hw6

Reading

Chapters 6, Sections 7.1, 7.2, 7.3 of the textbook.

Programming Problems

Note: Please note that the doctest file uses `random.seed`. This guarantees that any numbers generated by the `random` module will be exactly the same each time. If you use the minimum (and correct) calls to the random function, your output should match the doctest exactly. If you make extra calls to random, then they will not match.

`interleaved`

Write a function `interleaved` that accepts two sorted sequences of numbers and returns a sorted sequence of all numbers obtained by interleaving the two sequences. Guidelines:

- each argument is a `list`
- assume that each argument is sorted in non-decreasing order (goes up or stays the same, never goes down)
- you are **not allowed** to use `sort` or `sorted` (or any other form of sorting) in your solution
- you must interleave by iterating over the two sequences simultaneously, choosing the smallest current number from each sequence.

Sample usage:

```
>>> interleaved( [-7, -2, -1], [-4, 0, 4, 8])
[-7, -4, -2, -1, 0, 4, 8]
>>> interleaved( [-4, 0, 4, 8], [-7, -2, -1])
[-7, -4, -2, -1, 0, 4, 8]
>>> interleaved( [-8, 4, 4, 5, 6, 6, 6, 9, 9], [-6, -2, 3, 4, 4, 5, 6, 7, 8])
[-8, -6, -2, 3, 4, 4, 4, 4, 5, 5, 6, 6, 6, 6, 7, 8, 9, 9]
>>> interleaved( [-3, -2, 0, 2, 2, 2, 3, 3, 3], [-3, -2, 2, 3])
[-3, -3, -2, -2, 0, 2, 2, 2, 2, 3, 3, 3, 3]
>>> interleaved( [-3, -2, 2, 3], [-3, -2, 0, 2, 2, 2, 3, 3, 3])
[-3, -3, -2, -2, 0, 2, 2, 2, 2, 3, 3, 3, 3]
>>> interleaved([1,2,2],[])
[1, 2, 2]
>>> interleaved([],[1,2,2])
[1, 2, 2]
```

```
>>> interleaved([],[])
[]
>>> interleaved( list(range(-2,12,3)), list(range(20,50,5)) )
[-2, 1, 4, 7, 10, 20, 25, 30, 35, 40, 45]
>>> interleaved( list(range(20,50,5)), list(range(-2,12,3))  )==[-2, 1, 4,
7, 10, 20, 25, 30, 35, 40, 45]
True
```

`primeFac`

(Perkovic, Problem 5.42) Write a function `primeFac` that computes the prime factorization of a number:

- it accepts a single argument, an integer greater than 1
- it returns a list containing the prime factorization
  - each number in the list is a prime number greater than 1
  - the product of the numbers in the list is the original number
  - the factors are listed in non-decreasing order

Sample usage:

```
>>> primeFac(5)
[5]
>>> primeFac(72)
[2, 2, 2, 3, 3]
>>> primeFac(72)==[2, 2, 2, 3, 3]
True
>>> [ (i,primeFac(i)) for i in range(10,300,23)]
[(10, [2, 5]), (33, [3, 11]), (56, [2, 2, 2, 7]), (79, [79]), (102, [2, 3,
17]), (125, [5, 5, 5]), (148, [2, 2, 37]), (171, [3, 3, 19]), (194, [2,
97]), (217, [7, 31]), (240, [2, 2, 2, 2, 3, 5]), (263, [263]), (286, [2, 11,
13])]
>>> [ (i,primeFac(i)) for i in range(3,300,31)]
[(3, [3]), (34, [2, 17]), (65, [5, 13]), (96, [2, 2, 2, 2, 2, 3]), (127,
[127]), (158, [2, 79]), (189, [3, 3, 3, 7]), (220, [2, 2, 5, 11]), (251,
[251]), (282, [2, 3, 47])]
```

`piggyBank`

Write a function `piggyBank` that accepts a sequence of "coins" and returns counts of the coins and the total value of the bank.

- the coins will be given by a list of single characters, each one of `'Q','D','N',P'`, representing quarter, dime, nickel, penny
- the function returns a tuple with two items:
  1. a dictionary with counts of each denomination. They should always be listed in the order `'Q','D','N',P'`
  2. the total value in whole cents of the amount in the bank

Sample usage:

```
>>> piggyBank(['D', 'P', 'Q', 'Q', 'D', 'P', 'P'])
({'Q': 2, 'D': 2, 'N': 0, 'P': 3}, 73)
>>> piggyBank(['D', 'D', 'N', 'N', 'N'])
({'Q': 0, 'D': 2, 'N': 3, 'P': 0}, 35)
>>> piggyBank(['P', 'D', 'N', 'P', 'N', 'N', 'N', 'Q', 'D', 'P', 'Q', 'Q', 'D'])
({'Q': 3, 'D': 3, 'N': 4, 'P': 3}, 128)
>>> piggyBank(['D', 'P', 'Q', 'Q', 'D', 'P', 'P'])==({'Q': 2, 'D': 2, 'N': 0, 'P': 3}, 73)
True
```

`craps`

(Perkovic, Problem 6.31a) Craps is a single player dice game, that proceeds as follows:

1. the player rolls 2 six-sided dice once
   - if the total is 7 or 11, the player wins
   - if the total is 2, 3 or 12, the player loses
   - otherwise, the game continues, ... see 2 ...
2. the player the continues to roll the dice repeatedly, until ...
   - the total is the same as the original total (from 1), in which case the player wins
   - the total is 7, in which case the player loses

Write a function `craps` that simulates a single game of craps (may be many rolls) and returns `1` if the player wins and `0` otherwise.

```
>>> import random
>>> random.seed(0)
>>> craps()
0
>>> random.seed(1)
>>> craps()
1
>>> random.seed(2)
>>> craps()
0
>>> [ (i,random.seed(i),craps()) for i in range(20)]
[(0, None, 0), (1, None, 1), (2, None, 0), (3, None, 1), (4, None, 0), (5, None, 1), (6, None, 0), (7, None, 1), (8, None, 0), (9, None, 0), (10, None, 1), (11, None, 1), (12, None, 1), (13, None, 1), (14, None, 0), (15, None, 0), (16, None, 1), (17, None, 0), (18, None, 0), (19, None, 1)]
```

`testCraps`

(Perkovic, Problem 6.31b) Write a function `testCraps` that accepts a positive integer `n` as an input, simulates `n` games and craps, and returns the fraction of games the player won.

- the `testCraps` function should *not* make dice rolls directly, instead ...
- `testCraps` function should *call* the `craps` function repeatedly and keep track of the results
- if you your `craps` and `testCraps` function simulate the game correctly without any *extra* rolls, you should be able to hit the results below *exactly*

```
>>> random.seed(0)
>>> testCraps(10000)
0.5
>>> random.seed(1)
>>> testCraps(10000)
0.4921
>>> [(i,random.seed(i),testCraps(100*i)) for i in range(1,10)]
[(1, None, 0.49), (2, None, 0.46), (3, None, 0.47333333333333333), (4, None,
0.5125), (5, None, 0.476), (6, None, 0.47333333333333333), (7, None,
0.4514285714285714), (8, None, 0.48), (9, None, 0.4855555555555556)]
>>>
```