1) Consider the following linear search (sequential search) code:

```
def linearSearch(target, aList):
    """Returns the index position of target in aList or -1 if target
    is not in aList"""
    for position in xrange(len(aList)):
        if target == aList[position]:
            return position
    return -1
```

a) What is the basic operation of a search?

b) For the following `aList` value, which `target` value causes linearSearch to loop the fewest ("best case") number of times?

	0	1	2	3	4	5	6	7	8	9	10
aList:	10	15	28	42	60	69	75	88	90	93	97

c) For the following `aList` value, which `target` value causes linearSearch to loop the most ("worst case") number of times?

d) For a *successful search* (i.e., `target` value in `aList`), what is the "average" number of loops?

```
def linearSearchOfSortedListA(target, aList):
    """Returns the index position of target in sorted aList or -1 if target
    is not in aList"""

    breakOut = False
    for position in xrange(len(aList)):
        if target <= aList[position]:
            breakOut = True
            break

    if not breakOut:
        return -1
    elif target == aList[position]:
        return position
    else:
        return -1
```

e) The above version of linear search assumes that `aList` is sorted in ascending order. When would this version perform better than the original `linearSearch` at the top of the page?

2) Consider the following binary search code:

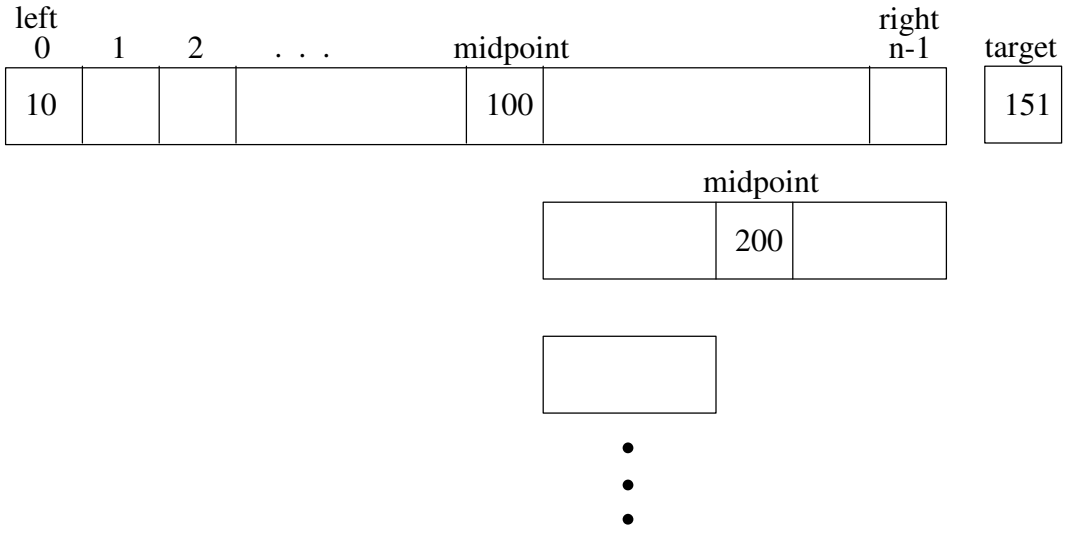
```
def binarySearch(target, lyst):
    """Returns the position of the target item if found, or -1 otherwise."""
    left = 0
    right = len(lyst) - 1
    while left <= right:
        midpoint = (left + right) / 2
        if target == lyst[midpoint]:
            return midpoint
        elif target < lyst[midpoint]:
            right = midpoint - 1
        else:
            left = midpoint + 1
    return -1
```

a) For binary search, what is the best-case time complexity $B(\ )$?

b) What is the basic operation for binary search?

c) "Trace" binary search to determine the total number of worst-case basic operations?

loop #	# of basic operations	worst-case # elements
1	2	"n"



d) If the list size is 1,000,000, then what is the maximun number of comparisons of list items on a *successful search*?

e) If the list size is 1,000,000, then how many comparisons would you expect on an *unsuccessful search*?

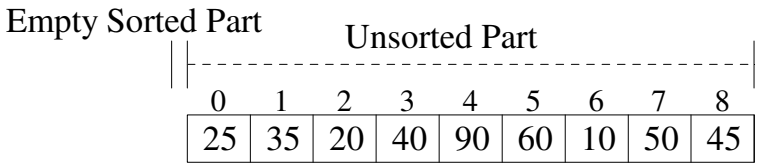
All *simple sort*s consist of two nested loops where:

- 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.

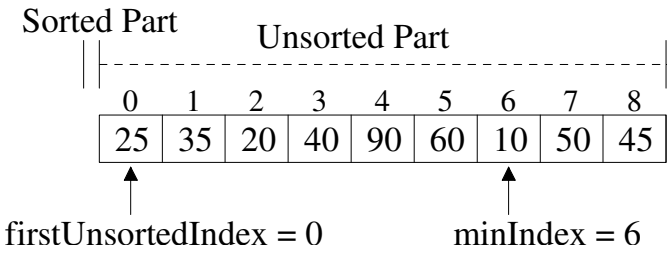
Initially, the sorted part is typically empty. The simple sorts differ in how their inner loops perform their job.

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

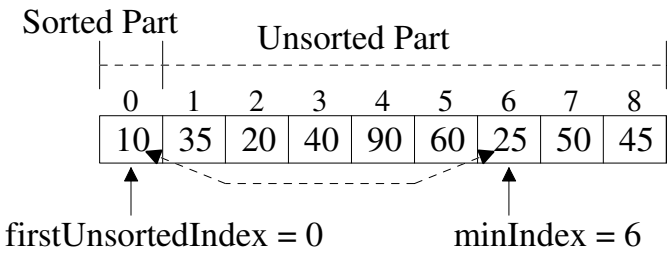
At the start of the first iteration of the outer loop, initial list is completely unsorted:



The inner loop scans the unsorted part and determines that the index of the minimum item, minIndex = 6.



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



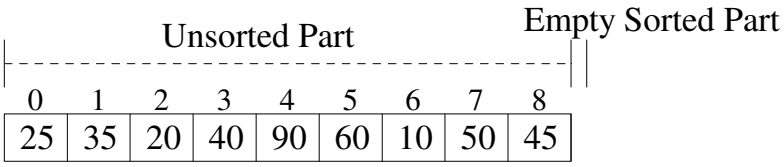
a) Write the code for the outer loop

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

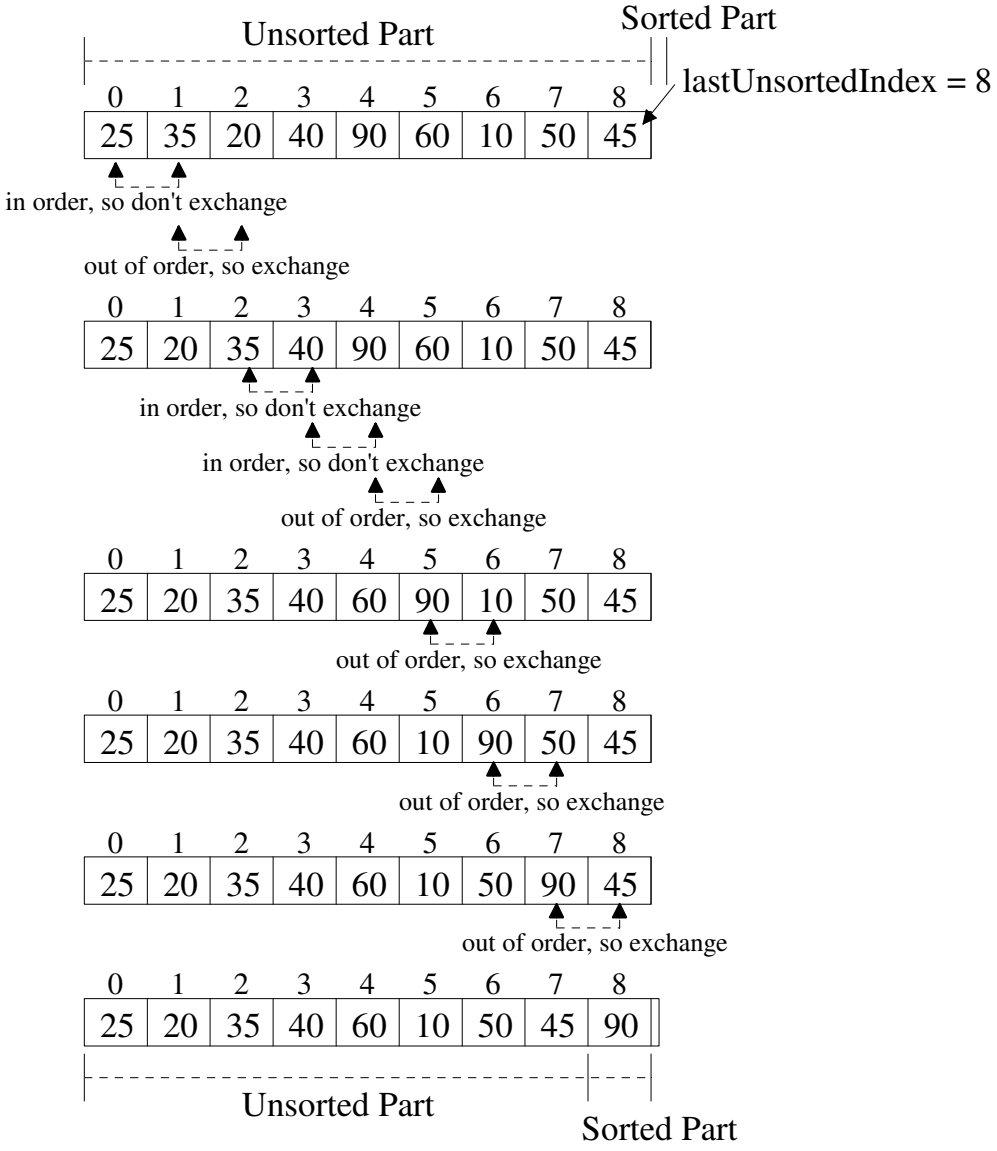
c) Write the code to exchange the list items at positions firstUnsortedIndex and minIndex.

4) *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 complexity of bubble sort?

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