Homework #9
Due by Monday 12/11, 11:55pm

Submission instructions:

1. You should submit your homework to Gradescope under Homework #9, NOT NYU Classes. For Gradescope's autograding feature to work:
   - Make sure to include **all** classes used. **Do not use import statements that require external files.**
   - Name all functions and methods **exactly as they are in the assignment specifications**.
   - Make sure there are **no print statements** in your code. If you have tester code, please put it in a "main" function and **do not call it**.
   - Make sure to use your netID in the file name and **not** your N number. Your netID follows an **abc123** pattern, not N12345678.

2. For this assignment, you should turn in 4 files:
   - A '.pdf' file containing your answers to questions 1 and 2.
     Name this file 'YourNetID_hw9.pfd'.
   - A '.py' file for each one of questions 3-5.
     Name your files 'YourNetID_hw9_q3.py', 'YourNetID_hw9_q4.py', etc.

Question 1:

In this question, you will insert following set of keys: 12, 56, 22, 106, 36, 72, 902, 86, 96, 62 and 42, to three different hash tables.
In all cases, collisions are resolved by chaining. The tables will differ by their size and by the compression method used.

a. Draw the table resulted after inserting the keys to a table of size $N=10$ (a non-prime table size), where we use the division method as a compression function. That is, the compression function is: $h_2(k) = k \bmod 10$.

b. Draw the table resulted after inserting the keys to a table of size $N=13$ (a prime table size), where we use the division method as a compression function. That is, the compression function is: $h_2(k) = k \bmod 13$.

c. Draw the table resulted after inserting the keys to a table of size $N=10$ (a non-prime table size), where we use the MAD method as a compression function. For the MAD constants, we picked are: $p=1009$, $a=125$ and $b=342$. Therefore, the compression function is: $h_2(k) = ((125*k+342) \bmod 1009) \bmod 10$.

Notes:

1. For all sections, assume that we will insert the keys to the table without coding them. That is the code-function $h_1(k)$, doesn't change the keys ($h_1(k)=k$). This way, the hash function $h(k)$, will only compress the keys into the slots of the table ($h(k) = h_2(k)$).
2. For all sections, you do not need to rehash the table

Question 2:

In this question, you will insert and delete items to/from a $N=11$ length open-addressing hash table, where we use the division method for compression (the compression function is: $h_2(k) = k \bmod 11$), and linear probing for resolving collisions.

- We start with the following insertions: 59, 39, 135, 91, 46, 132, 169 and 277
- We then delete: 39 and 46
- Finally, we insert: 157

Draw the table resulted after executing the operations above.
Notes:

1. In this question too, assume that we will do not use a coding-function to code the keys.
2. You do not need to rehash the table

<u>Question 3</u>:
Modify the implementation of the `ChainingHashTableMap` class, so that the `__iter__` method would report the keys in the map according to first-in, first-out (FIFO) order. That is, the key that has been in the map the longest is reported first, etc. (The order is unaffected when the value for an existing key is updated).

<u>Implementation requirements</u>: You should support searching inserting and deleting in $\Theta(1)$ expected time (average-case), and iterating in $\Theta(n)$ worst case.

<u>Question 4:</u>
In this question, we give two implementations for the function:

```
def intersection_list(lst1, lst2)
```

This function is given two lists of integers `lst1` and `lst2`. When called, it will create and return a list containing all the elements that appear in both lists.

For example, the call:
`intersection_list([3, 9, 2, 7, 1], [4, 1, 8, 2])` could create and return the list `[2, 1]`.

<u>Note</u>: You may assume that each list does not contain duplicate items.

a) Give an implementation for `intersection_list` with the best **worst-case** runtime.

b) Give an implementation for `intersection_list` with the best **average-case** runtime.

<u>Question 5:</u>
An ***inverted file*** is a critical data structure for implementing a search engine or the index of a book.
Given a document (text file) *D*, which can be viewed as an unordered, numbered list of words, an *inverted file* is a map, such that, for each word *w*, we associate the indices list of the places in *D* where *w* appears.

For example, if "row your boat.txt" is the following text file:

```
Row, row, row your boat
Gently down the stream,
Merrily, merrily, merrily, merrily
Life is but a dream

Row, row, row your boat
Gently down the brook,
If you catch a little fish
Please let it off the hook
```

It can be looked at as an unsorted list of words: *D = ['row', 'row', 'row', 'your', 'boat', 'gently', 'down', 'the', 'stream', … , 'please', 'let', 'it', 'off', 'the', 'hook']*.
In the inverted file map, the list associated to the word *'row'* is *[0, 1, 2, 18, 19, 20]*.

More on inverted files you can find in https://en.wikipedia.org/wiki/Inverted_index

Implement the following class:

```
class InvertedFile:
    def __init__(self, file_name):
        ''' Initializes an InertedFile object representing
        the inverted file of file_name'''

    def indices(self, word):
        ''' Returns a list containing all the indices of
        the places in the file where word appears '''
```

Notes:

1. Your implementation should ignore the casing of the letters (That is the word *'Row'* should be considered equivalent to the word *'row'*).
2. Your implementation should ignore punctuation marks (That is *'Row,'* should be considered equivalent to *'row'*).
3. If you are asked for the indices of a word that doesn't exist in the file, you should return the empty list.

For example, after implementing, you should expect the following behavior:

```
>>> inv_file = InvertedFile("row your boat.txt")
>>> inv_file.indices("row")
[0, 1, 2, 18, 19, 20]
>>> inv_file.indices("the")
[7, 25, 37]
>>> inv_file.indices("done")
[]
```

Implementation requirements:

1. The `__init__` method should run in linear expected time. That is if there are n words in the file, the initialization should take in $\Theta(n)$ expected time (average-case).
2. The `indices` method should run in $\Theta(1)$ expected time (average-case).