Programming Project 1
CSci 1913: Introduction to Algorithms,
Data Structures, and Program Development
February 8, 2017

Last revision February 20, 2017

0. Introduction.

A *mnemonic* (pronounced *nee-MON-ick*) is a simple word or phrase that helps you remember something more complex. In this project, you will write a Python program that makes mnemonics to help you remember telephone numbers. The program uses a *probabilistic algorithm* that makes decisions at random. It is not guaranteed to work for all possible telephone numbers, but it may still work often enough to be useful.

1. Theory.

The keypad of a US telephone looks something like this. It has ten keys for the digits **0** through **9**. Each key **2** through **9** has three letters of the alphabet, but the letters **Q** and **Z** are missing. The keys **0** and **1** have no letters.

The letters on the keys let you turn many telephone numbers into mnemonic words or phrases. For example, the number **686–2377** turns into the word **NUMBERS** because the key **6** has the letter **N**, the key **8** has the letter **U**, the key **6** has the letter **M** the key **2** has the letter **B**, the key **3** has the letter **E**, the key **7** has the letter **R**, and the key **7** has the letter **S**. Words like these may be easier to remember, and easier to key in, than numbers.

How could a program construct a mnemonic word for a telephone number? It might choose letters purely at random. Suppose that the program must construct a word for **686–2377**. The possible letters for **6** are **MNO**, so it might choose **M** first. The possible letters for **8** are **TUV**, so it might choose **V** next, etc. Continuing in this way, it might construct **MVNBDPR**. Unfortunately, this looks nothing like an English word, so it's not easier to remember than the number.

The program would do better if it chose letters according to how they appear in English words. Suppose that the program must still construct a word for **686–2377**. The possible letters for **6** are **MNO**, so it might choose **M**. The possible letters for **8** are **TUV**, but since **U** is more likely to follow **M** than **T** or **V**, it might choose **U**. Continuing in this way, choosing each letter based on how likely it is to follow the previous letter, the program might construct **MUNCERS**. This isn't an English word, but it looks like one, so it's easier to remember than the number.

How can a program know what letters are likely to follow other letters? One way is for it to read a list of common words, and record how their letters appear. For this project, we will use the 315-word **Dolch List,** compiled by Edward William Dolch in 1936. It's a list of common words from children's books of the early twentieth century. The Dolch List is still used to help teach reading in US primary schools.

The program will also need an algorithm that generates random integers. No algorithm can generate truly random integers, but it can generate *pseudo-random* integers that seem random, even though they're not. Python has its own random number generator, but for this project you must implement your own.

The *Park-Miller algorithm* (named for its inventors) is a simple way to generate a sequence of pseudo-

random integers. It works like this. Let N_0 be an integer called the *seed*. The seed is the first term of the sequence, and it must be between 1 and $2^{31} - 2$. Starting from N_0, later terms N_1, N_2 ..., are produced by the following equation.

$$N_{j+1} = (7^5 * N_j) \% (2^{31} - 1)$$

Here 7^5 is 16807, and 2^{31} is 2147483648. The operator `*` multiplies two integers, and the operator `%` returns the remainder after dividing one integer by another. We always get the same sequence of integers for a given seed. For example, if we start with the seed 101, then we get a sequence whose first few terms are 1697507, 612712738, 678900201, 695061696, 1738368639, 246698238, 1613847356, and 1214050682.

Some of these terms are large, but we can make them smaller by using `%` again. If t is a term from the sequence, and m is an integer greater than or equal to 1, then t `%` m is an integer between 0 and $m - 1$. We can use this fact to choose a random element from a Python sequence. If `S` is a sequence, then the Python expression `S[`*t*` % len(S)]` returns a randomly chosen element from `S`.

2. Implementation.

You must write two Python classes. The first class must be called `Random`, and it must implement the Park-Miller algorithm described in section **1**. The class `Random` must have the following methods.

`__init__(self, seed)`

(5 points.) Make an integer variable `n` and initialize it to the integer `seed`. You may assume that `seed` is in the proper range for the Park-Miller algorithm to work.

`next(self, range)`

(5 points.) Here `range` is an integer greater than 0. Use `n` to generate the next random number in the sequence. Then use the random number to compute an integer greater than or equal to 0, but strictly less than the integer `range`. Return that integer.

`choose(self, letters)`

(5 points.) Here `letters` is a string of one or more characters. Choose a single character from `letters` at random, using an index obtained by calling `next`. Return the chosen character.

The second class must be called `Mnemonic`. Its methods will generate mnemonic words for telephone numbers. The class `Mnemonic` must have the following methods.

`__init__(self, seed)`

(5 points.) Make an instance of the class `Random` called `random`, whose seed is `seed`. Also make two dictionaries, called `follow` and `letters`. These will be used by the other methods in this class.

The dictionary `follow` records how letters can follow other letters. It has lower-case letters `'a'` through `'z'` as its keys, and empty strings `''` as the initial values of those keys. We'll concatenate letters onto those empty strings later.

The dictionary `letters` tells which letters appear on which keys. It has digit characters `'0'` through `'9'` as its keys, and strings of lower-case letters as its values. The key `'0'` has the value `'z'`, and the key `'1'` has the value `'q'` (these digits do not have letters on the keypad). The key `'2'` has the value `'abc'`, and the key `'3'` has the value `'def'`, etc., as shown on the keypad.

`add(self, word)`

(10 points.) Here `word` is a string of two or more lower-case letters. The method `add` uses `follow` to record how letters appear in `word`. For example, suppose that `word` is the string `'and'`. Then `add` gets the string for `'a'` from `follow` and concatenates `'n'` to it. It also gets the string for `'n'` from `follow` and concatenates `'d'` to it. After `add` has been called on many different words, each letter in `follow` will have a string of all possible letters that can follow it. This string may have duplicate letters: they are important for making this algorithm work correctly.

```
make(self, number)
```

(10 points.) Here `number` is a telephone number, made up of digit characters. The method `make` tries to construct a mnemonic word from `number`. Let's say the word it constructs is in the local variable `mnemonic`.

It starts by getting the first digit from `number`. Then it gets a string of letters for that digit, using `letters`. It picks one of the letters from the string using `choose`. This letter is the first letter of `mnemonic`.

After that, `make` gets the remaining digits from `number`, one at a time. Each time it gets a digit, it gets the last letter from `mnemonic`. It also gets a string of letters from `follow` that can follow the last letter. It temporarily removes from that string all letters which do not appear on the same key for digit, using `letters`. If the resulting string is empty, then `make` can't continue, so it gives up and returns `''`, the empty string. However, if it is not empty, then `make` picks one of its letters using `choose` and concatenates that letter to the end of `mnemonic`.

The method `make` continues in this way until it has seen all the digits from `number`; then it returns the string in `mnemonic`.

Note that my code for `Random` and `Mnemonic` fits on about one page—Python can be very concise! If you find yourself writing many pages of Python code, then you have probably made mistakes.

3. Example.

To run the program, you must first make an instance `m` of the class `Mnemonic`, giving it a seed to start the random number sequence. Here we'll use the seed `101`.

```
m = Mnemonic(101)
```

Then you must give `m` the example words from the Dolch List by calling the method `add` on each word, like this:

```
m.add('about')
m.add('after')
m.add('again')
m.add('always')
m.add('an')
          ⋮
m.add('would')
m.add('write')
m.add('yellow')
m.add('you')
m.add('your')
```

To save space, not all the words are shown here. Python code for all these calls are available on the file **dolch.py** so you don't have to type them in. The words `'a'` and `'I'` are in the original Dolch List, but they don't appear here, because they don't say anything about how letters can follow other letters.

Finally you can call the method `make` to make a mnemonic string for a telephone number. As stated above, `make` does not always work: it sometimes gives up and returns an empty string. As a result, it's a good idea to call it a few times and see what happens.

```
print('"' + m.make('6862377') + '"')  #  "ounadrr"
print('"' + m.make('6862377') + '"')  #  "ounadrs"
print('"' + m.make('6862377') + '"')  #  "ntobers"
print('"' + m.make('6862377') + '"')  #  "ouncerr"
print('"' + m.make('6862377') + '"')  #  "ntoadrr"
print('"' + m.make('6862377') + '"')  #  "muncess"
print('"' + m.make('6862377') + '"')  #  "ounadrs"
print('"' + m.make('6862377') + '"')  #  "munadrs"
print('"' + m.make('6862377') + '"')  #  "ouncers"
print('"' + m.make('6862377') + '"')  #  "otoadrr"
```

The comments show what will be printed by each call; not all the results look exactly like English, but they're close. You should get the same output if you start with the same seed and use the same numbers. You should also test your `make` method with some other numbers to make sure that it works correctly.

4. Deliverables.

This programming project is worth 40 points. It will be due in **two weeks,** at **11:55 PM** on **February 22, 2017**. Unlike the lab assignments, you must work on this project individually, without a partner. IT MUST BE COMPLETED BY YOURSELF, ALONE. Also unlike the laboratory assignments, the TA's will read your Python code in detail, awarding partial credit where possible. As a result, you must turn in (1) Python source code for your classes `Random` and `Mnemonic`, (2) Python source code for any tests that demonstrate your classes work correctly, and (3) any output produced by those tests. You must submit one `.py` text file that contains all three things. Output from your tests must appear in comments at the end of the file.