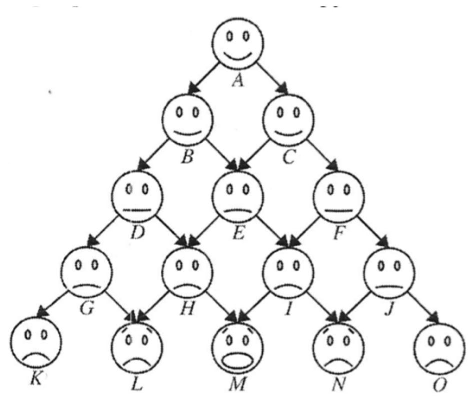
Project 2: Human Pyramids

CS 1410

Background



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 "distributed weight", however, includes not only their own body weight but 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!

The purpose of this assignment is to become familiar with **recursion** and solve a problem that is *inherently recursive* (i.e., it is difficult to figure out how to do it with loops and no recursion).

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.

Requirements

1. Write a *recursive function* (use no loops), `weight_on(r,c)`, which returns the weight on the back of the person in row `r` and 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.

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*.

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 *repeated* (try it for 22 rows or more!). For example, on my 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, 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.

   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 (the number of times the number was retrieved from the cache instead of re-evaluating it) to process 23 rows! 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: 0.0001396100000000529 seconds
Number of function calls: 70
Number of cache hits: 42
```

Turn in the final version of your code and the output for both parts (files *part2.txt* and *part3.txt*). Your function `weight_on` and your cache (which must be at the top-level and must be named `cache`) will be tested on arbitrary values.

FAQs

Q. How do you round a floating point number to 2 decimals.
A. Call the built-in function round: `round(n,2)`.

Q. But do I really have to round the weights before `weight_on` returns them?
A. No! Just make sure that when you print them, you use a format of "`.2f`" in your format string. It rounds automatically. You need to do this anyway so you get two zeroes in some cases. You don't need to use `round()` for this assignment—but you asked, so I told you!

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. How do I count the number of function calls?
A. Well, you just increment a counter every time `weight_on` is called.

Q. But where do I define my counters and initialize them to zero?
A. Well, you can't do it *inside* of `weight_on`, because it would be reinitialized to zero anytime `weight_on` is called. So, it has to be *outside* of `weight_on`. What does that tell you?

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.