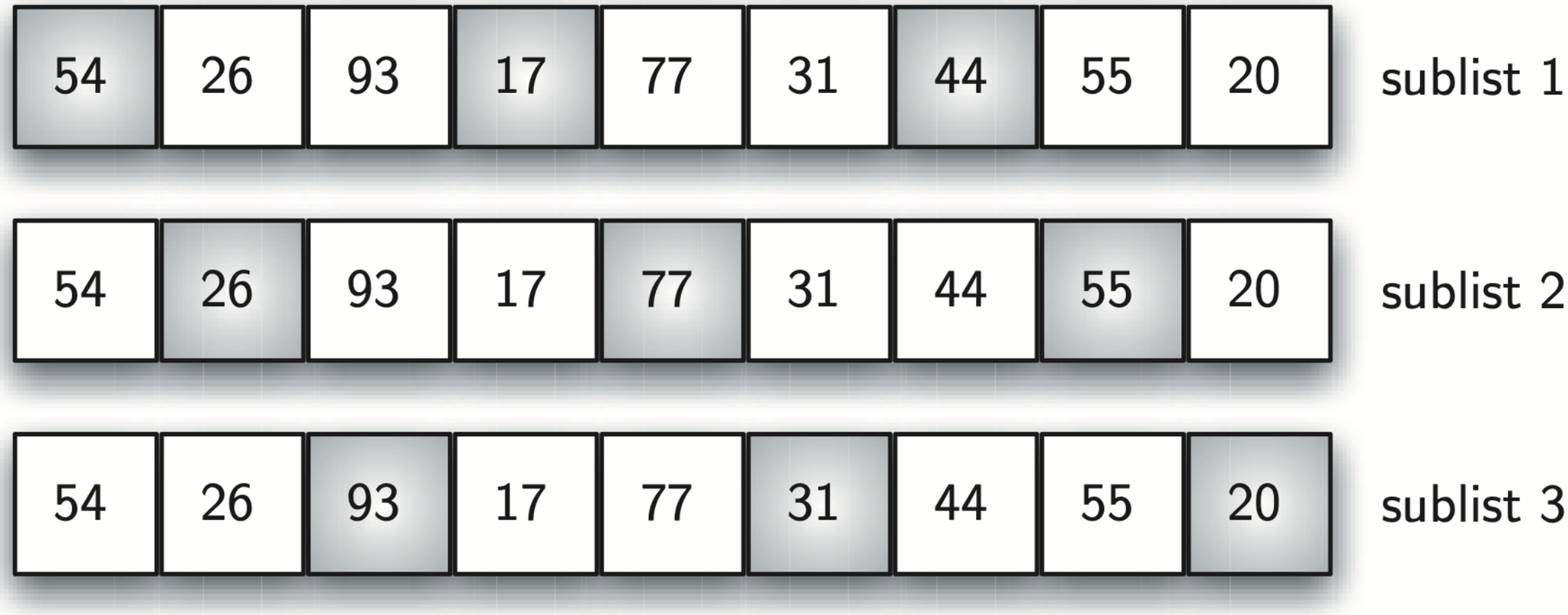
Implementation (cont.)

```
def insertion_sort(alist):
    for index in range(1, len(alist)):
        currentvalue = alist[index]
        position = index

        while position > 0 and alist[position-1] > currentvalue:
            alist[position] = alist[position-1]
            position = position-1

        alist[position] = currentvalue
```

Shell Sort

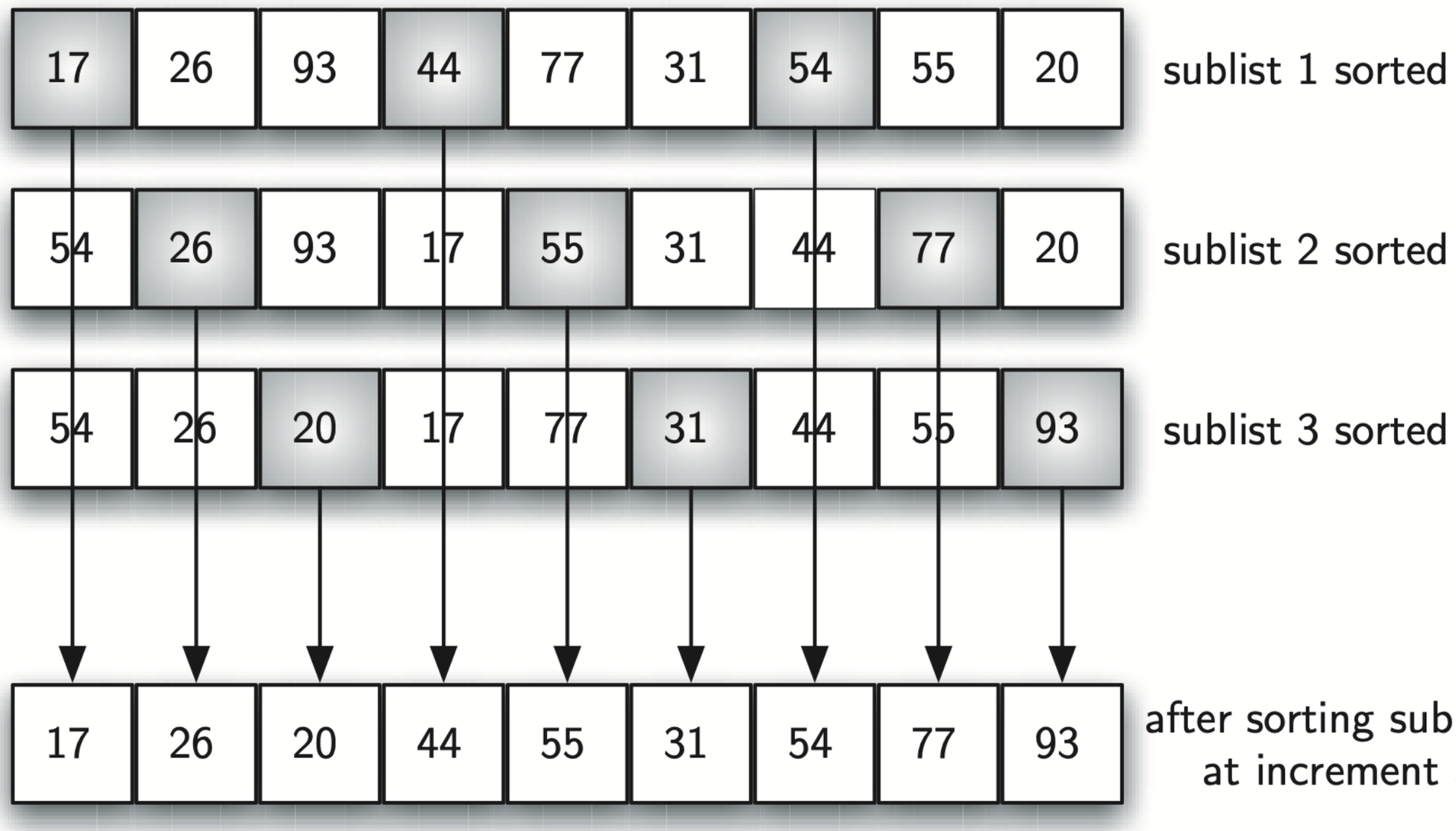
Shell Sort

With Increments of Three



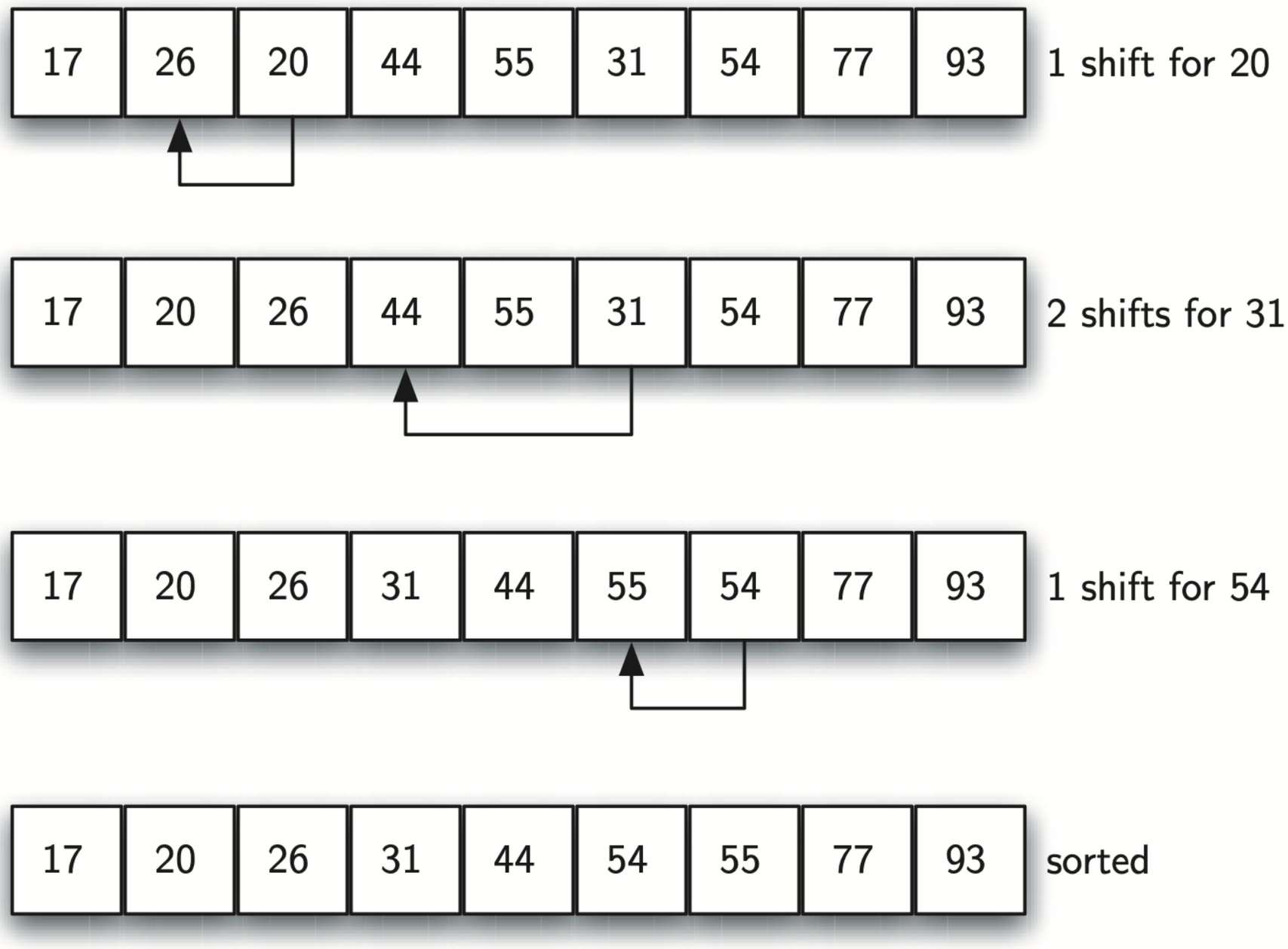
Shell Sort

After Sorting Each Sublist



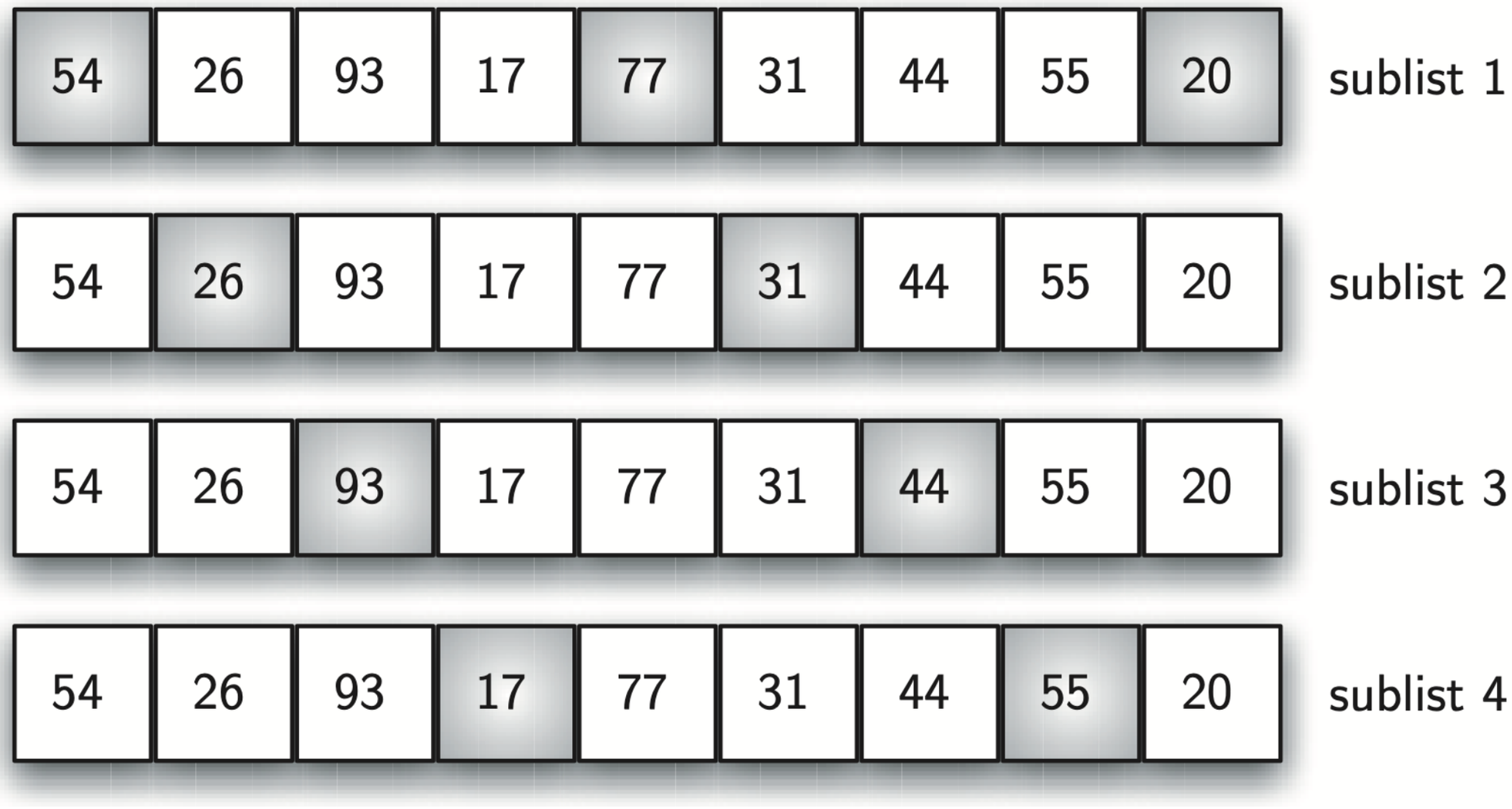
Shell Sort

A Final Insertion Sort with Increment of 1



Shell Sort

Initial Sublists for a Shell Sort



Shell Sort

Implementation

```
def shell_sort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:
        for startposition in range(sublistcount):
            gap_insertion_sort(alist, startposition, sublistcount)

    print("After increments of size", sublistcount, "The list is", alist)

    sublistcount = sublistcount // 2
```

Shell Sort

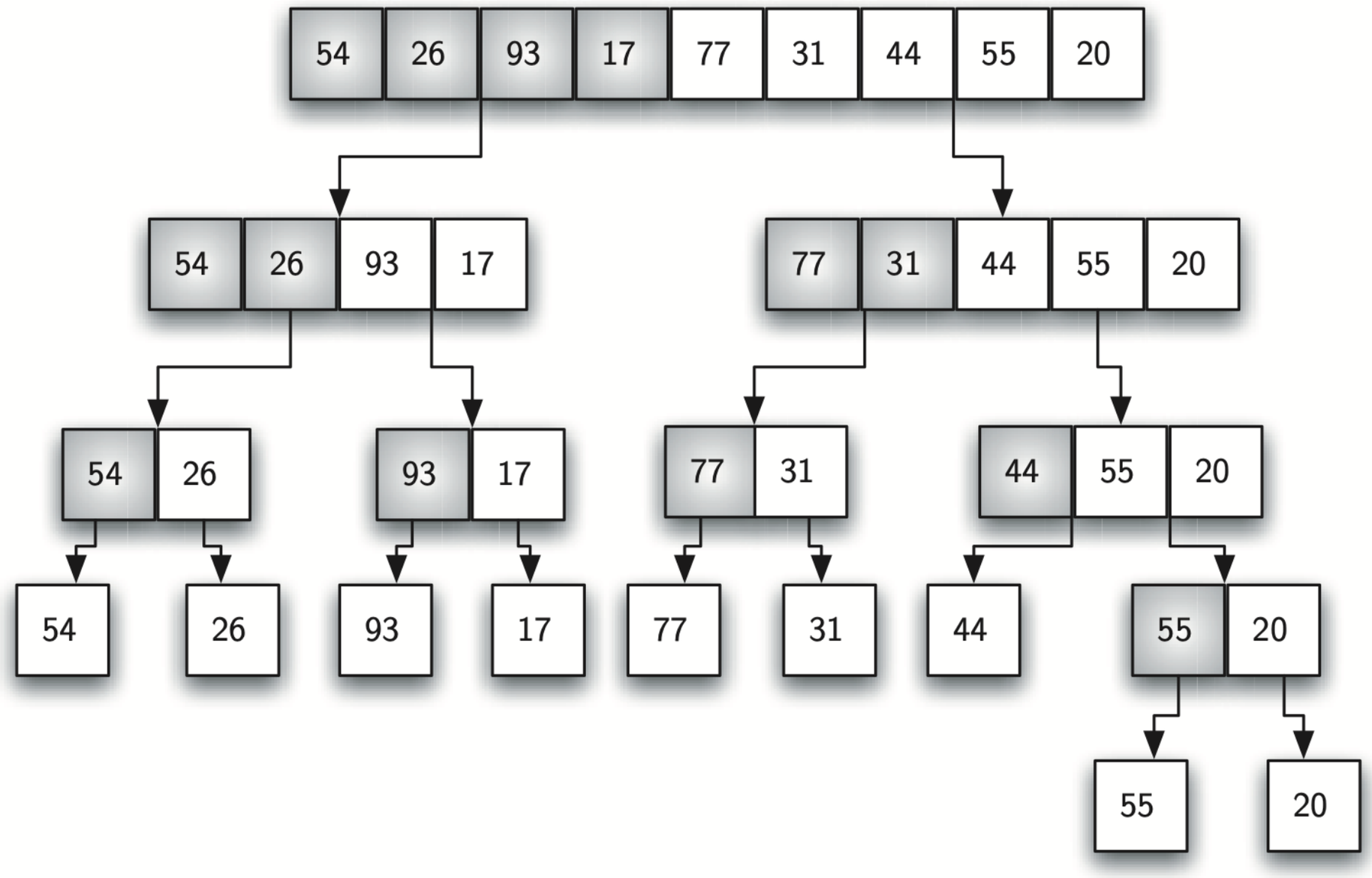
Implementation

```
def gap_insertion_sort(alist, start, gap):
    for i in range(start+gap, len(alist), gap):
        currentvalue = alist[i]
        position = i

        while position >= gap and \
              alist[position-gap] > currentvalue:
            alist[position] = alist[position-gap]
            position = position-gap

        alist[position] = currentvalue
```

Merge Sort

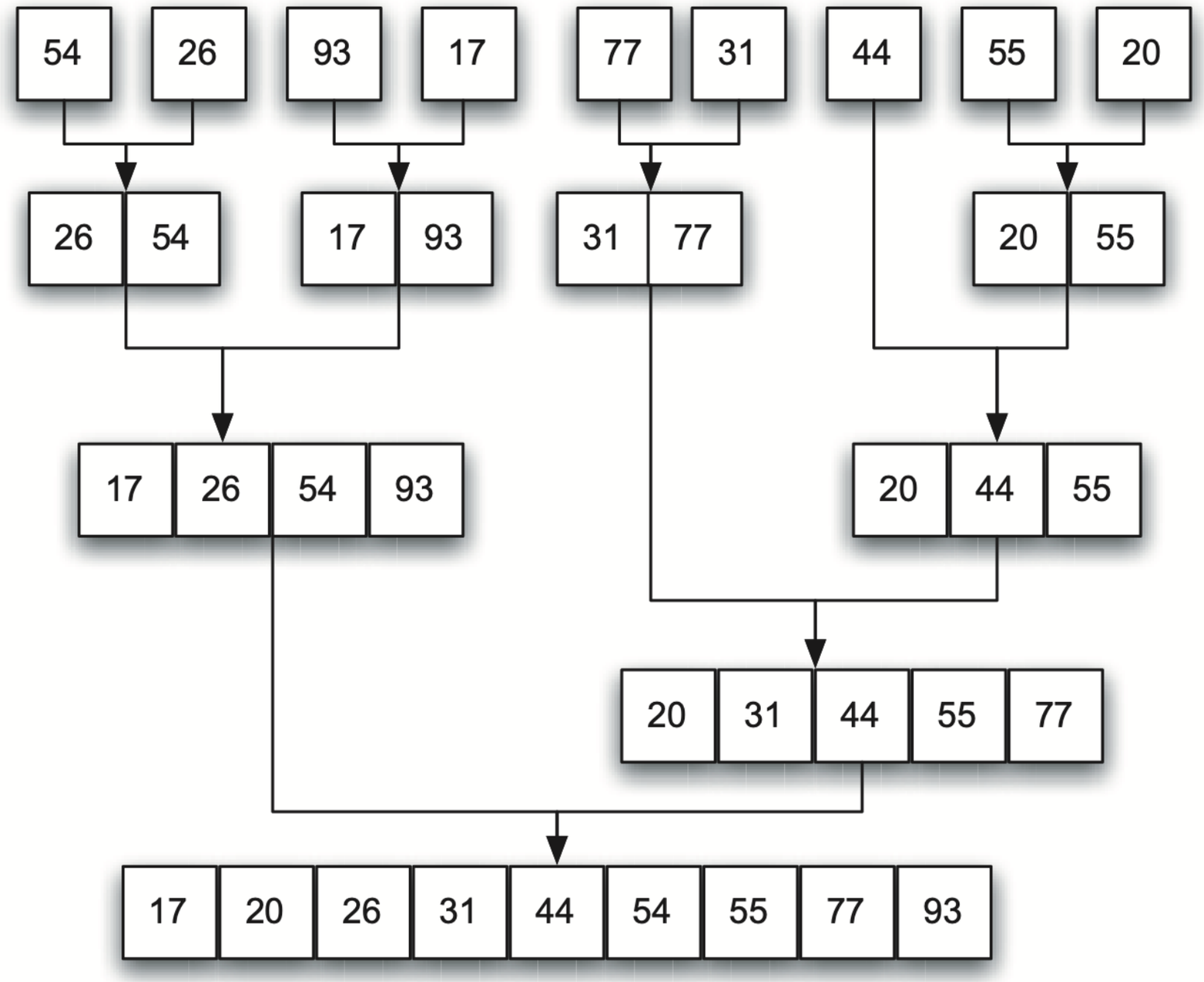
Merge Sort

Splitting and Merging



Merge Sort

Splitting and Merging



Merge Sort

Implementation

```
def merge_sort(alist):
    print("Splitting ", alist)
    if len(alist) > 1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        merge_sort(lefthalf)
        merge_sort(righthalf)

        i = 0
        j = 0
        k = 0
```

Merge Sort

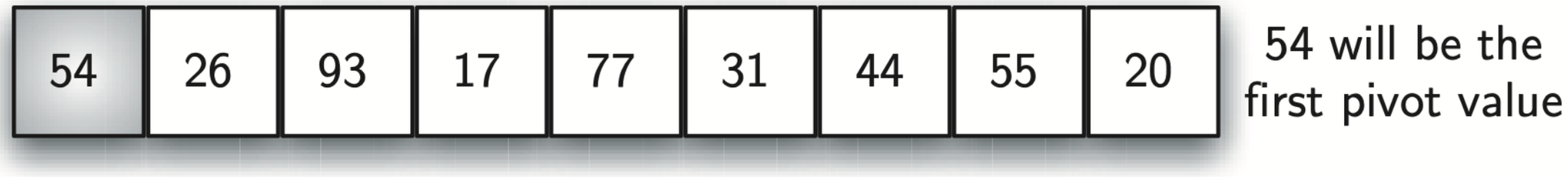
Implementation (cont.)

```
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k] = lefthalf[i]
                i = i+1
            else:
                alist[k] = righthalf[j]
                j = j+1
            k = k+1

        while i < len(lefthalf):
            alist[k] = lefthalf[i]
            i = i+1
            k = k+1

        while j < len(righthalf):
            alist[k] = righthalf[j]
            j = j+1
            k = k+1
    print("Merging ", alist)
```

Quick Sort

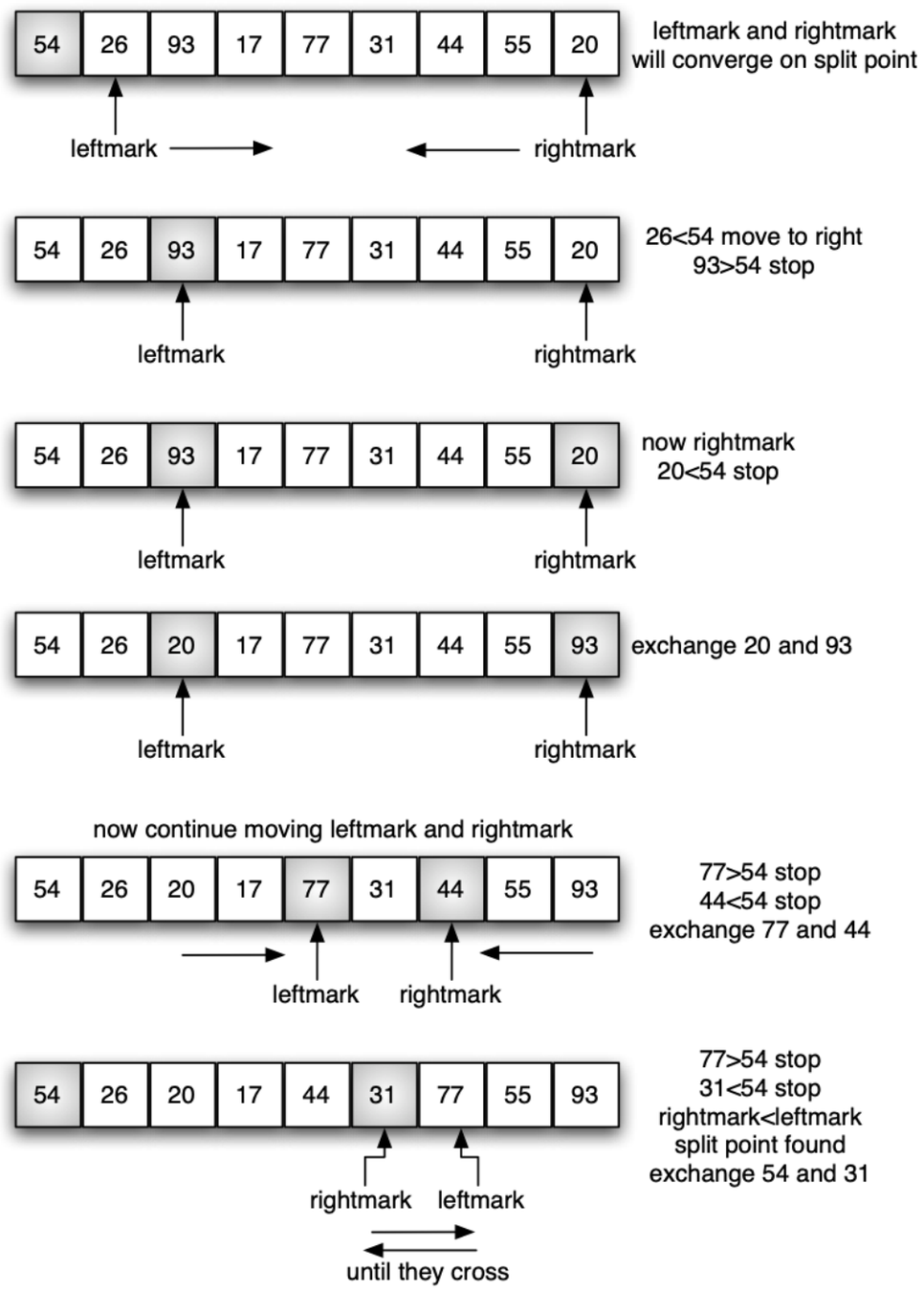
Quick Sort

The First Pivot Value



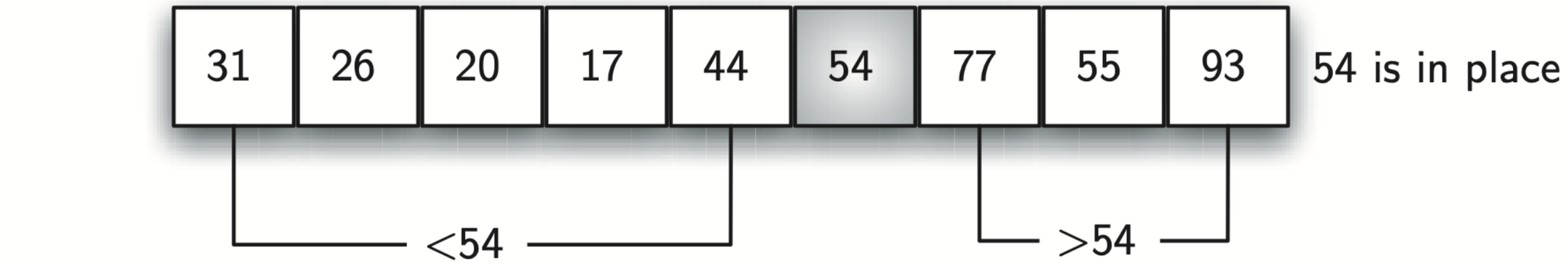
Quick Sort

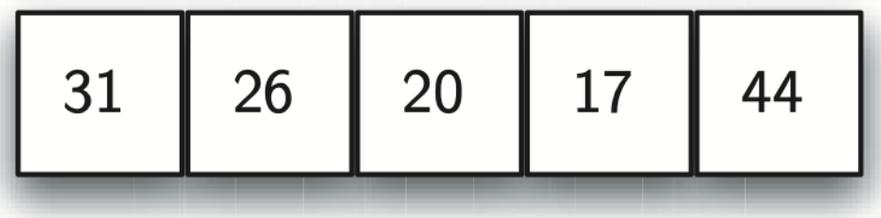
Finding the Split Point for 54



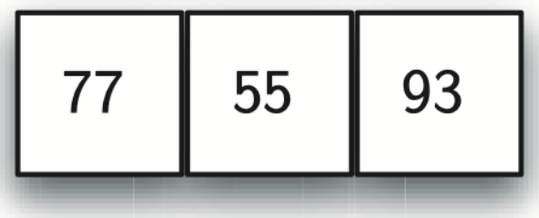
Quick Sort

Completing the Partition Process to Find the Split Point for 54





quicksort left half



quicksort right half

Quick Sort

Implementation

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

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

      splitpoint = partition(alist,first,last)

      quick_sort_helper(alist, first, splitpoint-1)
      quick_sort_helper(alist, splitpoint+1, last)
```

Quick Sort

Implementation (cont.)

```
def partition(alist, first, last):
    pivotvalue = alist[first]

    leftmark = first+1
    rightmark = last

    done = False
    while not done:
        while leftmark <= rightmark and \
                alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1

        while alist[rightmark] >= pivotvalue and \
                rightmark >= leftmark:
            rightmark = rightmark -1

# Continued on next slide...
```

Quick Sort

Implementation (cont.)

```
        if rightmark < leftmark:
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark] = alist[rightmark]
            alist[rightmark] = temp

    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp

    return rightmark
```

Analysis

Analysis

- A bubble sort, a selection sort, and an insertion sort are $O(n^2)$ algorithms
- A shell sort improves on the insertion sort by sorting incremental sub-lists
  - It falls between O(n) and $O(n^2)$
- A merge sort is O(n log n), but requires additional space for the merging process

Analysis

- A quick sort is O(n log n), but may degrade to O(n^2) if the split points are not near the middle of the list
- It does not require additional space

Questions?