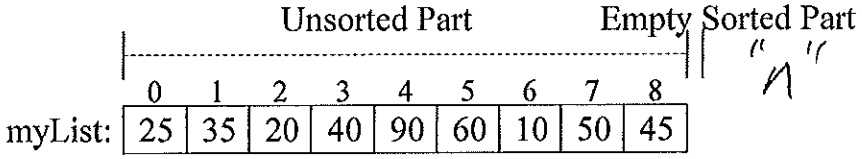
2. All *simple sorts* consist of two nested loops where:

- 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.

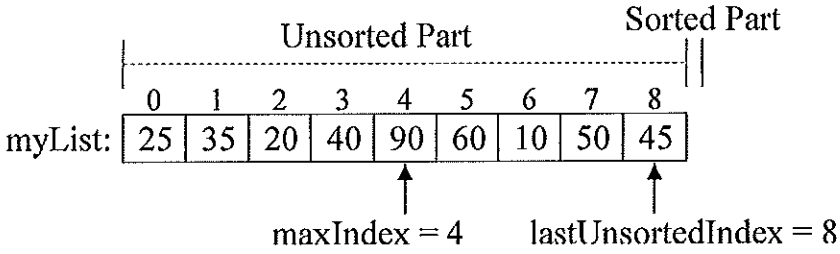
Initially, the sorted part is typically empty. The simple sorts differ in how their inner loops perform their job.

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.

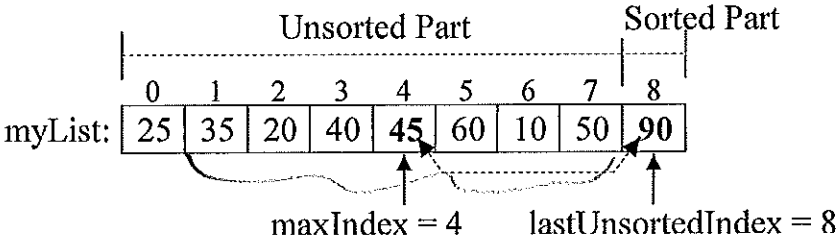
At the start of the first iteration of the outer loop, initial list is completely unsorted:



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, lastUnsortIndex+1, 1):
    if myList[testIndex] > myList[maxIndex]:
        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
```

O(n²)

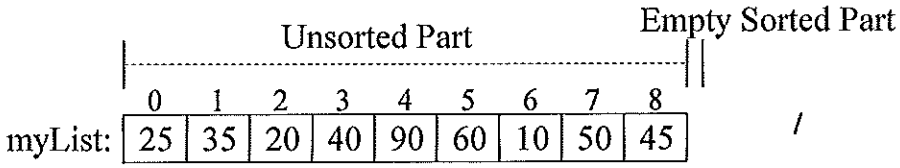
d) What is the big-oh notation for selection sort?

#compares = (n-1)+(n-2)+(n-3)+ ... +3+2+1 = n x (n-1)/2 = n²/2 - n/2

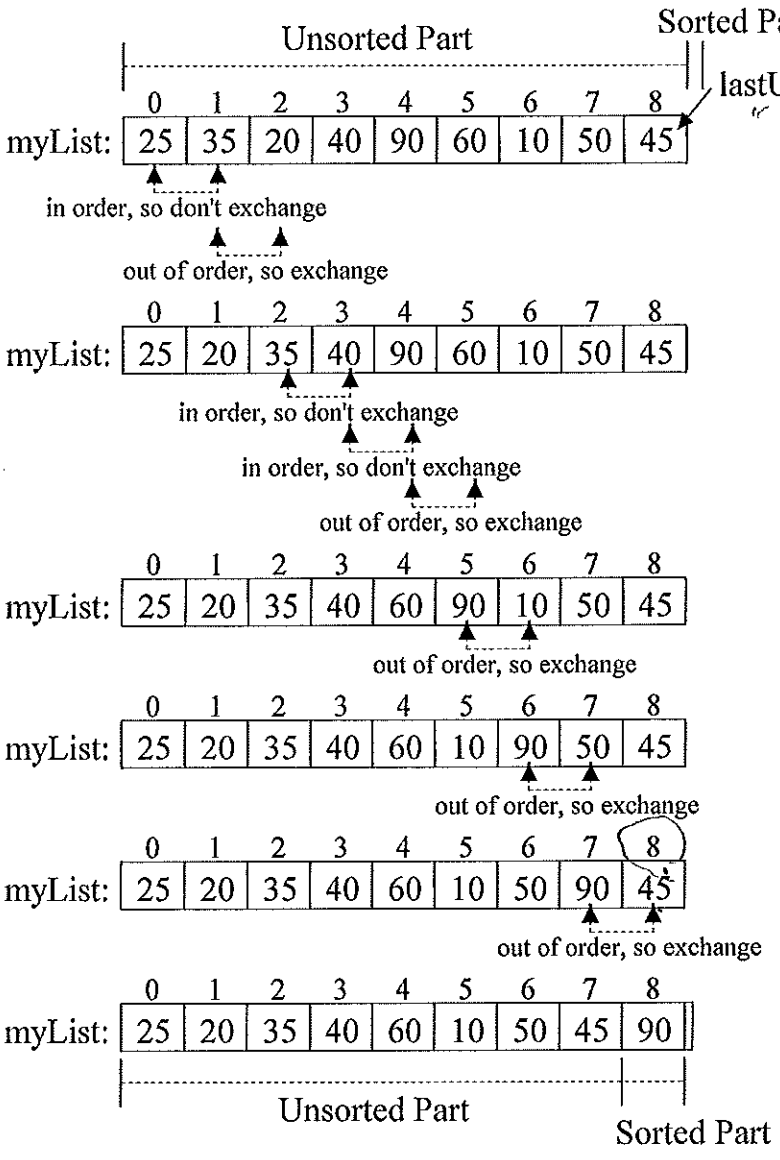
#moves = (n-1)x3 O(n)

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.

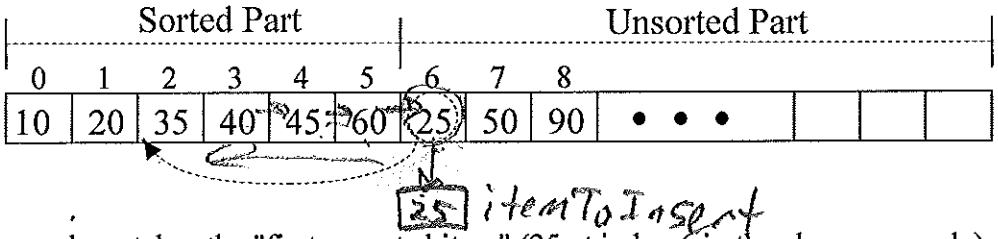
a) What would be the worst-case big-oh of bubble sort? $O(n^2)$

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

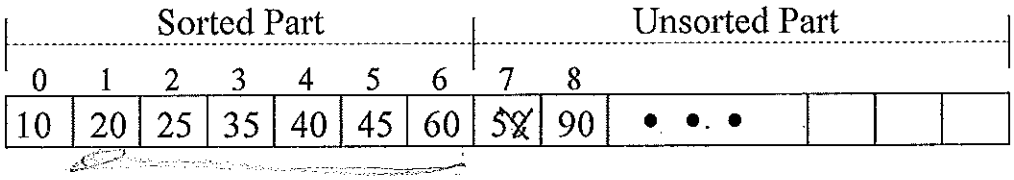
```
def bubbleSort(myList):
    for lastUnsortedIndex in range(len(myList)-1, 0, -1):
        alreadySorted = True
        for testIndex in range(0, lastUnsortedIndex, 1):
            if myList[testIndex] > myList[testIndex+1]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex+1]
                myList[testIndex+1] = temp
                alreadySorted = False
        if alreadySorted:
            return
```

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?

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

descending order initially

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

$O(n^2)$

d) What is the worst-case $O()$ notation for the number of item comparisons? $O(n2)$

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

ascending

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

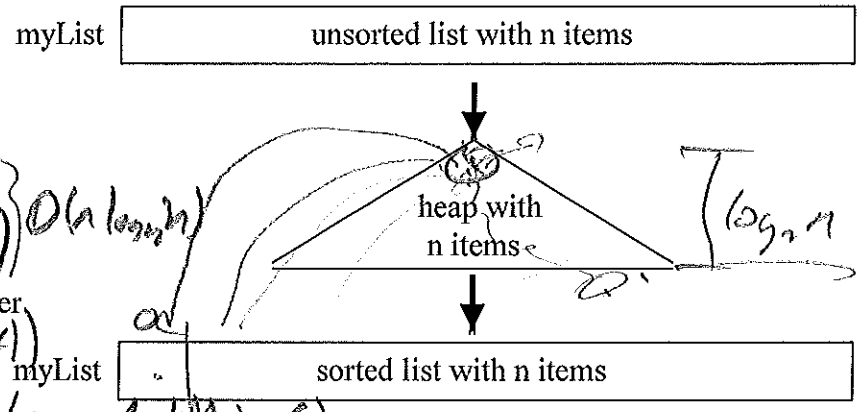
$O(n)$

1. So far, we have looked at simple sorts consisting of nested loops. The # of inner loop iterations n*(n-1)/2 is $O(n^2)$. Consider using a min-heap to sort a list. (methods: `BinHeap()`, `insert(item)`, `delMin()`, `isEmpty()`, `size()`)

a) If we insert all of the list elements into a min-heap, what would we easily be able to determine?

General idea of Heap sort:

1. Create an empty heap
2. Insert all n list items into heap
3. delMin heap items back to list in sorted order



O(1) myHeap = BinHeap()
O(1) for item in myList:
myHeap.insert(item) } O(n log n)
1x → for index in range(len(myList)):
myList[index] = myHeap.delMin()
log n
n

b) What is the overall $O(\,)$ for heap sort?

O(n log n)

2. Another way to do better than the simple sorts is to employ divide-and-conquer (e.g., Merge sort and Quick Sort). Recall the idea of **Divide-and-Conquer** algorithms. Solve a problem by:

- dividing problem into smaller problem(s) of the same kind
- solving the smaller problem(s) recursively
- use the solution(s) to the smaller problem(s) to solve the original problem

In general, a problem can be solved recursively if it can be broken down into smaller problems that are identical in structure to the original problem.

a) What determines the "size" of a sorting problem?

b) How might we break the original problem down into smaller problems that are identical?

c) What base case(s) (i.e., trival, non-recursive case(s)) might we encounter with recursive sorts?

d) How do you combine the answers to the smaller problems to solve the original sorting problem?

e) Consider why a recursive sort might be more efficient. Assume that I had a simple n^2 sorting algorithm with n = 100, then there is roughly $100^2 / 2$ or 5,000 amount of work. Suppose I split the problem down into two smaller sorting problems of size 50.

- If I run the n^2 algorithm on both smaller problems of size 50, then what would be the approximate amount of work?

- If I further solve the problems of size 50 by splitting each of them into two problems of size 25, then what would be the approximate amount of work?