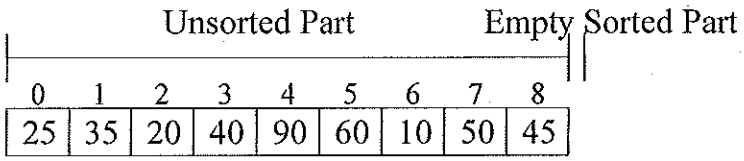
2. All *simple sorts* consist of two nested loops where:
- the **outer loop** keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
    - the **inner loop's** job is to do the work to extend the sorted part's size by one.

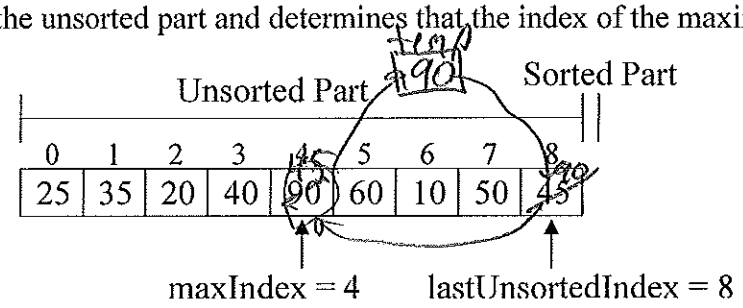
Initially, the sorted part is typically empty. The simple sorts differ in how their inner loops perform their job.

Selection sort is an example of a simple sort. Selection sort's inner loop scans the unsorted part of the list to find the maximum item. The maximum item in the unsorted part is then exchanged with the last unsorted item to extend the sorted part by one item.

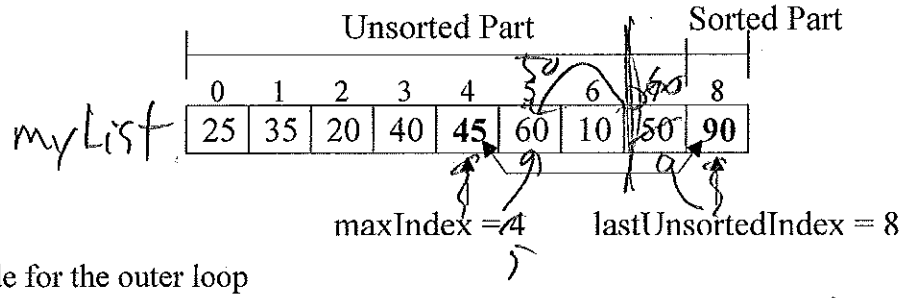
At the start of the first iteration of the outer loop, initial list is completely unsorted:



The inner loop scans the unsorted part and determines that the index of the maximum item, maxIndex = 4.



After the inner loop (but still inside the outer loop), the item at maxIndex is exchanged with the item at lastUnsortedIndex. Thus, extending the Sorted Part of the list by one item.



a) Write the code for the outer loop

```
for lastUnsortedIndex in range(len(myList)-1,0,-1):
```

b) Write the code for the inner loop to scan the unsorted part of the list to determine the index of the maximum item

```
maxIndex = 0
for testIndex in range(1, lastUnsortedIndex+1):
    if myList[maxIndex] < myList[testIndex]:
        maxIndex = testIndex
```

c) Write the code to exchange the list items at positions maxIndex and lastUnsortedIndex.

```
temp = myList[maxIndex]
myList[maxIndex] = myList[lastUnsortedIndex]
myList[lastUnsortedIndex] = temp
```

5+5+5+5 moves O(n)
3(n-1) moves
(n-1) pairs / 2 comparisons O(n²)

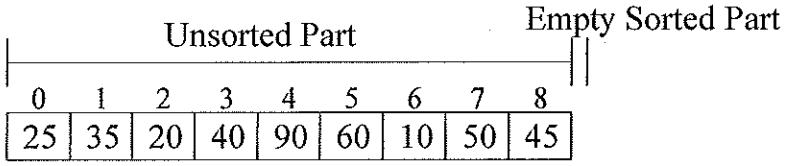
d) What is the big-oh notation for selection sort?

$n \equiv len(myList)$

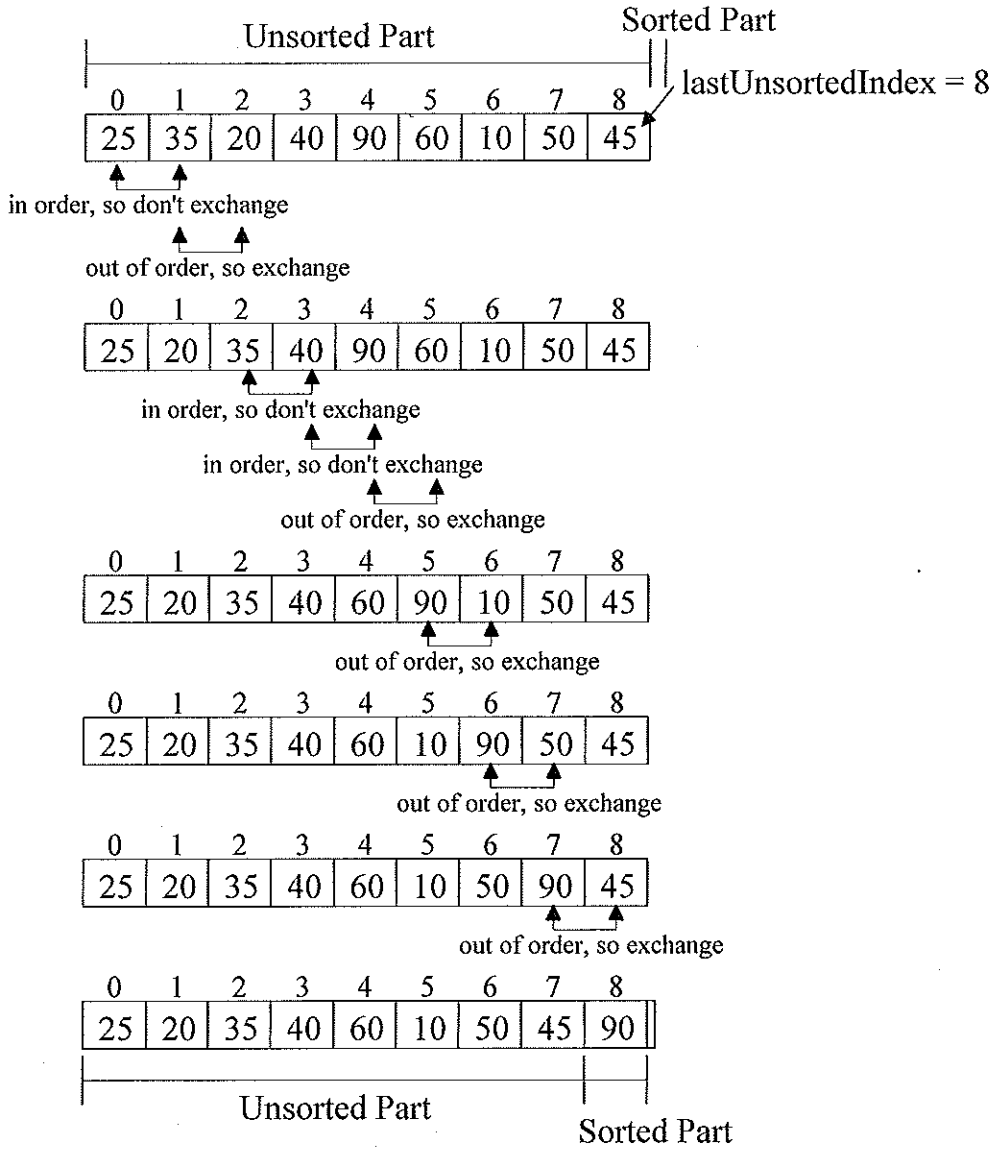
$= (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1 = n + n + \ldots + n = n\frac{(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$

3. *Bubble sort* is another example of a simple sort. Bubble sort's inner loop scans the unsorted part of the list comparing adjacent items. If it finds adjacent items out of order, then it exchanges them. This causes the largest item to "bubble" up to the "top" of the unsorted part of the list.

At the start of the first iteration of the outer loop, initial list is completely unsorted:



The inner loop scans the unsorted part by comparing adjacent items and exchanging them if out of order.



After the inner loop (but still inside the outer loop), there is nothing to do since the exchanges occurred inside the inner loop.

a) What would be the worst-case big-oh of bubble sort?

b) What would be true if we scanned the unsorted part and didn't need to do any exchanges?

Bubble Sort

```
for lastUnsortedIndex in range(len(myList)-1, 0, -1):
    exchanged = False
    for testIndex in range(0, lastUnsortedIndex):
        if myList[testIndex] > myList[testIndex+1]:
            temp = myList[testIndex]
            myList[testIndex] = myList[testIndex+1]
            myList[testIndex+1] = temp
            exchanged = True
    if not exchanged:
        break
```

compares $O(n^2)$

moves worst case $O(n^2)$

↑ initial arrangement is descending

best with exchanged flag

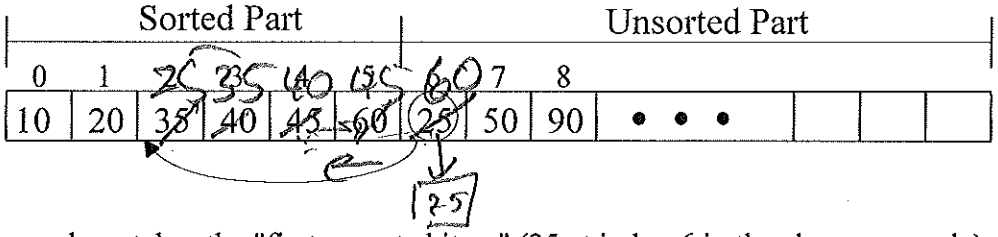
compares $O(n)$

moves $O(1)$

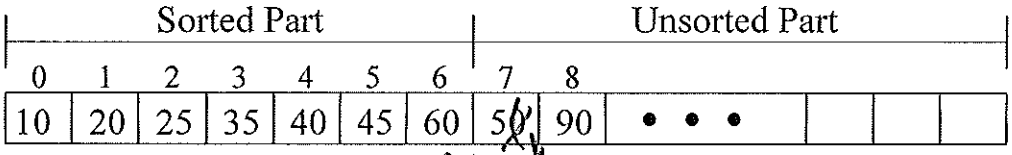
4. Another simple sort is called insertion sort. Recall that in a simple sort:

- the outer loop keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
  - the inner loop's job is to do the work to extend the sorted part's size by one.

After several iterations of insertion sort's outer loop, a list might look like:



In insertion sort the inner-loop takes the "first unsorted item" (25 at index 6 in the above example) and "inserts" it into the sorted part of the list "at the correct spot." After 25 is inserted into the sorted part, the list would look like:



Code for insertion is given below:

```
def insertionSort(myList):
    """Rearranges the items in myList so they are in ascending order"""

    for firstUnsortedIndex in range(1,len(myList)):
        itemToInsert = myList[firstUnsortedIndex]

        testIndex = firstUnsortedIndex - 1

        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1

        # Insert the itemToInsert at the correct spot
        myList[testIndex + 1] = itemToInsert
```

a) What is the purpose of the `testIndex >= 0` while-loop comparison?

have not run testIndex off left end of myList

b) What initial arrangement of items causes the is the overall worst-case performance of insertion sort?

descending order

c) What is the worst-case $O(\,)$ notation for the number of item moves?

$(1+2)+(2+2)+(3+2)+\ldots+(n-1+2) = 2\times(n-1)+n\frac{(n-1)}{2}$ $O(n^2)$

d) What is the worst-case $O(\,)$ notation for the number of item comparisons? $O(n^2)$

e) What initial arrangement of items causes the is the overall best-case performance of insertion sort?

$O(n)$ ascending order initial

f) What is the best-case $O(\,)$ notation for insertion sort?