CSE 101: Introduction to Computational and Algorithmic Thinking

Homework #2-3

Fall 2018

Assignment Due: Monday, December 10, 2018, by 11:59 PM

Getting Started

To help you get started on this assignment, we have given you a skeleton Python file named `game_calculations.py`. This file contains several *function stubs* and a few tests you can try out to see if your code seems to be correct. Stubs are functions that have no bodies (or have very minimal bodies). You must fill in the bodies of these functions for the assignments. **Do not, under any circumstance, change the names of the functions or their parameter lists.** The automated grading system will be looking for exactly those functions provided in `2-3.py`. **Please note that the test cases we provide you in the bare bones files will not necessarily be the same ones we use during grading!**

Directions

This assignment is worth a total of 20 points. The automated grading system will execute your solution several times, using different input values each time. Each test that produces the correct/expected output will earn 1 or more points. You must earn at least 15 points (75%) in order to pass (receive credit for) this assignment.

- At the top of the `game_calculations.py` file, include the following information in comments, with each item on a separate line:
  - your full name ***AS IT APPEARS IN BLACKBOARD***
  - your Stony Brook ID #
  - your Stony Brook NetID (your Blackboard username)
  - the course number (CSE 101)
  - the assignment name and number (Assignment #2-3)

⚠ Any functions that we tell you to complete (i.e., that we provide stubs for in the starter code) **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters, or that use different parameter names) can't be graded by our automated grading system, and may receive a grading penalty.

⚠ Be sure to submit your final work as directed by the indicated due date and time. Late work will not be accepted for grading. Work is late if it is submitted after the due date and time.

⚠ Programs that crash will likely receive a grade of zero, so make sure you thoroughly test your work before submitting it.

Game-Related Calculations

Part I: Power Distribution (10 points)

In the board game *Terra Mystica*, each player has exactly 12 power units to spend performing actions in the game. Power units are distributed across three "bowls", labeled I, II, and III (the sum of units in these bowls is always exactly 12). As the game progresses, power is gained or spent by shifting units between bowls as follows:

- To spend a unit of power, move it from bowl III to bowl I (as long as you have at least one unit in bowl III). If bowl III is empty, nothing happens.
- When a player gains a unit of power:
  - If all 12 units are already in bowl III, nothing happens. Otherwise:
  - If there is a unit in bowl I, move it to bowl II.
  - Otherwise, if bowl I is empty but there is a unit in bowl II, move one unit from bowl II to bowl III.
  - If bowls I and II are both empty, then nothing happens.

If you gain or spend multiple units at once, use a loop to process them one at a time according to these rules.

For example, consider the following sequence of actions (bowls are listed in order: I, then II, then III):

```
Initial:       5 |  7 |  0
Gain  3  ==>   2 | 10 |  0   (3 units have been moved from I to II)
Gain  6  ==>   0 |  8 |  4   (move 2 power units from I to II,
                               then the remaining 4 units from II to III)
Gain  7  ==>   0 |  1 | 11   (7 units have been moved from II to III)
Spend 4  ==>   4 |  1 |  7   (4 units have been moved from III to I)
Gain  1  ==>   3 |  2 |  7   (1 unit has been moved from I to II)
```

Complete the Python function `updatePower()`, which takes two parameters: a three-element list of integers (representing the number of units in bowls I, II, and III, in that order) and a non-zero integer representing the number of power units being gained or spent (a positive value indicates that power is being gained, and a negative value indicates that power is being spent). The function does not return anything, but directly modifies the list parameter according to the rules above. ***You may assume that the sum of the values in the list is always 12, and that the number of power units being spent will never exceed the current value in bowl III.***

Examples:

Method Call	Final List Contents
`updatePower([2, 3, 7], 4)`	`[0, 3, 9]`
`updatePower([3, 4, 5], -3)`	`[6, 4, 2]`
`updatePower([3, 3, 6], -2)`	`[5, 3, 4]`
`updatePower([2, 6, 4], 7)`	`[0, 3, 9]`
`updatePower([4, 6, 2], 11)`	`[0, 3, 9]`
`updatePower([8, 1, 3], 2)`	`[6, 3, 3]`
`updatePower([1, 1, 10], -5)`	`[6, 1, 5]`
`updatePower([0, 1, 11], -10)`	`[10, 1, 1]`

Part II: Managing Mana Costs (10 points)

The fictional computer game *Underworld Quest*[1] allows players to cast magic spells by combining specific magic words in a particular order. A valid spell consists of 2–4 magic words, drawn from specific categories. Each magic word has a *mana* (energy) cost associated with it; we can use these mana costs to calculate the total energy that a player must spend in order to cast that spell (as you would expect, the higher the mana cost, the more powerful the spell). For this problem, we are not concerned with what an actual spell does within the game; we want to determine whether a given sequence of words constitutes a valid spell, and if so, what its total energy cost is.

Magic words fall into one of four categories: power, effect, element, and range. Each category consists of six magic words, each with a different mana cost. A valid spell consists of 2–4 magic words, one from each of the categories above, with the following restrictions:

- The first magic word **MUST** be drawn from the "power" category
- The second, etc., words must be drawn from the remaining categories (effect, element, and range). Only one word can be drawn from a particular category (i.e., you can't have two words from the same category). Not every category is required, but those categories that are included must be included in the following order: effect, element, and range (meaning that a word from the "effect" category must come before any words from the "element" or "range" categories). For example, a valid three-word spell can contain words corresponding to one of the following category sequences:
  1. power, effect, element
  2. power, effect, range
  3. power, element, range

The following table lists the legal magic words, along with their individual power costs:

Power	Word	Ya	Lo	Kath	Ku	Sar	Ros
	Base Cost	1	2	3	4	5	6
Effect	Word	Des	Zo	Oh	Ven	Ew	Dain
	Base Cost	3	4	4	5	6	7
Element	Word	Vi	Ee	Ra	Mon	Pal	Neta
	Base Cost	4	5	6	7	7	9
Range	Word	Um	On	Gor	Ful	Bro	Ir
	Base Cost	2	2	3	6	6	8

The total energy cost of a spell is equal to the sum of the cost of all of its magic words, calculated according to the following formula:

- The cost of the "Power" word is just equal to its base cost
- Each other word in the spell costs $\left\lfloor \frac{(power+1) \times Base\ Cost}{2} \right\rfloor$, where `power` is the base cost of the "Power" word, and *Base Cost* is the base cost of the current magic word.

  Hint: Use the Python function `math.floor()` or simple floor division to calculate this value.

[1] Not a real game (to my knowledge), but based on/similar to various other fantasy-based computer games

For example, consider the spell "Lo Ee Gor". The "Power" word's cost is just its base cost, or 2. The cost of the "Element" word "Ee" is equal to ((2 + 1) * 5) // 2, or 7. The cost of the "Range" word "Gor" is equal to ((2 + 1) * 3) // 2, or 4. The total energy cost of this spell is therefore 2 + 7 + 4, or 13.

Complete the `spellCost()` function, which takes a single argument: a list of 2–4 strings (each string is guaranteed to be one of the words from the table above). If the elements of the list (in order) represent a valid spell according to the rules above (meaning that there is only one word per category and the categories are represented in a legal order), the function returns the total energy cost of the spell. Otherwise, the function returns the built-in Python value `None`.

As a general strategy, start by determining whether the input constitutes a valid spell:

- If the input's length is less than 2 or greater than 4, the "spell" is invalid, so return `None`.
- Otherwise, if the first word of the input is not a "Power" word, the spell is invalid, so return `None`.
- Otherwise, check to see if the remaining words occur in the correct order of categories. One way to do this is as follows:
  - Create an empty list.
  - For each input word after the first, check to see which category it belongs to (we have given you a dictionary for each category). If the word belongs to the "Effect" category, append 1 to your list. If it belongs to the "Element" category, append 2 to your list. If it belongs to the "Range" category, append 3 to your list.
  - If the elements in your list are in ascending order (**Hint:** compare your list to the one returned by Python's `sorted()` function), then the spell is valid. Otherwise, return `None` to indicate that the spell words are in an incorrect order.

If the input passes all of the tests above, calculate and return its energy cost. One way to do this is as follows:

1. Set `power_value` to the cost of the first word in the input
2. Set `total_power` equal to `power_value`
3. For each remaining word in the input:
   - Using an `if-elif-else` chain, determine which category the current word belongs to, and set the variable `cost` to its base cost.
   - Use the formula above to calculate the energy cost of the current word and add it to `total_power`

Examples:

Method Call	Return Value
`spellCost(["Sar", "Ee"])`	`20`
`spellCost(["Ku", "Ven", "Um"])`	`21`
`spellCost(["Gor"])`	`None`
`spellCost(["Ya", "Oh", "Ful"])`	`11`
`spellCost(["Kath", "Pal", "Des"])`	`None`
`spellCost(["Lo", "Oh", "Mon", "On", "Um"])`	`None`
`spellCost(["Kath", "Ven", "Pal", "Ir"])`	`43`
`spellCost(["Zo", "Des", "Pal", "Bro"])`	`None`