Implementation of SortedList ADT using Python list

In this lab you will be implementing the `SortedList` ADT using Python list . In particular, you will be using a generic comparison for searching and sorting algorithms.

The objectives of this lab are:

1. Implementing SortedList using Python list
2. Understanding the binary search and mergesort algorithms
3. Using generic comparison
4. Profiling code with counters

Part 1 (in-lab)

In this part you are required to create a class `SortedList` with its `__init__` method that takes a `list` as an argument. The default value for the list is `[]`.

`__init__` will call the `sort` method to sort the list, and will then store it as a class attribute `_L`. An implementation of `selectionSort` will be provided to you to call here.

Note: In this write-up, we give examples where that the items are integers; however, your code cannot make this assumption. We may use ANYTHING to test your `SortedList`, such as strings or tuples or even other lists.

Since we do not know what kind of items will be in the list, your code can NEVER make comparisons directly. It must call the `_compare` method of the `SortedList`. Originally it will be defined as follows:

```
def _compare(self, x, y):
    if (x < y):
        return -1
    elif (x == y)
        return 0
    elif (x > y)
        return 1
```

Please rewrite the `selectionSort` method so it calls this `_compare` method, and never makes comparisons directly!

However, sometimes we want to compare the items using a different criterion than <. For example, we can think about arranging the numbers in the list not according to their values, but rather according to the sum of the digits of each number. Here is an example to explain the difference: If the numbers are 34, 123, 9, 26, then according to their values our list will look like 9, 26, 34, 123 , but according to the second criterion our list will look like 123, 34, 26, 9.

To enable this flexibility we need to define an attribute of `SortedList` called `cmp` , with default value of `None` . The `cmp` attribute will be passed as a second argument to `__init__` , which will set `_cmp` to point to this function:

```
def __init__(self, L = [], cmpIn = None):
    # set self._L to L, self._cmp to cmpIn
```

This attribute will point to a function that does the comparison of list items for us. In particular, this function is supposed to receive two items, and compare them according to the required criterion. The function returns `0` if the items are equal, `-1` if the first item is less than the second, `1` if the first item is greater than the first.

You will be provided with a comparison function that you can use here.

Now, remember to fix your `_compare` method! If `_cmp` is not `None` , your `_compare` method should call it instead of using `<` , `>` , `==` .

Finally, let's do a quick implementation of `__str__` ; it will just return `str(_L)` .

Here is an example of a code snippet with this part:

```
sl = SortedList([26, 9, 34], cmpBySum)
print(sl) # prints [34, 26, 9]
```

In this case, the `cmpBySum` function will be written outside the class and it will be passed as an argument to the `setComparison` method.

The final issue that you have to deal with is counting the number of comparisons. To do so you have to define an appropriate attribute `ctComparisons` , initialized to `0` . Every time you make a comparison, the `_compare` has to increment this attribute. That is, you just increment this attribute within the compare method. This allows us to profile your code, to see how many comparisons your sort is making. Such profiling of methods is common practice.

Notice that you have to go back and initialize the counting attribute, `ctComparisons` , to zero at

the beginning of `selectionSort` , so as to know the number of comparisons for the sort.

```
sl = SortedList([26, 9, 34])
print(sl) # prints [9, 26, 34]
print(sl.ctComparisons)  # prints the count of how many times
# selectionSort compared items to sort [26, 9, 34]
```

Part 2:

Since our list may start out empty, we need an `add` method to add elements to it one at a time.

Now you have to implement the `add` method that receives an item and adds it to the list while keeping it sorted. To implement the `add` method, you need to compare the new item with the existing items in the list in order to find new item's place in the list.

Remember to call `_compare` whenever you need to compare the list items!

We may also want to change the `cmp` attribute that our `SortedList` uses. The `cmp` attribute can be reset by calling the `setComparison` method. The `setComparison` method will receive an external function as a parameter and set `cmp` to point to this function. Here is an example of a code snippet with this part:

```
sl = SortedList()
sl.setComparison(cmpBySum)
sl.add(26)
sl.add(9)
sl.add(34)
print(sl) # prints [34, 26, 9]
```

Note: When you call `setComparison` , your list may already have items in it. And they may be sorted in a different order from the one you are setting! So don't forget to re-sort your list as part of the `setComparison` method! It may no longer be sorted the same as before!

Part 3:

Implement the `__contains__` magic method using the binary search algorithm for searching the sorted list. Again, in this implementation you have to use the `_compare` method to enable generic comparison. Notice that you have to initialize the counting attribute, `ctComparisons` , to zero at the beginning of this method to so as to know the number of comparisons for a

given search.

```
sl = SortedList([6, 33, 11, 45, 8, 1, 34])
test = 11 in sl # test is set to True
print(sl.ctComparisons)
# there should be just 1 comparison, since 11 is in the middle of list
```

Part 4

Mergesort is a classic sorting algorithm which takes a divide-and-conquer approach. It is implemented recursively, where the function takes a list as input, divides the list into two halves, then calls itself recursively for the two halves. Once the smaller portions of the list cannot be divided any further, the algorithm builds the list back up again, in sorted form, by combining the sorted pieces. So, given a list of length `n`, the asymptotic running time of `mergeSort` is `O(n log n)`.

In this part, you will need to complete the `mergeSort` method as well as its `merge(A, B, L)` helper function, where `L` is the current list to be sorted and `A` and `B` are two smaller sub-lists.

`mergeSort(L)`

First, we will implement the base case. In this case, if the size of the list is one or empty, this list is already sorted and we can return.

To divide, we simply want to create two new lists, `A` and `B`. `A` will contain the first half of the elements in `_L`, and `B` will contain the second half of `_L`.

To conquer, or solve the original problem using much smaller sub-problems, we recursively call `mergeSort` twice with the new lists `A` and `B` as the inputs. Once we have done this, we will call the `merge` method in order to combine the two lists `A` and `B`.

`merge(A, B, L)`

Now, we will move to the `merge(A, B, L)` method. This is a helper function for the main `mergeSort` method, and this is where the actual merging of two lists will be done.

We want to iterate over the two lists `A` and `B`, each time comparing two elements, one from each list. Like before, use the generic compare method to perform the comparison. This also keeps track of the number of comparisons. The third parameter, `L`, is the new, combined list

that we are about to build up.

We will need two index variables for each of the lists `A` and `B`, say `i` and `j`. With each comparison, we will add the smaller element to `L` in order to end up with a list that is ultimately sorted from smallest to largest. If the elements are equal, it does not matter in which order they are added to the list. If you reach the end of one list but there are still remaining elements in the other list, simply add all remaining elements into `L`. Note that the input lists `A` and `B` are already sorted. The `merge` method merely combines two sorted sub-lists.

```
A           2    3   5
B           1    4   5
index       0    1   2
```

For example, let's look at the sample lists in the table above. We will begin by comparing `A[0]` and `B[0]`. `1 < 2`, and so we have `L = [1]`. We will now compare `A[0]` and `B[1]`. `2 < 4`, and so we will add `4` to the list to get `L = [1, 2]`. Next, we compare `A[1]` and `B[1]`; `3 < 4`, and so we have `L = [1, 2, 3]`. We continue this same process to ultimately result in `L = [1, 2, 3, 4, 5, 5]`. We return the length of `L`, which in this case would be `6`.

Part 5

Note that there is a significant difference in the value of `ct.comparison` for the merge sort and the selection sort algorithms for the same list. This is because merge sort runs in `O(n log n)` and selection sort takes `O(n^2)`. Look at the following examples.

```
L = [9, 8, 7, 6]
sl = SortedList(L)
print(sl.ctComparisons)        # prints 6 for selectionSort
sl.mergeSort(L)
print(sl.ctComparisons)        # prints 4 for mergeSort

L = [i for i in range(500)]
sl = SortedList(L)
print(sl.ctComparisons)        # prints 124750 for selectionSort
sl.mergeSort(L)
print(sl.ctComparisons)        # prints 2272 for mergeSort
```

For smaller lists, the difference isn't much. However, as `n` gets larger, the difference is highly significant. Write a test function, `testPerformance`, that takes size `n` as a parameter. It should create a list of that size of random numbers. Run the two different sort methods for the the same list and print the value of `ctComparisons`. Try this for different values of `n` (`10`,

`100`, `1000`, `10000`, `100000` and `1000000`). You can generate random numbers in the following way:

```
from random import randint

# randint takes 2 arguments - start and end
L = [randint(0, 50000) for i in range(n)]
```