1. The Dictionary implementation using open-address hashing was the `OpenAddrHashDict` class in `lab7.zip`.

```
from entry import Entry
class OpenAddrHashDict(object):
    EMPTY = None  # class variables shared by all objects of the class
    DELETED = True

    def __init__(self, capacity = 8, hashFunction = hash,
                 linear = True):
        self._table = [OpenAddrHashDict.EMPTY] * capacity
        self._size = 0
        self._hash = hashFunction
        self._homeIndex = -1
        self._actualIndex = -1
        self._linear = linear
        self._probeCount = 0

    def __getitem__(self, key):
        """Returns the value associated with key or
        returns None if key does not exist."""
        if key in self:
            return self._table[self._actualIndex].getValue()
        else:
            return None

    def __delitem__(self, key):
        """Removes the entry associated with key."""
        if key in self:
            self._table[self._actualIndex] = OpenAddrHashDict.DELETED
            self._size -= 1

    def __setitem__(self, key, value):
        """Inserts an entry with key/value if key does not exist or
           replaces the existing value with value if key exists."""
        entry = Entry(key, value)
        if key in self:
            self._table[self._actualIndex] = entry
        else:
            self._table[self._actualIndex] = entry
            self._size += 1

    def __contains__(self, key):
        """Return True if key is in the dictionary; return False otherwise"""
        entry = Entry(key, None)
        self._probeCount = 0
        # Get the home index
        self._homeIndex = abs(self._hash(key)) % len(self._table)
        rehashAttempt = 0
        index = self._homeIndex

        # Stop searching when an empty cell is encountered
        while rehashAttempt < len(self._table):
            self._probeCount += 1
            if self._table[index] == OpenAddrHashDict.EMPTY:
                self._actualIndex = index
                return False # An empty cell is found, so key not found
            elif self._table[index] == entry:
                self._actualIndex = index
                return True

            # Calculate the index and wrap around to first position if necessary
            rehashAttempt += 1
            if self._linear:
                index = (self._homeIndex + rehashAttempt) % len(self._table)
            else:  # Quadratic probing
                index = (self._homeIndex + (rehashAttempt ** 2+ rehashAttempt) // 2) % len(self._table)

        return False   # tried all the slots in the hash table and did not find key

    def __len__(self):
        return self._size

    def __str__(self):
        resultStr = "{"
        for item in self._table:
            if not item in (OpenAddrHashDict.EMPTY,OpenAddrHashDict.DELETED):
                resultStr = resultStr + " " + str(item)
        return resultStr + " }"

    def __iter__(self):
        """Iterates over the keys of the dictionary"""
        for item in self._table:
            if isinstance(item, Entry):
                yield item.getKey()
        raise StopIteration
```

a) Complete the `__iter__` method.

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:

Unsorted Part | Empty Sorted Part

	0	1	2	3	4	5	6	7	8
myList:	25	35	20	40	90	60	10	50	45

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

Unsorted Part | Sorted Part

	0	1	2	3	4	5	6	7	8
myList:	25	35	20	40	90	60	10	50	45

maxIndex = 4 lastUnsortedIndex = 8

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.

Unsorted Part | Sorted Part

	0	1	2	3	4	5	6	7	8
myList:	25	35	20	40	**45**	~~60~~ 50	10	~~50~~ 60	**90**

maxIndex = ~~4~~ 5 lastUnsortedIndex = ~~8~~ 7

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

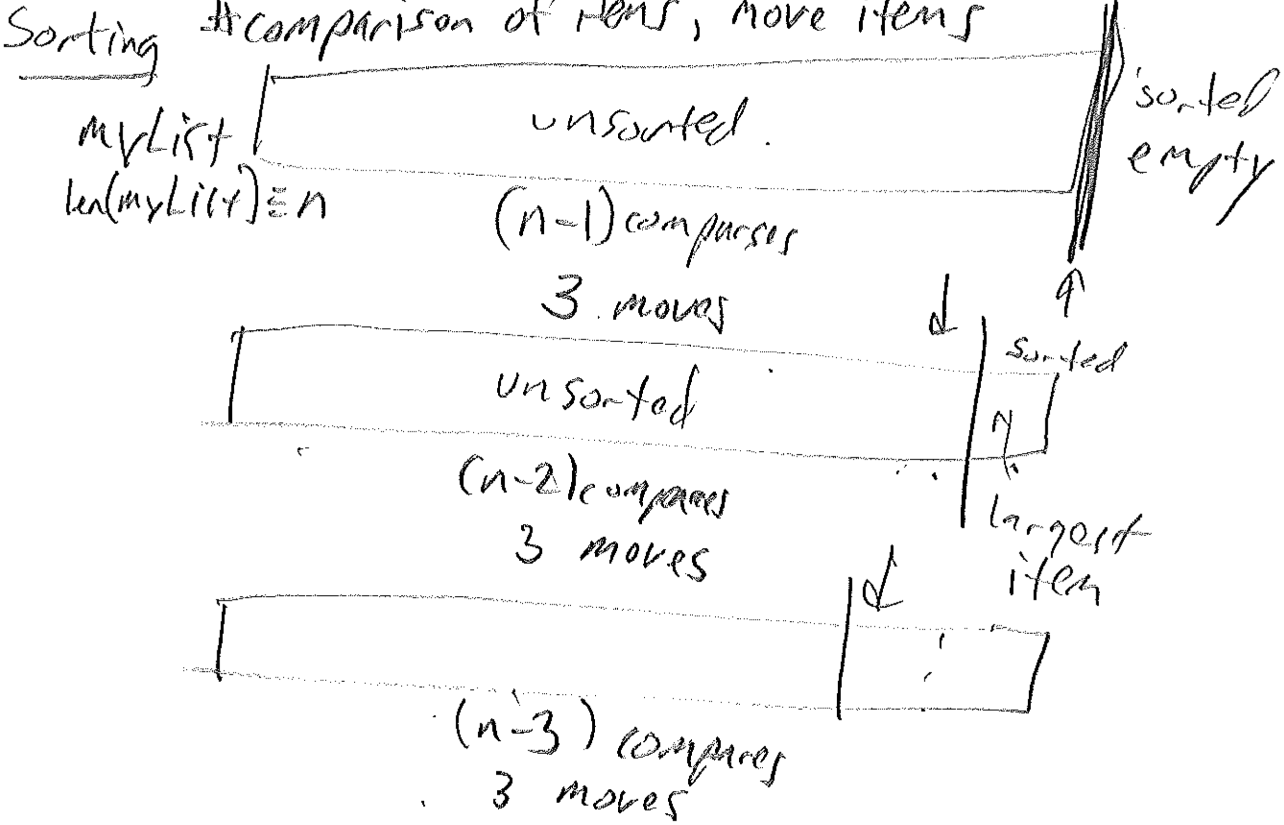
```
maxIndex = 0
for testIndex in range(1, lastUnsortedIndex+1, 1):
    if myList[maxIndex] < myList[testIndex]:
        maxIndex = testIndex
```

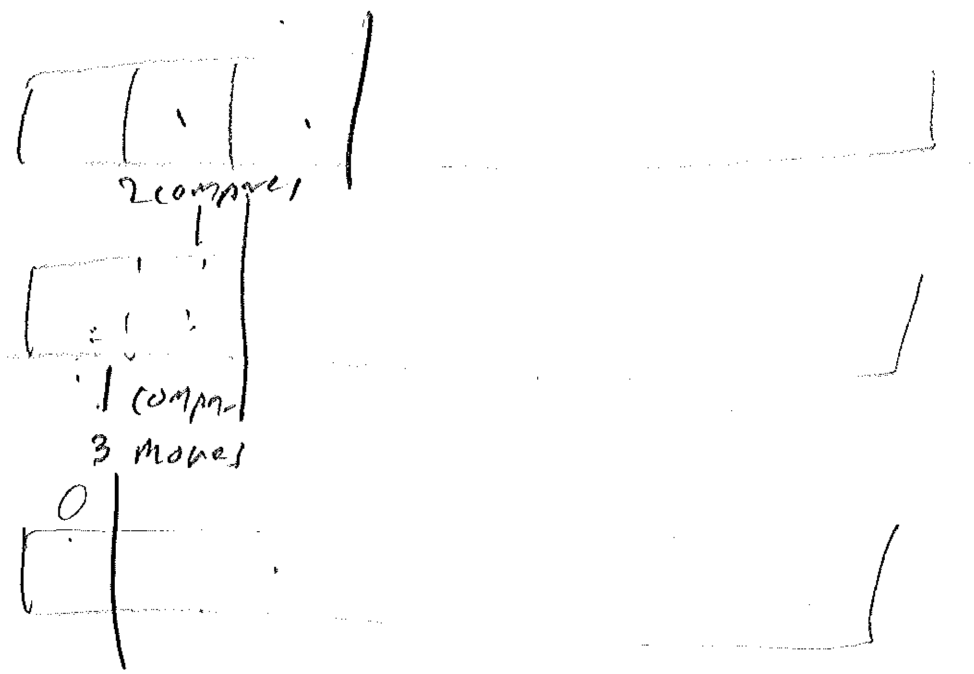
c) Write the code to exchange the list items at positions maxIndex and lastUnsortedIndex.

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

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

Sorting #comparison of items, move items





#moves

$3 \times (n-1)$

$O(n)$

$$\#\text{compares} = \underbrace{(n-1) + \underbrace{(n-2) + \underbrace{(n-3) + \cdots + 3}_{n} + 2}_{n} + 1}_{n}$$

$$= n + n + n + \cdots + n = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \quad O(n^2)$$

Bubble sort

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

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:

Unsorted Part | Empty Sorted Part

	0	1	2	3	4	5	6	7	8
myList:	25	35	20	40	90	60	10	50	45

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

Unsorted Part | Sorted Part

lastUnsortedIndex = 8

	0	1	2	3	4	5	6	7	8
myList:	25	35	20	40	90	60	10	50	45

in order, so don't exchange

out of order, so exchange

	0	1	2	3	4	5	6	7	8
myList:	25	20	35	40	90	60	10	50	45

in order, so don't exchange

in order, so don't exchange

out of order, so exchange

	0	1	2	3	4	5	6	7	8
myList:	25	20	35	40	60	90	10	50	45

out of order, so exchange

	0	1	2	3	4	5	6	7	8
myList:	25	20	35	40	60	10	90	50	45

out of order, so exchange

	0	1	2	3	4	5	6	7	8
myList:	25	20	35	40	60	10	50	90	45

50 45 60 out of order, so exchange

	20 0	25 1	2	3	10 4	60 5	60 6	7	8
myList:	~~25~~	~~20~~	35	40	~~60~~	~~10~~	~~50~~	45	90

Unsorted Part | Sorted Part

20 25 35 40 10 50 45 | 60 90

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?

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:

Sorted Part						Unsorted Part					
0	1	2	3	4	5	6	7	8			
10	20	35	40	45	60	25	50	90	• • •		

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:

Sorted Part							Unsorted Part				
0	1	2	3	4	5	6	7	8			
10	20	25	35	40	45	60	50	90	• • •		

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?

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

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

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

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