The 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 $n-1$ 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.

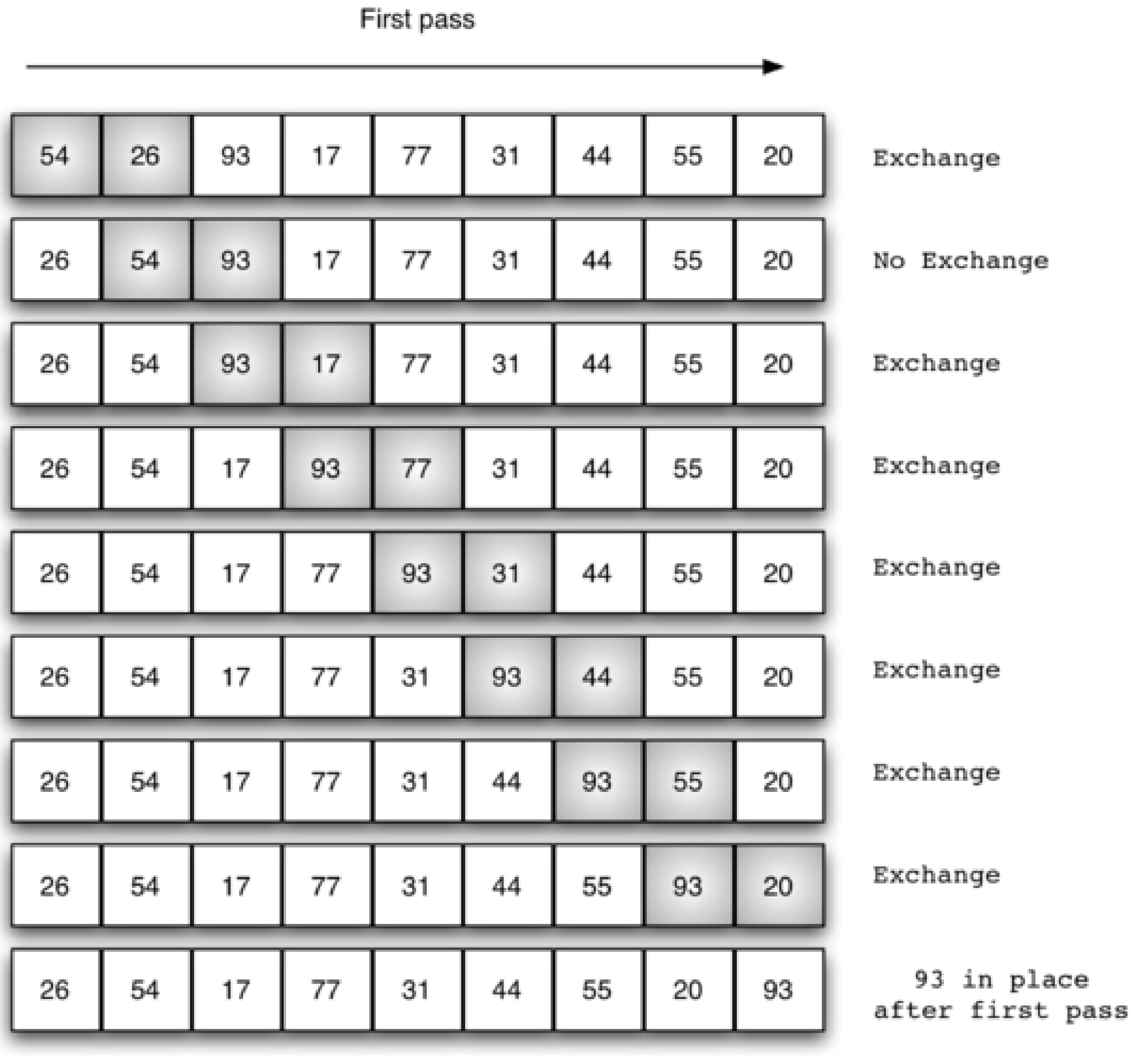
First pass
54 26 93 17 77 31 44 55 20 Exchange
26 54 93 17 77 31 44 55 20 No Exchange
26 54 93 17 77 31 44 55 20 Exchange
26 54 17 93 77 31 44 55 20 Exchange
26 54 17 77 93 31 44 55 20 Exchange
26 54 17 77 31 93 44 55 20 Exchange
26 54 17 77 31 44 93 55 20 Exchange
26 54 17 77 31 44 55 93 20 Exchange
26 54 17 77 31 44 55 20 93
93 in place after first pass

Figure 1: `bubbleSort` : The First Pass

.html)

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. 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 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.

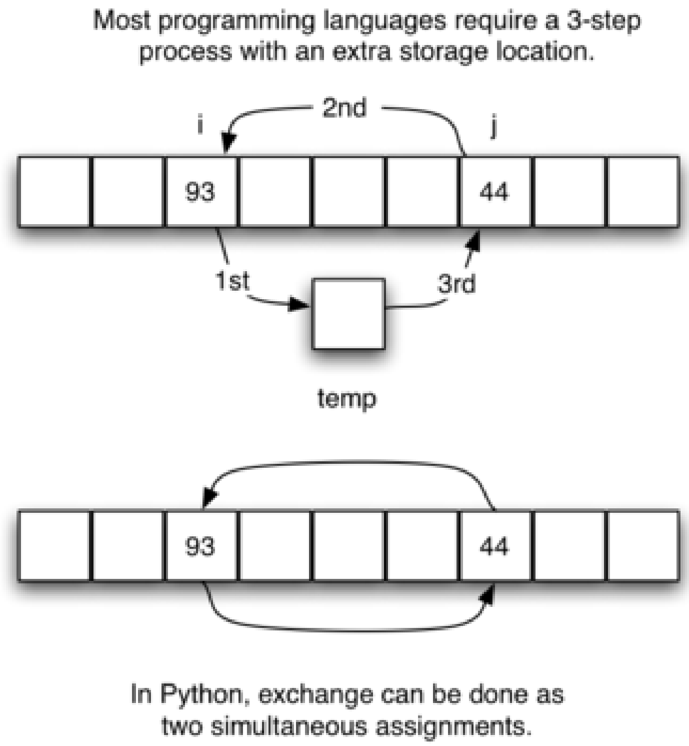
Most programming languages require a 3-step process with an extra storage location.
i
2nd
j
93
44
1st
3rd
temp
93
44
In Python, exchange can be done as two simultaneous assignments.

Figure 2: Exchanging Two Values in Python

The following activecode example shows the complete `bubbleSort` function working on the list shown above.

Run Save Load Show CodeLens

```
def bubbleSort(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

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)

```

ActiveCode: 1 The Bubble Sort (lst_bubble)

The following animation shows `bubbleSort` in action.

Initialize Run Stop

Beginning Step Forward Step Backward End

To analyze 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 1 shows the number of comparisons for each pass. The total number of comparisons is the sum of the first $n-1$ integers. Recall that the sum of the first *n* integers is $\frac{1}{2}n^2 + \frac{1}{2}n$. The sum of the first $n-1$ integers is $\frac{1}{2}n^2 + \frac{1}{2}n - n$, which is $\frac{1}{2}n^2 - \frac{1}{2}n$. This is still $O(n^2)$ comparisons. In the best case, if the list is already ordered, no exchanges will be made. However, in the worst case, every comparison will cause an exchange. On average, we exchange half of the time.

Table 1: Comparisons for Each Pass of Bubble Sort

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

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. In particular, 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. ActiveCode 2 shows this modification, which is often referred to as the **short bubble**.

Run Save Load Show CodeLens

```
def shortBubbleSort(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

alist=[20,30,40,90,50,60,70,80,100,110]
shortBubbleSort(alist)
print(alist)

```

ActiveCode: 2 The Short Bubble Sort (lst_shortbubble)

Self Check

Q-42: Suppose you have the following list of numbers to sort: <br> [19, 1, 9, 7, 3, 10, 13, 15, 8, 12] which list represents the partially sorted list after three complete passes of bubble sort?

- ◯ [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]
- ◯ [1, 3, 7, 9, 10, 8, 12, 13, 15, 19]
- ◯ [1, 7, 3, 9, 10, 13, 8, 12, 15, 19]
- ◯ [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]

Check Me Compare me

❮ (sorting.html) ❯ (TheSelectionSort.html)

user not logged in