National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2018/2019

Mission 9
Writing a Sorting Spell

Release date: 10 October 2018
Due: 18 October 2018, 23:59

Required Files

- mission09-template.py
- shelf.py

Background

"Finally, I'm home! I am never going to take trains again. And where did those magic-immune trains appear from anyway?!" you say to yourself.

Now that you're home, you start unpacking your bags to do your laundry. After the clothes are dried, you notice that everything is a mess and wondered if there is a way to sort all these clothes and get them organized.

"How am I going to sort them? I can't remember those long incantations to sort things out!" you exclaim.

"No, hold on, I may not remember the steps, but I remember the general idea behind sorting. Maybe, I can write my own sorting spell!"

Administrivia

The template file `mission09-template.py` has been provided for you to write your answers in. You will also require `shelf.py` which is imported in the template file.

Sample input and output have been provided in the template file. In addition to these samples, you should test your functions with your own test cases.

This mission will guide you through the basics of **insertion sort**. So as not to undermine your learning, you are barred from using Python built-in sorting functions like `sorted` and `.sort` in this mission.

This mission consists of **four** tasks.

Task 1: Searching for the Right Spot (8 marks)

There are many ways to sort a list. The most common and intuitive way shares the same approach as how people sort a hand of playing cards.

To sort a hand of cards, we look at each card, remove and place it in a sorted portion of the hand. This sorted portion increases incrementally from left to right every time we remove a card from the unsorted portion and insert the card into the sorted portion. When all of the cards are placed in the sorted portion, we are done.

To illustrate, suppose you have an unsorted list as shown below.

4	5	7	2	6

First, you take the first element and insert it into the sorted list which is initially empty.

4	5	7	2	6

4

You then move to the second element and insert it into the sorted list.

4	**5**	7	2	6

4	**5**

When you finish inserting the last element, you get a complete sorted list.

4	5	**7**	2	6

4	5	**7**

4	5	7	**2**	6

2	4	5	7

4	5	7	2	**6**

2	4	5	**6**	7

In fact, we are doing these steps:

1. Pick the next number in the list. (If this is the first iteration, we pick the first number.)
2. Search for the position to insert the number.
3. Insert the number into the position that we found in step 2.
4. If there is still a number in the list, go back to step 1. Otherwise, we're done!

In this task, we will write this algorithm step by step starting with the search function.

(a) Write a `search` function that takes in a value **x** and a sorted sequence (either a tuple or a list) and returns the **position** that the value should go to by iterating through the elements of the sequence starting from the first element. The position that **x** should go to in the list should be the first position such that it will be less than or equal to the next element in the list. **(4 marks)**

Here are some examples.

```
>>> search(-5, (1, 5, 10))
0
>>> search(3, (1, 5, 10))
1
```

```
>>> search(7, [1, 5, 10])  # Note that the sequence could be a list!
2
>>> search(5, (1, 5, 10))
1
>>> search(42, [1, 5, 10])
3
>>> search(42, (-5, 1, 3, 5, 7, 10))
6
```

(b) Write a function `binary_search` that uses an algorithm with $\Theta(\log n)$ time complexity to find the position to insert the element. Note that a solution with slicing operations is of $\Theta(n)$ time complexity, and not $\Theta(\log n)$. Solutions that use the $\Theta(n)$ solution will not get the full marks. **(4 marks)**

```
>>> binary_search(-5, (1, 5, 10))
0
>>> binary_search(3, (1, 5, 10))
1
>>> binary_search(7, [1, 5, 10])  # Note that the sequence could be a list!
2
>>> binary_search(5, (1, 5, 10))
1
>>> binary_search(42, [1, 5, 10])
3
>>> binary_search(42, (-5, 1, 3, 5, 7, 10))
6
```

Task 2: The Insertion (11 marks)

Now that we have the `search` function, we can insert an element into the sequence with the position found by `search`.

(a) Write an `insert_list` function that takes in an element and a **list** and inserts the element into the list at the first position where it is less than or equal to the next element. The function returns the resulting list. **(4 marks)**

Note: You should be inserting the item into *the input list itself*, instead of creating a new list that contains the item.

```
>>> insert_list(2, [1, 5, 9])
[1, 2, 5, 9]
>>> insert_list(10, [1, 5, 9])
[1, 5, 9, 10]
>>> insert_list(5, [2, 6, 8])
[2, 5, 6, 8]
```

To insert an element into a list, you could use the `list.insert(index, element)` function. This will insert `element` into the list at the position `index`. Here's a simple sequence of executions to illustrate how to use this function.

```
>>> lst = [5, 4, 1, 10]
>>> lst.insert(0, 1) # note that list.insert does not return anything
>>> lst
[1, 5, 4, 1, 10]
>>> lst.insert(2, 6) # insert number 6 into position 2
>>> lst
[1, 5, 6, 4, 1, 10]
>>> lst.insert(5, 2) # insert number 2 into position 5
>>> lst
[1, 5, 6, 4, 1, 2, 10]
```

(b) Write an `insert_tup` function that takes in an element and a **tuple** and inserts the element into the resulting tuple at the first position where it is less than or equal to the next element. The function returns the resulting tuple. You are not allowed to convert into a list nor use any `list`'s operations like `.insert()`. **(4 marks)**

```
>>> insert_tup(2, (1, 5, 9))
(1, 2, 5, 9)
>>> insert_tup(10, (1, 5, 9))
(1, 5, 9, 10)
>>> insert_tup(5, (2, 6, 8))
(2, 5, 6, 8)
```

(c) Finally, let's examine the output's value and identity. Compare the input and output for both insert functions using `==` and `is`, and write down the output in the template. Explain the outputs. **(3 marks)**

Task 3: Let's sort it out (9 marks)

Use the functions that you have just defined to complete the following tasks.

(a) Now that you have your insert function, we can now write the sorting function. Define a function `sort_list` that takes in a **list** and returns a **list** sorted in the ascending order. **(4 marks)**

Note: your code should make use of the `insert_list` function defined earlier.

```
>>> sort_list([5, 1, 9])
[1, 5, 9]
>>> sort_list([6, 2, 5, 1, 3, 4])
[1, 2, 3, 4, 5, 6]
>>> sort_list([5, 0, -1, 4, -2])
[-2, -1, 0, 4, 5]
```

(b) What is the time complexity of `sort_list`? **(1 mark)**

Note: List concatenation and slicing (for a list with n items) are each $O(n)$ in time complexity.

(c) Define a function `sort_tuple` that takes in a **tuple** and returns a **tuple** sorted in ascending order. The algorithm defined here should use `insert_tup` defined in Task 2b. **(4 marks)**

Hint: Since we are working with tuples here, you can use `accumulate` to help you with this task.

```
>>> sort_tuple((5, 1, 9))
(1, 5, 9)
>>> sort_tuple((6, 2, 5, 1, 3, 4))
(1, 2, 3, 4, 5, 6)
>>> sort_tuple((5, 0, -1, 4, -2))
(-2, -1, 0, 4, 5)
```

Task 4: Blocks and Shelves (8 marks)

After organizing your socks, t-shirts and jeans you go back to your room to see your bookshelf in a mess.

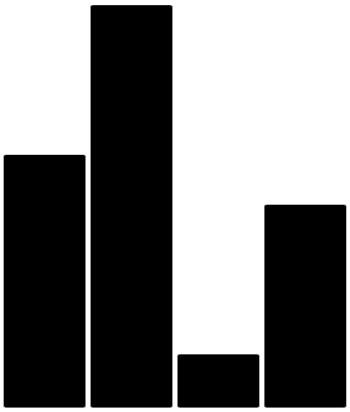
"Ack! Who messed up my books? Hmm... Good thing I know how to do sorting now."

The books in this task are represented as Block objects and the bookshelf is represented as a Shelf object. A Shelf is a collection of Blocks.

Let's try creating some blocks and putting them in a shelf.

```
>>> import shelf
>>> s = shelf.init_shelf([5, 8, 1, 4])
```

You should see a window entitled **Python Turtle Graphics** appearing on your machine and four black blocks in the window.



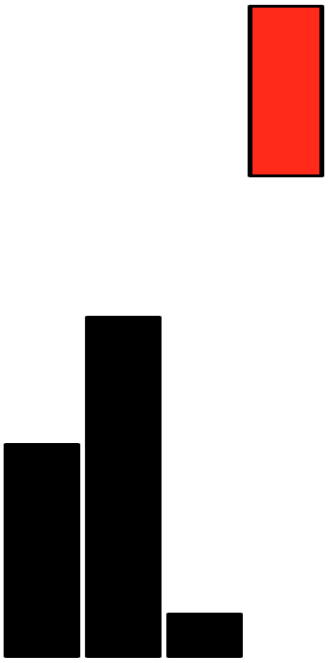
The values of elements in the list that we supplied to the function `init_shelf()` represent the size of the blocks. Try changing the set of numbers that you pass into the `init_shelf()` function!

Note: To clear the screen, call the `shelf.clear_window()` function.

Now, let's try to manipulate the shelf! To remove a block from the shelf, call the `pop()` function on the shelf like this:

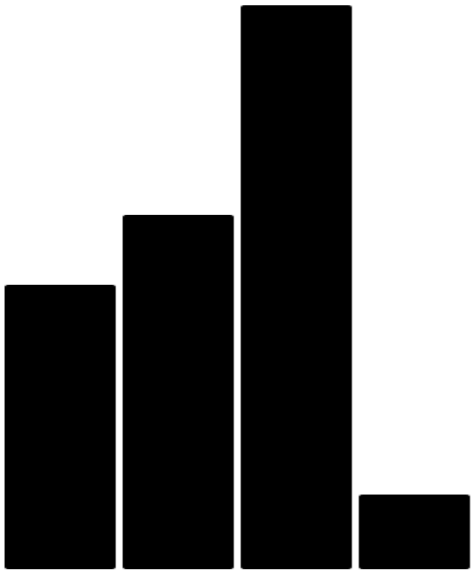
```
>>> b = s.pop(3)
```

`pop()` is a function that takes in the position of the block you want to remove. The block that is removed from the shelf is returned from the function. Note that we have assigned the returned block to the variable `b` which will be referenced later. You should see this when you execute the code above:



To insert a block, call the `insert` function which takes in a position that you want to insert the block into and the block object that you want to insert. So, let's say if we want to insert the block that we have removed into the first position, we will call `insert` this way:

```
>>> s.insert(0, b)
```



One last thing you need to know about your blocks is the sizes of the blocks. To find out the size of a block, simply do this:

```
>>> s[1].size
5
>>> s[0].size
4
>>> s[0].size < s[1].size
True
>>> type(s[0].size)
builtins.int
```

This will return the size of the block in the form of an integer.

Note: Do not assign blocks to an index of the shelf directly like this `s[0] = block` or do direct list assignments like this `s[:3] = s`. Unexpected things will happen and doing so will void the warranty of your computer! You have been warned! Use the `insert` and `pop` functions to manipulate the shelf.

Now, you should be ready to cast some sorting spells on your bookshelf!

(a) Let's cast the insert spell on our bookshelf. Make use of `pop`, `size` and `insert` to define a function `insert_animate(block_pos, shelf, high)`. The function should take in a block position, the shelf object and a high position. This function should remove the block at `block_pos` from the shelf and insert the block into the position such that its size is less than or equal to the succeeding block's size. The function should find this position between zero and the high position, **non-inclusive** of high. If no such position is found, the block is inserted at the high position.

Say, we have a shelf `s = [1, 5, 8, 2, 3]`. Calling `insert_animate(3, s, 3)` will pop block 2 and insert it into position 1 as it is less than block 5, giving us a mutated shelf `[1, 2, 5, 8, 3]` **(4 marks)**

```
>>> s = shelf.init_shelf((5, 2, 4, 1))
>>> print(insert_animate(1, s, 1))
[Block size: 2, Block size: 5, Block size: 4, Block size: 1]
>>> print(insert_animate(3, s, 3))
[Block size: 1, Block size: 2, Block size: 5, Block size: 4]
```

(b) Define a function `sort_me_animate` that uses the `insert_animate` that you have defined earlier to sort the shelf according to their sizes in the ascending order. The algorithm should be similar to the sorting technique that we described earlier where it maintains a sorted portion of the shelf and inserts every element into the sorted portion. **(4 marks)**

```
>>> s = shelf.init_shelf((8,5,2,4,1))
>>> print(sort_me_animate(s))
[Block size: 1, Block size: 2, Block size: 4, Block size: 5, Block
size: 8]
```