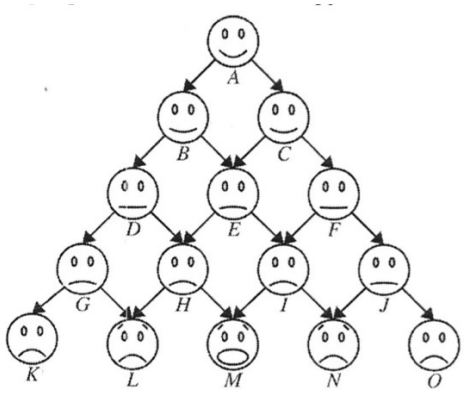
Project 6: Hashing and Caching

Problem

Expected duration: 8-10 hours



You may recall a version of this project from CS 1410. In 1410 you would have used a built-in dictionary to do caching to trade space for speed. For this assignment, you must implement your own dictionary type, which we will call HashMap. The ADT definition is given under Part 3.

A **human pyramid** is a way of stacking people vertically in a triangle. With the exception of the people in the bottom row, each person splits their weight evenly on the two people below them in the pyramid. Their total weight, however, includes both their own body weight and the weight they are supporting above them. For example, in the pyramid above, person A splits her weight across persons B and C, and person H splits his weight—plus the accumulated weight of the people he's supporting—onto people L and M. In this question, you'll explore just how much weight that is.

For simplicity we will assume that everyone in the pyramid weighs exactly 200 pounds. Person A at the top of the pyramid has no weight on her back. People B and C are each carrying half of person A's weight. That means that each of them is shouldering 100 pounds.

Now, let's look at the people in the third row. Let's begin by focusing on person E. How much weight is she supporting? Well, she's directly supporting half the weight of person B (100 pounds) and half the weight of person E (100 pounds), so she's supporting at least 200 pounds. On top of this, she's feeling some of the weight that people B and C are carrying. Half of the weight that person B is shouldering (50 pounds) gets transmitted down onto person E and half the weight that person C is shouldering (50 pounds) similarly gets sent down to person E, so person E ends up feeling an extra 100 pounds. That means she's supporting a net total of 300 pounds.

Not everyone in that third row is feeling the same amount, though. Look at person D for example. The only weight on person D comes from person B. Person D therefore ends up supporting

- half of person B's body weight (100 pounds), plus
- half of the weight person B is holding up (50 pounds), for a total of 150 pounds, only half of what E is feeling!

Going deeper in the pyramid, how much weight is person H feeling? Well, person H is supporting

half of person D's body weight (100 pounds),
half of person E's body weight (100 pounds), plus
half of the weight person D is holding up (75 pounds), plus half of the weight person E is holding up (150) pounds.

The net effect is that person H is carrying 425 pounds—ouch! A similar calculation shows that person I is also carrying 425 pounds—can you see why?

Person G is supporting

- half of person D's body weight (100 pounds), plus
- half of the weight person D is holding up (75 pounds) or a net total of 175 pounds.

Finally, let's look at person M in the middle of the bottom row. How is she doing? Well, she's supporting

- half of person H's body weight (100 pounds),
- half of person I's body weight (100 pounds),
- half of the weight person H is holding up (212.5 pounds), and
- half of the weight person I is holding up (also 212.5 pounds), for a net total of 625 pounds!

In 1410 and 2420, you have become familiar with various forms of recursive problems. The pyramid problem is an example of problems that are difficult to solve with loops and iteration, but elegantly solved with recursion. This problem is broken into 3 parts. If you did this problem in 1410, then you have already done most of this problem. If you haven't, then you'll need to do the whole thing**. For our purposes in 2420, the focus is on caching, as you will see below in Part 3.**

Keep in mind that the first and last people in each row calculate their weight differently than people in interior positions. The weight on any person can be computed recursively, with the **base case** being the person at the top the person at the top of the pyramid (in row 0), who is shouldering 0 pounds.

Part 1

Write a *recursive function* (use no loops), weight_on(r,c), which returns the weight on the back of the person in row rand and column c. Rows and columns are 0-based, so the top position is (0,0), for example, and person H is in position (3,1). The following also hold:

```
weight_on(0,0) == 0.00
weight_on(3,1) == 425.00
```

Weights should be floating-point numbers.

Part 2

When run as a main module, accept the number of rows to process via **sys.argv[1]**, and then print each row as a line at a time as `weight_on` computes them, using 2 decimals:

```
$ python3 pyramid.py 7 0.00
100.00 100.00
150.00 300.00 150.00
175.00 425.00 425.00 175.00
187.50 500.00 625.00 500.00 187.50
193.75 543.75 762.50 762.50 543.75 193.75
196.88 568.75 853.12 962.50 853.12 568.75 196.88

Elapsed time: 0.0002778119999999988 seconds Number of function
calls: 466
```

Name your file `pyramid.py`. Use `time.perf_counter` to time your main function. Save your output from this step into a file *`part2.txt`*.

Part 3

After finishing part 2, you will notice that it takes a **long time** for a large number of rows because many recursive function calls are **repeatedly** recalculating the same values--try it for 22 rows or more, for example. On one machine, processing 23 rows took 8.73 seconds and 33,554,106 function calls!

To avoid calling `weight_on` for the same row and column more than once, we want to use **caching**. Instead of recomputing a value, we will store the result in a look-up table, and only compute new values if the one we want is not in the cache.

We will save them in a **dictionary** named `cache` at the module level. The key for the dictionary is the *`tuple`* `(r,c)` and the value is the previously computed weight on the person in that `(r,c)` position. Since tuples are immutable in Python, they can be used as keys. **Big Note: the tuple type has a hash method, however you must implement your own hash function for keys.**

The first thing that `weight_on` should do, therefore, is check to see if there is a previously computed entry for the key `(r,c)` in `cache`. If there is, simply return it. Otherwise, compute the weight recursively by appropriately adding the weights of the people above it and save the result in `cache` before returning it.

With caching it took only 0.001 seconds, 782 calls to `weight_on`, and 506 cache hits to process 23 rows! A cache hit is the number of times a value is retrieved from the cache. Print out the elapsed run time, the total number of calls to `weight_on`, and the total number of "cache hits" in the following format:

```
0.00
100.00 100.00
150.00 300.00 150.00
175.00 425.00 425.00 175.00
187.50 500.00 625.00 500.00 187.50
193.75 543.75 762.50 762.50 543.75 193.75
196.88 568.75 853.12 962.50 853.12 568.75 196.88

Elapsed time: 0.00013961000000000529 seconds Number of function
calls: 70
Number of cache hits: 42
```

HashMap ADT

And now, for the drumroll: As noted at the top of this document, in 1410 you would have used a built-in dictionary to do caching. For this assignment, you must implement your own dictionary type, which we will call HashMap.

Our HashMap ADT supports the following operations:

- **get(key):** Return the value for *key* if *key* is in the dictionary. If key is not in the dictionary, raise a KeyError.
- **set(key, value):** add the (key,value) pair to the hashMap. After adding, if the load-factor
  >= 80%, rehash the map into a map double its current capacity.
- **remove(key):** Remove the key and its associated value from the map. If the key does not exist, nothing happens. Do not rehash the table after deleting keys.
- **clear:** empty the HashMap
- **capacity:** Return the current capacity--number of buckets--in the map.
- **size:** Return the number of key-value pairs in the map.
- **keys:** Return a list of keys.

Hashing Function

The hashing function you need is to maps any (r,c) tuple to a unique number. How you do that is up to you, and you have to write your own—not rely on built-in hash functions. Hint: Take advantage of the structure of the pyramid: the hashing function can be specific to **this** project, not generic.

Collision Resolution

Create your HashMap with an initial capacity of 7 buckets. The number of buckets can dynamically grow, so the capacity is not fixed, but also cannot be magically infinite. See **load factor** below. Which collision resolution strategy you use is up to you, given the restriction that the capacity cannot be infinite.

Load Factor and Rehashing

A critical statistic for a hash table is the load factor, defined as

$$f = \frac{n}{k}$$

where

- f is the load factor.
- n is the number of entries in the hash map.
- k is the number of buckets.

As the load factor grows larger, the hash table becomes slower, and it may even fail to work, depending on the method used, and the hashing function. The expected constant time property of a hash table assumes that the load factor be kept below some bound. Our solution here is to double the size of the number of buckets and rehash all the entries. **For this project rehash when the load factor is >= 80%.**
new k = 2k-1.

Using odd k increases the chances that key values and bucket size are relatively prime.

Submission

Submit in a zip file `project7.zip` in Canvas:

- Your output file for Part 2.
- Your `pyramid.py` file for Part 3 and the output file for part 3.
- Your `hashmap.py` file.

Grading (100 points)

pylint will be run on `hashmap.py`. Expected minimum score is 8.5.

Score is the sum of:

- Score on Part I x 15 points
- Score on Part II x 15 points
- Score on Part III caching results x 10 points
- Percentage of pytest test cases passed x 40 points
- min(Coding style score/8.5, 1) x 20 points
- Possible adjustment due to physical inspection of the code, to make sure you actually implemented the ADT.

Each operation defined by the ADT will be tested at least once.

FAQs

Q. If I didn't do this problem in 1410, do I really have to do Parts 1 and 2?
A. Yes. By now, it should be easier to do than it would have been in 1410. You should be able to do all three parts.

Q. Tell me again what a "cache hit" is?
A. It is when you get a result saved in the cache instead of recomputing it. So only increment that counter when that happens. You never want to recompute the weight for the same row-column combination twice.

Q. Should my numbers of function calls and cache hits match yours?
A. Absolutely. Put your counters in the right place. And remember that row and column

numbers start at zero. So, 7 rows means rows 0 through 6.

Q. What's the big deal with saving .0001 seconds?
A. Well, that's just for 7 rows. Try it for 20 rows or more, and you'll see a huge difference. The number of function calls is the driving issue here. We want to minimize those.