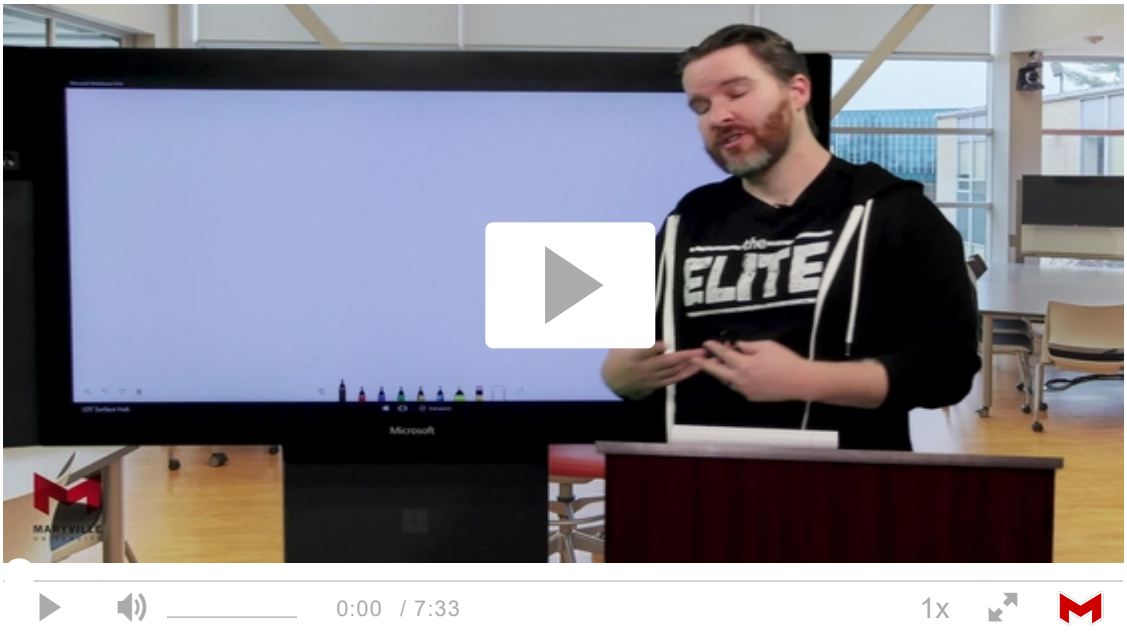
# Sorting

Let's begin by waching the video "Sorting" (7:33).




Sorting is the process of placing elements from a collection in some kind of order. For example, a list of words could be sorted alphabetically or by length. A list of cities could be sorted by population, by area, or by zip code. We have already seen a number of algorithms that were able to benefit from having a sorted list (recall the final anagram example and the binary search).

There are many, many sorting algorithms that have been developed and analyzed. This suggests that sorting is an important area of study in computer science. Sorting a large number of items can take a substantial amount of computing resources. Like searching, 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. On the other hand, for larger collections, we want to take advantage of as many improvements as possible. In this section we will discuss several sorting techniques and compare them with respect to their running time.

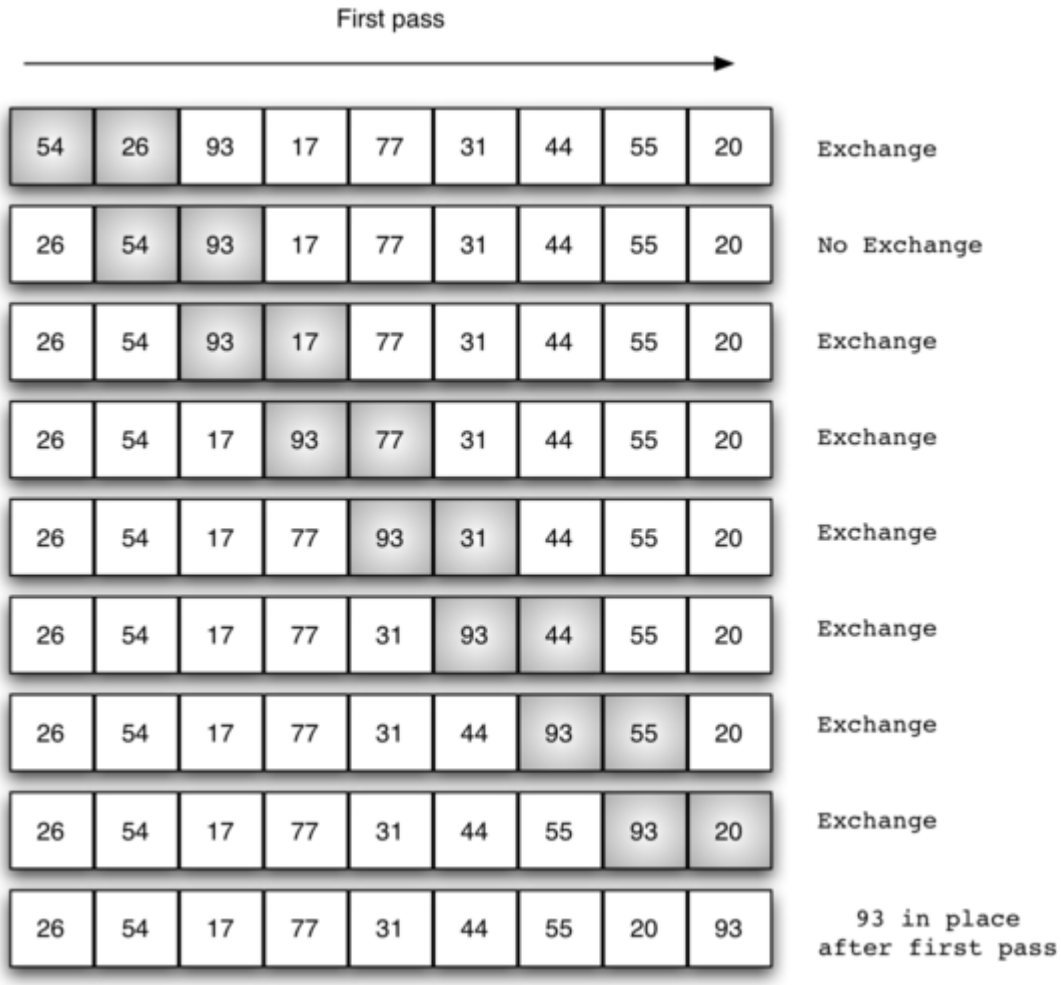
Before getting into specific algorithms, we should think about the operations that can be used to analyze a sorting process. First, it will be necessary to compare two values to see which is smaller (or larger). In order to sort a collection, it will be necessary to have some systematic way to compare 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 Sorting

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.

The image below 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  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  items left to sort, meaning that there will be  pairs. Since each pass places the next largest value in place, the total number of passes necessary will be . After completing the  passes, the smallest item must be in the correct position with no further processing required. The code below shows the

complete `bubbleSort` function. It takes the list as a parameter, and modifies it by exchanging items as necessary.

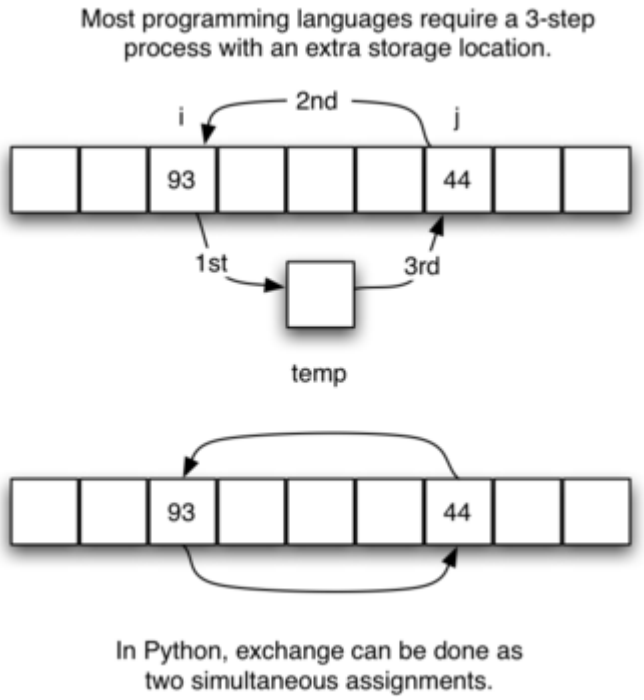
The exchange operation, sometimes called a “swap,” is slightly different in Python than in most other programming languages. Typically, swapping two elements in a list requires a temporary storage location (an additional memory location). A code fragment such as

```
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. Using simultaneous assignment, the exchange operation can be done in one statement.

Lines 5-7 perform the exchange of the  and  items using the three–step procedure described earlier. Note that we could also have used the simultaneous assignment to swap the items.



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

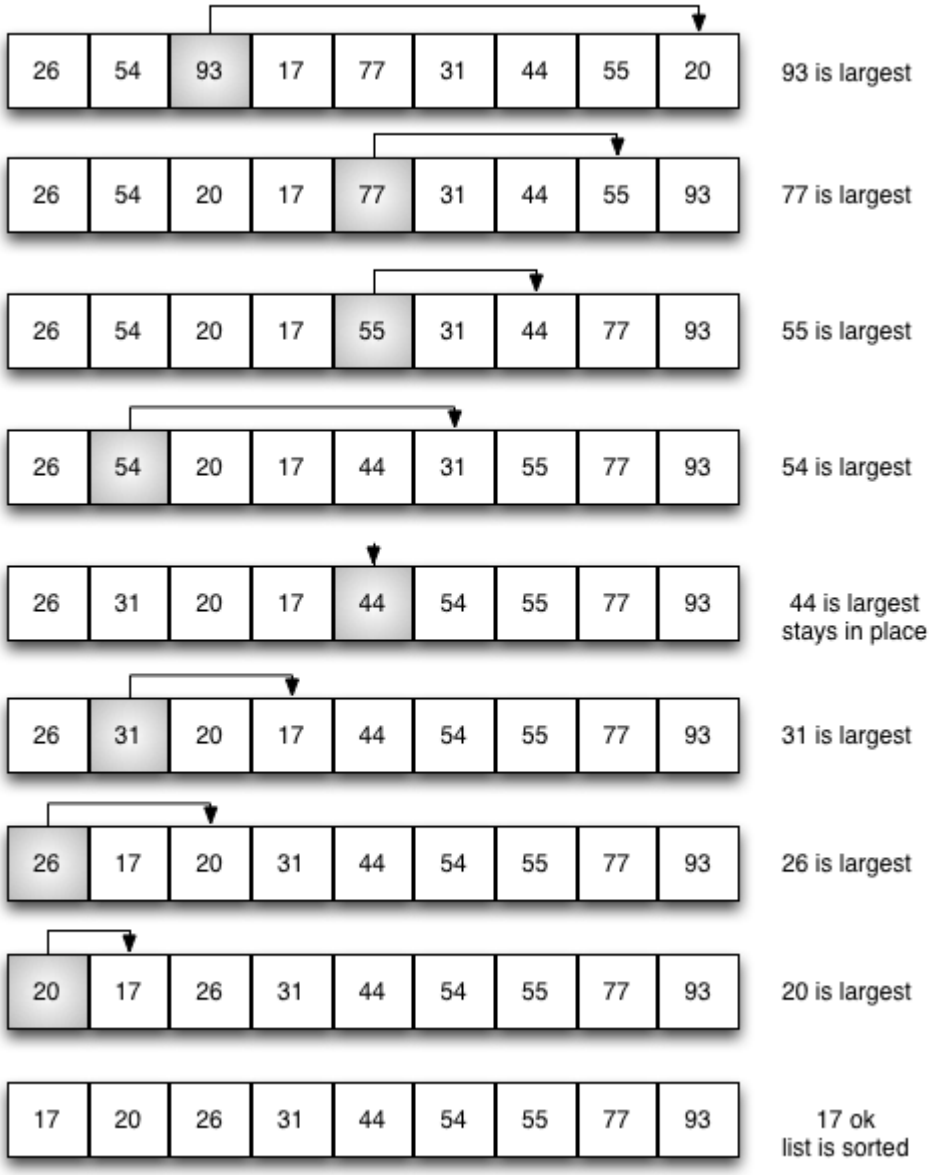
```
def bubbleSort(alist):
 for passnum in range(len(alist)-1,0,-1):
 for i in range(passnum):
 if alist[i]>alist[i+1]:
 temp = alist[i]
 alist[i] = alist[i+1]
 alist[i+1] = temp

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

## Selection Sort

The **selection sort** improves on the bubble sort by making only one exchange for every pass through the list. In order 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 passes to sort *n* items, since the final item must be in place after the  st pass.

The image below 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 shownbelow.



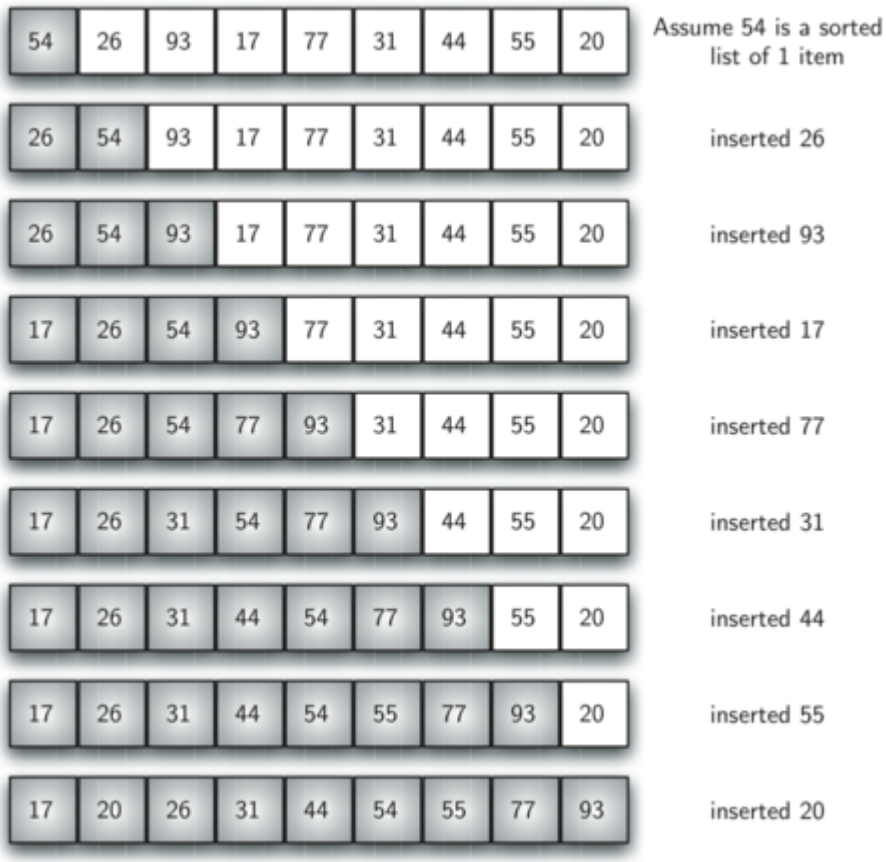

```
def selectionSort(alist):
 for fillslot in range(len(alist)-1,0,-1):
 positionOfMax=0
 for location in range(1,fillslot+1):
 if alist[location]>alist[positionOfMax]:
 positionOfMax = location

 temp = alist[fillslot]
 alist[fillslot] = alist[positionOfMax]
 alist[positionOfMax] = temp

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

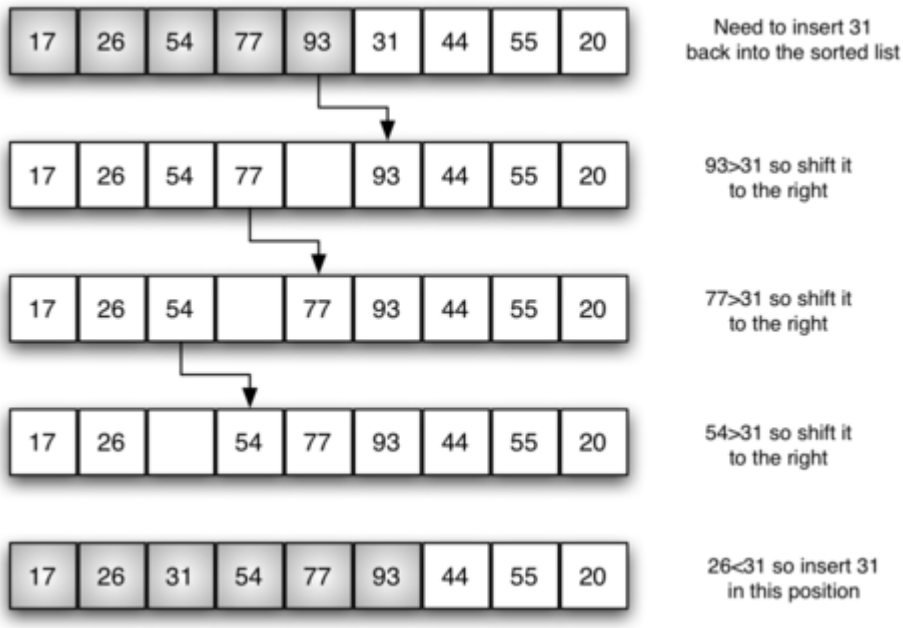
# Insertion Sort

The **insertion sort**, although still , 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. The figure below 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 ) is already sorted. On each pass, one for each item 1 through , 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.

The figure below 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 `insertionSort` shows that there are again passes to sort *n*items. The iteration starts at position 1 and moves through position , 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.

```
def insertionSort(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

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

The maximum number of comparisons for an insertion sort is the sum of the first integers. Again, this is . However, 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.

Source: **Problem Solving and Algorithms in Python** (http://interactivepython.org/runestone/static/pythonds/index.html#) from Bradley Miller on **www.interactivepython.org** (http://interactivepython.org/runestone/static/pythonds/index.html#).