Computer Laboratory 04

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Tuesday February 25st at 11:59am and will be turned in on Gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.

2 Introduction

This lab is intended to help you practice working with the linear and binary search algorithms. However, because these algorithms are well-known and well-documented, we will be applying them to slightly modified list formats. Understanding this modified data representation will require some effort, but once you understand it, it should not be hard to adapt the search algorithms and code we've seen in lecture to this novel task.

This laboratory is intended to give you practice with three core skills:

- Using the linear and binary search algorithms
- Thinking flexibly about different ways of representing data
- Modifying algorithms you know for slightly different problems

Note Parts of this lab will talk about big-O runtimes, which we have not directly covered in lecture yet. You've read about these, and most of the requirements will follow directly from the algorithm requirements (linear/binary search). So long as you're code isn't excessively inefficient most straightforward solutions will hit the runtime requirements.

3 Files

This lab will involve the following files:

- (REQUIRED) pricebook.py. – you will program this.
- (PROVIDED) pricebook_tester.py – we will provide this to help you test your code.

You should begin by downloading all of the provided files You should then make a new python file for your solution. Don't forget that you are required to have a comment at the top of any file you author that contains your name, and the name of any lab partners you worked with.

4 Initial Steps

The initial steps for this lab can reasonably be done in collaboration with other groups – but you should make sure to stop before you get to actually coding a solution to the problem.

1. Create a new PyCharm project for this lab.
2. Download all of the provided files and move them into your PyCharm project folder.
3. Read and discuss the next section – make sure you understand how the data in this assignment is organized.
4. You will also need to understand linear and binary search algorithms. It might make sense to perform a simple binary search on your own, tracking low, and high variables manually.
5. Preview the required functions and work through a few examples using pencil/paper to make sure you understand what behaviors you've been asked to implement.

Remember – all of this can be discussed with other groups, not just your partner. Once you've understood all of this, you're ready to start working on the specific computations we want you to program.

5 pricebooks association lists

Task-specific data structures are common in computer science – these are often variants of normal data structures that we design for one specific job. For example, when working in a grocery store, you might regularly need a "pricebook" which is a mapping of products to their costs:

For example:

Product	Price
Lego	80.0
Laptop	690.0
Camera	300.0
Knife	5.90
Headphone	194.4

The natural data structure to store this information is a python dictionary:

```
prices = {'Lego' : 80.0, 'Laptop' : 690.0, 'Camera' : 300.0, '
   Knife' : 5.9, 'Headphone' : 194.4 }
```

However, just for this lab, we're going to **imagine a world without dictionaries**. This means we'll have to find some other way to store this information.

Most programmers would react to not having dictionaries by writing functions to simulate the behavior of a dictionary while really using some other data structure. This is the approach we are going to take – we are going to "pretend" that dictionary is not implemented in Python and write functions to implement our own pricebook management code. Think of this as a preview of the work involved towards the end of the semester, when we'll see how dictionaries are implemented, and be writing our own version!

The key of our pricebook representation will be "association lists" – these are list objects that store the product-price pairs using a tuple. For example, we could represent the previous pricebook with the following association list:

```
prices = [(80.0, 'Lego'),
          (690.0, 'Laptop'),
          (300.0, 'Camera'),
          (5.9, 'Knife'),
          (194.4, 'Headphone')]
```

Notice that each element in this list is a tuple, with the first position being the *price* and the second position being the *product name*. This choice is arbitrary (I.E. basically any order could be made to work). While (price, product name) might not be your first choice, it's as good as any other, and what we'll be using.

The order of the tuples within the list is essentially arbitrary, we could represent the same pricebook many different ways:

```
price1 = [(80.0, 'Lego'), (690.0, 'Laptop'), (300.0, 'Camera'),
   (5.9, 'Knife'), (194.4, 'Headphone')]
price2 = [(690.0, 'Laptop'), (300.0, 'Camera'), (5.9, 'Knife'),
   (194.4, 'Headphone'), (80.0, 'Lego')]
price3 = [(300.0, 'Camera'), (5.9, 'Knife'), (194.4, 'Headphone'),
    (80.0, 'Lego'), (690.0, 'Laptop')]
```

All three list-of-tuples above represent the same pricebook. Because we have this freedom we're going to explore two options in this lab:

1. unsorted pricebooks – (Store tuples in any order)

2. sorted pricebooks – (Sort the prices by product name)

As we will see, it's possible to get much faster price-lookup with a sorted pricebook, but maintaining a pricebook in sorted order requires a more complicated algorithm.

6 Formal Requirements

You will be required to implement 6 functions:

1. `is_sorted(pricebook)`
2. `price_average(pricebook)`
3. `unsorted_get(pricebook, name)`
4. `unsorted_put(pricebook, name, price)`
5. `sorted_get(pricebook, name)`
6. `sorted_put(pricebook, name, price)`

6.1 is_sorted

Since we have different functions for manipulating sorted and unsorted pricebooks, it makes sense to want to check if a pricebook is currently stored in sorted order. This function takes a pricebook (I.E. a list of tuples as defined earlier) and returns either True or False to indicate if the list is sorted.

Notes:

- If given an empty list, return True
- We want lists to be sorted based on *product names* not prices. – remember those are the second position in the tuple, not the first.
- We want the list sorted smallest to biggest (I.E. alphabetic order)
  - **HINT** – this is the "natural order" for python strings – so you can just use the `<` and `>` operators
- This function should not modify the list passed to it.
- This function is required to run in time $O(n)$ – if you use a single loop you should be fine.
- This function doesn't need to be too complicated – I recommend looping in-order and seeing if each element is bigger than the one preceding it!
- (The real purpose of this function is to "warm up" for the later functions and make sure you're familiar with the pricebook list-of-tuples format.)

6.2 price_average

A common pricebook-task is to get the average price for a given product category or shopping cart. The price_average function should take a pricebook (list-of-tuples as above – this can be sorted or unsorted) and returns the average price of all products in the pricebook.

Notes:

- if given an empty list, return 0.0
- This function should not modify the list passed to it.
- This function is required to run in time $O(n)$ – if you use a single loop you should be fine.
- This function doesn't need to be too complicated – like the last one it's *real* purpose is to help you familiarize yourself with the pricebook list-of-tuples format before tackling the harder problems.

6.3 unsorted_get

The unsorted get function replicates the behavior of the `dict.get(key)` method. It takes two parameters: an unsorted pricebook and a product's name. The function should use a linear search algorithm to check if the named product is in the pricebook. If the product is in the pricebook, the function should return its price. Otherwise, the special value None should be returned. (Note that's not a string, it's just None)

```
return None
```

Notes:

- This function should not modify the list passed to it.
- This function must run in time $O(N)$ – use the linear search algorithm and you should have no trouble making this runtime requirement.

6.4 unsorted_put

The unsorted put function replicates the behavior of the dictionary assignment statement: `dict[key] = values`. It takes three parameters: an unsorted pricebook, the products name, and the product's price. This function should then update the pricebook to record that product's new price (either updating an existing position in the list, or adding a new element to the list)

Notes:

- If the product is contained in one of the tuples in the pricebook, then only the price should be modified – remember tuples cannot be modified so you will need to replace the entire `(price, name)` tuple with an updated version.
- If the product is not currently in the pricebook you will need to add the price-product pair somewhere in the list. For grading purposes, we are requiring you to add the `(price, product)` tuple to the end of the list in this case.
- This function **SHOULD** modify the list given to it.

- This function must run in time $O(N)$ – use the linear search algorithm and you should have no trouble making this runtime requirement.

6.5 sorted_get

Like the unsorted get function, the sorted get function replicates the behavior of the `dict .get(key)` method. It takes two parameters: a **sorted** pricebook and a product's name. The function should then use a binary search to check if the product is in the pricebook. If the product is in the pricebook, the function should return its price. Otherwise, the special value None should be returned (as before – this isn't a string, it's a special value)

```
return None
```

Notes:

- You can assume that the pricebook provided is sorted by products name in alphabetic order.
- You may find it useful to look at the examples in the test file to see how this is structured if you are unsure – coding for the wrong structure or sort order is a good way to lose time on this assignment.
- You are free to reference the in-class or textbook binary search code – just know that you will need to modify it for the pricebook's format.
- This function should not modify the list passed to it.
- This function must run in time $O(\log N)$ – use the binary search algorithm and you should have no trouble making this runtime requirement.

6.6 sorted_put

Like the unsorted put method, the sorted put method replicates the behavior of the dictionary assignment statement: `dict[key] = values`. It takes three parameters: a **sorted** pricebook, the products name, and the product's price. This function should then update the pricebook to record that product's new price, while carefully maintaining the sorted order. *The sorted order here means sort by the key (product name), instead of value (price).*

Notes:

- You'll need to check if the product is already in the pricebook or not. You are free to use a binary search or linear search for this step (as noted later – we only require $O(N)$ runtime for this function.)
- If the product is already in the pricebook, only the price for that product should update – remember that tuples cannot be modified so you will need to replace the entire `(price,name)` tuple with an updated version.
- If the product is not contained in the pricebook, you will need to add a new `(price ,name)` tuple somewhere in this list. This is the hard part: you must add the tuple in such a way that the list stays sorted (so you can't just throw the new tuple at the end). Some hints about this:
  - You will ultimately need to call the `list.insert` method.

  - The `list.insert` method takes $O(N)$ time all on it's own. This means that your function cannot run faster than $O(N)$. As such, feel free to use a simple linear search to find where the new tuple belongs. You are not required to use a modified binary search for this.
  - Don't forget to handle the case where the new tuple belongs at the beginning of the list (product is first in alphabetic order) or the end of the list (product is last in alphabetic order) – these cases may require special cases in your code.
- This function is required to run in time $O(N)$
  - Because we do not require $O(\log N)$ time – you don't need to use binary-search logic anywhere in this function.
  - Note – sorting in python is typically, $O(N \log N)$ – which is bigger than $O(N)$. Using a full-list sorting algorithm or function will be treated as failing the runtime requirement for this function.
- **This function is probably the hardest part of the lab** – this is on purpose as it gives you a chance to design an algorithm on your own, rather than using a pre-designed algorithm. Make sure you're planning time for this function takes longer than the rest.
- This function **SHOULD** modify the list given to it.

7 Grading

Grading on this assignment will be **MOSTLY** automatic:

- 5 points for submitting a file meeting all requirements
- 6 points for passing tests on the price_average function
- 6 points for passing tests on the is_sorted function
- 7 points for passing tests for unsorted_get
- 12 points for passing tests for unsorted_put
- 18 points for passing tests for sorted_get
  - Note – one of these tests checks if your code runs in time $O(\log N)$ by creating a big pricebook and ensuring the run-time of sorted_get remains small. An $O(N)$ algorithm will fail this test.
- 24 points for passing test for unsorted_put

This leaves 22 points for manual reviews. We will perform 3 basic manual reviews:

- Code Style (10 points) – See past labs for a list of specific expectations. Remember to use your best judgment about how to make your code's design clear. Remember – code is communication!
- Efficiency (12 points – 2 per function) – we will manually review your algorithms for their runtime according to the requirements listed in this PDF. If your code style makes it impossible to determine the runtime efficiency you will lose points for code style **AND** efficiency – so make sure your code is legible and your algorithms are well-formed.
- Irregularities – we will be looking for any attempts to "cheat" the autopricer and get

points for a function that is clearly only able to work for the tested examples.