CompSci 107 – Computer Science Fundamentals
S1 – 2015
Lab/Assignment 8 – Hash tables, sorting

Assignment Due Date: 7pm May 22, 2015

Worth: This assignment is This assignment is worth 2.5% of your final grade. The lab will be marked out of 10. The programming exercises will be marked out of 9, with the remaining 1 mark from the code review process.

Topics covered:

- Map ADT, sorting algorithms

NOTE: Download the "A8.zip" file from the assignments website:

https://www.cs.auckland.ac.nz/courses/compsci107s1c/labs/

PLEASE NOTE

The following exercises **must be developed on your computer (or on your USB, or on your university drive). Note that you may be asked to produce the code you developed.** Once you are happy that your answer for each question is correct, you will then need to insert the part of the answer required into CodeRunner.

Exercise 1 (1 mark).

Define a hash function, `hash_function()`, which uses a mid-square technique (the keys are always integers of length greater than 7). To get the hash value of a key, the hash function takes the digits from positions two, three, four, five and six of the key (position 2 is the 3rd digit), adds 23456, squares the number, removes the first and last digit from the result and, finally, takes a modulus of 23, the table length.

Run the `A8Ex01` application which should print two lists which contain exactly the same numbers (the hash values).

Once you are happy that your function is correctly coded, insert the function code into CodeRunner, assignment 8 question 1.

Exercise 2 (2 marks).

The Map Abstract Data Type has the following operations:

put(key,value): add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
get(key): given a key, return the value stored in the map, None otherwise.
delete(key): delete the key-value pair from the map.

Define an implementation of the Map ADT. Your implementation uses two parallel lists, one to store the keys and one to store the corresponding values (the `__init__()` method is shown on the next page). Your implementation should use separate chaining to resolve collisions (each element of the two lists making up your hash table items stores a Python list).

You will also need to add the following methods to your implementation:

def __len__(self): Returns the size of the hash table.
def __contains__(self, key): Allows the use of the **`in`** operator.
def __delitem__(self, key): Allows the use of statements such as: **`del hash_t[a_key]`**
to delete a key-value pair.
def __getitem__(self, key): Allows the use of statements such as:
`value = hash_t[a_key]`
to retrieve the value corresponding to the key.
def __setitem__(self, key, data): Allows the use of statements such as:
`hash_t[a_key] = value`
to add a key-value pair.
def hash_function(self, key): Add your hash function from exercise 1 – you will need to add the parameter, `self,` and use `self.size` instead of the size 23.

The constructor of your implementation takes one parameter, the length of the hash table:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.slots = []
        self.data = []
        for i in range(self.size):
            self.slots.append([])
            self.data.append([])
```

Name your class, `HashTable`, and store it in a file named, `MapWithChaining.py`. To test your completed implementation, run the Lab8Ex02 application. The output produced should be as shown in the docstring comment at the bottom of the application. Note: the files needed for this exercise are:

HashTableWithChaining.py	#your Map implementation
IDNames.txt	# The file of id-names pairs
A8Ex02.py	# The application which tests your implementation

Once you are happy that your implementation is correctly coded, insert the nine methods of your implementation code into CodeRunner, assignment 8 question 2.

Exercise 3 (1 mark). The hash table load factor and other statistics

Using your Map implementation from exercise 2 above, add the method, `get_hash_table_stats()`. This method returns a tuple containing eight statistics about the state of the hash table:

(
the load factor,
total number of items in table,
the minimum chain length (only considering chains of length greater than zero),
how many slots have chains of the same length as the minimum length,
the maximum chain length,
how many slots have chains of the same length as the maximum length,
the average chain length (only considering chains of length greater than zero),
number of zero length chains
)

Run the A8Ex03 application which should print seven lines of statistics. The output produced should be as shown in the docstring comment at the bottom of the application.

Once you are happy that your method is correctly coded, insert the `get_hash_table_stats()` method (you may wish to define a helper method/s) into CodeRunner, assignment 8 question 3.

Question 4 (1 mark). Bubble sort with fewer comparisons.

The bubble sort function is given in the A8Ex04 application. Add code to the `a_bubble_sort()` function so that, as well as sorting the parameter list, the function returns a tuple containing the total number of swaps and the total number of comparisons.

Define a second function, `improved_bubble_sort()`, which performs the same bubble sort but stops the function execution as soon as all the elements in the parameter list are sorted. This function returns a tuple containing the total number of swaps and the total number of comparisons.

Run the A8Ex04 application. The first four lines of your output should be the same as

```
before:  [30, 46, 33, 35] ... [78, 80, 93]
after:   [30, 33, 35, 36] ... [78, 80, 93] True
Unimproved version: swaps: 6, comparisons: 55
Improved version:   swaps: 6, comparisons: 27
```

The rest of your output will be different to that shown inside the docstring comment at the bottom of the A8Ex04 file, but you should see a slight improvement in the number of comparisons needed when the `improved_bubble_sort()` function is used to bubble sort the elements.

Once you are happy that your functions are correctly coded, insert the definition of the `improved_bubble_sort()` function into CodeRunner, assignment 8 question 4.

Question 5 (1 mark). Insertion sort

The following list of numbers is being sorted using the insertion sort algorithm with the smallest element on the left of the list. The sorted part of the list is growing from left to right.

```
[11, 7, 12, 3, 1, 8, 14, 9, 13, 2]
```

In CodeRunner, assignment 8 question 5 show the list of elements **after** the 5th pass has been completed. Your answer should be a list of numbers separated by commas, e.g., [...]

Question 6 (1 mark). Selection sort

The following list of numbers is being sorted using the selection sort algorithm with the smallest element on the left of the list. The sorted part of the list is growing from right to left.

```
[11, 7, 12, 3, 1, 14, 8, 9, 13, 2]
```

In CodeRunner, assignment 8 question 6 show the list of elements **after** the 5th pass has been completed. Your answer should be a list of numbers separated by commas, e.g., [...]

Question 7 (1 mark). Shell sort

The following list of numbers is being sorted using the shell sort algorithm with the smallest element on the left of the list and the largest on the right. The list is being sorted using a gap size of 5 followed by a gap size of 2. Show the elements after the gap size of 2 has been completed.

```
[9, 8, 11, 6, 5, 3, 4, 7, 1, 2]
```

In CodeRunner, assignment 8 question 7 show the list of elements. Your answer should be a list of numbers separated by commas, e.g., [...]

Question 8 (1 mark). Mergesort

The following list of numbers is being sorted using the mergesort function (lower down):

```
[5, 6, 8, 14, 7, 11, 5, 12, 2, 5]
```

```
def mergesort(a_list):
    if len(a_list) > 1:
        middle = len(a_list) // 2
        left_half = a_list[:middle]
        right_half = a_list[middle:]

        merge_sort(left_half)
        merge_sort(right_half)
                  #merge_two_halves() function not shown here
        merge_two_halves(a_list, left_half, right_half)
        #POSITION X
```

POSITION X is the position in the code AFTER the left half of the list and the right half of list have been merged. Draw a recursive call tree for the above mergesort using the list of elements above. At one point, after the two halves have been merged, the merged list contains **exactly two occurrences of the element 5**. Show all the elements of this merged list.

In CodeRunner, assignment 8 question 8 show the list of elements. Your answer should be a list of numbers separated by commas, e.g., [...]

Exercise 9 (1 mark). Participation in the code review of the programming exercises.