Problem Set 3

All parts are due Thursday, March 20 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 3-1. [20 points] **Set Theory Language** In your new job at the Mathematically Intricate Technologies (MIT) company, you are responsible for implementing a little language that will allow student users to manipulate sets (of natural numbers) using the usual set operations. Specifically, they should be able to define a new set by inputting an unordered list (array) containing its elements, without any repeats. Once they have defined sets, they should be able to perform the following operations:

- Test whether a particular natural number k is in a set S.
- Add or delete a single element k to/from a set S.
- Define a new set T as the intersection of two previously-defined sets, $T = S_1 \cap S_2$.
- Define a new set T as the union of two previously-defined sets, $T = S_1 \cup S_2$.
- Define a new set T as the difference between two previously-defined sets, $T = S_1 - S_2$.
- Output the elements of a set S, in the form of an unordered list without repeats.

Your membership test, insertion, and deletion operations should take constant time $O(1)$, on the average. Each of your other operations should take time $O(n)$ on the worst case, where n is the total number of elements in the sets involved in the operation.

To implement this language, the company provides you with a proprietary abstract data type—a simple dictionary data type that supports just $Search(key)$, $Insert(key, value)$, and $Delete(key)$ operations for natural-number keys, all operating within time $O(1)$ on the average. (This is similar to, but not the same as, the Python dictionary data type. For example, the Python type also supports other operations such as keys(), which outputs a complete list of all the keys in the dictionary.) We assume that, if $Insert(k, v)$ is invoked when k is already in the dictionary, we get an "already there" response, and similarly, if $Delete(k)$ is invoked when k is not in the dictionary then we get a "not there" response. In both of these cases, the abstract dictionary does not change. For this problem, you should use the dictionary simply as an abstraction. You should not think about how the dictionary is implemented.

(a) [3 points] Describe the data structure that you will use to implement each set.

(b) [3 points] Describe (in clear English or pseudocode) your algorithm to produce a set from an unordered list of numbers (without repeats), and your algorithm to output a set in the form of an unordered list (without repeats). Analyze their time complexity.

(c) [2 points] Describe your set membership algorithm and analyze its time complexity.

(d) [3 points] Describe your insertion and deletion algorithms and analyze their time complexity.

(e) [3 points] Describe your set intersection algorithm and analyze its time complexity.

(f) [3 points] Describe your set union algorithm and analyze its time complexity.

(g) [3 points] Describe your set difference algorithm and analyze its time complexity.

Problem 3-2. [15 points] **Probabilistic Analysis of Hashing With Chaining**

Analysis of hashing with chaining usually assumes a "simple uniform hashing" property of the operation sequence: regardless of what has happened in the past, the key that appears in the next operation in the sequence is equally likely to be one that hashes to any location in the table. This assumption makes analysis easier, and is approximately achieved by real hash functions.

This problem considers some issues involved in analyzing the behavior of hash tables with chaining, under various uniformity assumptions. In all the examples, the hash table has m locations numbered $0, \ldots, m-1$. We are considering scenarios in which n Insert operations occur for successive, distinct keys $k_1, k_2, \ldots, k_n$, followed by a single Search operation. The various scenarios differ in their probabilistic assumptions, which leads to different probabilistic analysis results.

(a) [5 points] Suppose that the n Inserts happen to be all for keys that hash to the same table location. This is not *likely* but it could happen. The Search operation requests a key randomly and uniformly from the key space. The "uniform hashing assumption" then implies that the resulting hash value is equally likely to be any location in the hash table. What is the expected time for the Search operation to complete, as a function of n and m?

(b) [3 points] Again suppose that the n Inserts are all for keys that hash to the same table location. But this time, suppose that we know that the Search is successful, that is, it requests a key that is actually in the table. Suppose that it is equally likely to be any key in the table. Give a good bound on the expected time for the Search operation to complete.

(c) [7 points] But bad tables are not so likely. Now assume that all the n Insert operations satisfy the simple uniform hashing assumption. Suppose that we know that the Search is successful, and is equally likely to be any key in the table. Prove an upper bound of $n/m + O(1)$ on the expected time for the Search operation to complete.
Hint: For each inserted key k_i, define a random variable CH_i representing the number of inserted keys that hash to the same location as k_i. Analyze the expected value of CH_i.

Problem 3-3. [25 points] **Meandering Hashes**

Rolling hashes are special kinds of hash functions for strings with certain nice modification properties. Namely, given the hash $h(x)$ of a string x and a single character c, it is very efficient to calculate the hash of the string y that is obtained by appending c at the right end of x. Likewise, given $h(x)$, it is very efficient to calculate the hash of the the string obtained by removing the first symbol of x. In class, we showed how to implement rolling hashes, and how to use them to implement fast solutions to a basic string pattern-matching problem. Specifically, finding a match for a length k string in a much longer length n string, or determining that there is none, takes time $O(n)$ using rolling hashes. This analysis assumes that we can compare individual characters in time $O(1)$, can calculate a hash of a given length-k string in time $O(k)$, and can compare hash values and update hash values in time $O(1)$.

Now consider a 2-dimensional pattern-matching problem. You are given an $n \times n$ matrix T and a $k \times k$ pattern matrix S. Both are matrices of bits. The problem is to find a match for the pattern S in T if there is one, or determine that there is none. You may use a rolling hash as an abstract data type, that is, assume that you are given a rolling hash implementation.

(a) [3 points] The brute-force approach is simply to try to match S starting from every square in T as a potential upper left-hand square of S. What is the time complexity of this algorithm, according to the cost model given?

(b) [10 points] Design a more efficient algorithm for 2-dimensional pattern-matching. Describe it in words. Rolling hashing ideas may be useful, but describe carefully how you use them.

(c) [5 points] Give pseudocode for your algorithm in part (b).

(d) [5 points] Analyze the time complexity of your algorithm.

(e) [2 points] What is the space complexity of your algorithm over and above the space needed to store S and T?

Part B

Problem 3-4. [40 points] **An IMDb Dictionary with Open Addressing**

Shortly after being promoted to Master Coder at the Mathematically Intricate Technologies (MIT) company, your are assigned a new project. You are asked to analyze the IMDb database (Internet Movie Database), and to try to estimate the effect of several movies on actors' careers. For example, playing the lead female role in a 007 film has been rumored to be detrimental to an actress' career—a statement MIT would like to disprove, or support with evidence.

Before you are ready to work with IMDb data (there is quite a bit of it!), you need to implement an appropriate data structure. Normally, one would use a Python dictionary, but where is the fun in that, right? Instead, you will build your own dictionary abstract data type in three ways, using

three different open address hashing strategies. You will then use your dictionary implementations to answer a few queries based on an IMDb database.

For your dictionaries, the operations you must support are the following:

- $Insert(key, data)$, which adds $(key, data)$ to the dictionary provided that the key does not already appear, and returns some value "okay" (in this case `True`). If the key is already in the table, it returns some value for "already there" (in this case, `False`) and does not modify the dictionary.
- $Delete(key)$, which deletes any pair for the given key from the table, and returns some value "okay" (in this case `True`). If the key is not in the table, it returns some value for "not there" (in this case, `False`).
- $Search(key)$, which returns the data associated with the key in the dictionary, if the key appears there, otherwise returns some value "not there" (in this case, `None`).

The three hashing strategies you will implement are linear probing, double hashing, and Cuckoo hashing. The first two were covered in lecture and are described in CLRS, while the third is explained briefly below.

1. Open addressing with linear probing (you **have** to read CLRS, p. 272). This is the most straightforward strategy.
2. Open addressing with double hashing (you **have** to read CLRS, p. 272-274). This is more interesting, and generally performs better.
3. Cuckoo hashing. This is a relatively new method. You can find information about it on Wikipedia, or can look at the original 2001 paper by Rasmus Pagh and Flemming Friche Rodler called "Cuckoo Hashing". The basic idea is as follows.

   We use two *auxiliary* hash functions, h_1 and h_2, which map from the key space to a hash table index in $[0, m-1]$. Element x with key k can be stored only in locations $h_1(k)$ or $h_2(k)$, so lookup is very fast. Deletion involves simply looking up the element and removing it from the table location.

   Insertion is more interesting: To insert element x with key k, compute $h_1(k)$ and store the element at the computed location. If location $h_1(k)$ already contained an element, say x' with key k', then this entails *evicting* x' from its current location. That means we need a new place to put x'. Figure out whether x' was previously stored in $h_1(k')$ or $h_2(k')$. Then insert x' into the other one, if necessary evicting the current occupant of that location. And so on. Stop this if it goes on for m steps, to avoid any infinite eviction loops. Normally in the case of such a loop, it is appropriate to rehash the entire table and re-insert the last evicted item. However, we have designed our test cases in such a way that rehashing is not necessary. You need not implement it. See the comments and partial implementation in the code provided for details.

   For this part, you will only be modifying the `PSET3.py` file. Please read the comments in this source file carefully.

(a) [10 points] In the `HashFunctions` class, implement `linear_probing_hash` and `double_hashing_hash`. Both of these methods take a key k, a probe slot i, and the maximum capacity of an open-address hash-table m. They should return an index in the range $[0, m-1]$ that indicates where key k should be stored in that open-address hash-table.

Note that these `linear_probing_hash` and `double_hashing_hash` methods should implement hash functions define the *traversal order* for an open addressing hash-table using the linear probing scheme, and the double hashing scheme, respectively.

You may also find the comments in `PSET3.py` useful.

(b) [10 points] Here we ask you to implement an open-addressing hash table given a hash function and a preallocated array.

In the `OpenAddressedDictionary` class, implement the following methods:

- `insert(key, value)` - Inserts the (key, value) pair in the dictionary and returns `True`. If the key is already in the dictionary, perhaps with another value associated to it, then leave that value intact and return `False`. If the dictionary is full, then return `False`.
- `delete(key)` - Finds and deletes the matching (key, value) pair from the dictionary and returns `True`. If the key is **not** in the dictionary, then leave everything intact and return `False`.
- `search(key)` - Finds the matching (key, value) pair in the dictionary and returns **only** the value. If the key is **not** in the dictionary, then return `None`.

When an `OpenAddressedDictionary` is initialized in the class's `__init__` method you get a few things as parameters:

- `array` - this is the backing of your dictionary: it's where you should store the (key, value) pairs. It's a custom array object preallocated for you (don't worry too much about this).
  - You can use `len(array)` to get the capacity of this array. The capacity is also stored in `self.m` in the `OpenAddressedDictionary` class.
  - You can read a value from this array using the `val = self.array[idx]` syntax.
  - You can write a value in this array using the `self.array[idx] = val` syntax.
- `hash_function` - this is a hash function $h(k, i, m)$ as implemented in part a). This can be either one of the `linear_probe_hash` or `double_hashing_hash` methods you implemented in part a). Your implementation should make abstraction of which hash function it is given: it should work both with linear probing and double hashing. You will need to use this hash function in the three `OpenAddressedDictionary` methods you are implementing.

- For example, you may call it as follows: `self.hash_function(key, probe_no, self.m)`.

You may find the comments in `PSET3.py` to be helpful.

(c) [10 points] Here we will ask you to build a hash table that uses Cuckoo Hashing.

In the `CuckooDictionary` class, implement the following methods:

- `insert(key, value)`
- `delete(key)`
- `search(key)`

These methods behave the same as in part b) but now you are using Cuckoo Hashing instead of open-addressing. Note that this a more sophisticated hash table scheme.

When a `CuckooDictionary` is initialized in the class's `__init__` method, you get an array as a parameter, which serves the same purpose as in part b). You also get the two *auxiliary* hash functions that you need for Cuckoo Hashing. These are stored in `self.h1` and `self.h2`. You need to use these in the three methods you implement to map keys to locations in the arrays.

Important: You are not to worry about rehashing the array in case of an eviction loop. This is because we have tailored our test cases to prevent against such loops.

You may find the comments in `PSET3.py` helpful.

(d) [10 points] Now we are ready to use the data structures we have implemented to perform queries against a reduced IMDb database.

You are given the name of an actor A who starred in a certain film F in year Y. The idea is to see if the film F ruined or improved that actor's career. As a vague approximation for this metric, we will relate the average rating of the actor's subsequent movies to the average rating of the actor's preceding movies (up to and including year Y).

Specifically, you are to compute the actor A's career impact metric $C(A, F)$ which is defined as the difference between:

- $R_{after} \rightarrow$ average rating of films featuring actor A released after year Y in film F
- $R_{before} \rightarrow$ average rating of films featuring actor A released before or during year Y (including film F)

Thus, $C(A, F) = R_{after} - R_{before}$

If actor A did not star in any films after year Y, we define the career impact $C(A, F)$ to be zero.

You will implement your solution by modifying the `IMDB` class.

From looking at the parameters of the `__init__` method, it is evident you will have to use hash tables for this. We give you space for two such hash tables, in the `array_small` and `array_large` variables. Use them wisely.

The IMDb database is given to you in two lists:

- `self.films_list` - stores a list of `(film, rating, year, actor_list)` tuples.
- `self.actors_list` - stores a list of `(actor_name, film_list)` tuples.

Initialize the data structures you intend to use in the `__init__` method.

Implement the $C(F, A)$ career impact query in the `career_impact` method.

Important: You may not use Python's built-in dictionaries (again, that wouldn't be fun). You may find the comments in `PSET3.py` useful.