Decoding Button Specification

This document contains useful information on how to decode a RECEIVER message. Each stage has 3 coding challenges, each of which will give points. You need a total of 15 points to decode the receiver message. Successfully decoding the RECEIVER message will allow alliances to call in an additional PIONEER, as outlined in the game manual.

Download the starter code here. Write your code under each function, then run `python3 challenges.py` to test your code. These are just sanity checks; feel free to add your own tests to the file.

When you are ready to submit, have one team member do the following

- Go to gradescope.com and sign up as a student. The course entry code is BPJRZZ and the school is UC Berkeley. Your student ID can be any 10 digit number.
- Submit challenges.py to the assignment. It must be named challenges.py.
- After a few minutes, the autograder will give you your official score. If you run into any issues, let us know as soon as possible so we can help you submit.

Level 1 (2 pts each)

Is Palindrome?

```
def is_pal(word):
```

PiP (short for Pelipiningo) is an undercover FBI agent that is being interrogated by Aliens on the planet, Planet 14. In order to escape the interrogation, PiP made an agreement with the Aliens, that if PiP is able to identify if strings are palindromes, they will set him free without interrogating PiP. However, if PiP is unable to identify if strings are a palindrome, they will forever lock PiP in a tickle room full of feathers from a legendary Hippogriff.

But, here's the catch: PiP doesn't know how to read! So, PiP cannot identify if a string is a palindrome. On the bright side however, PiP is able to contact a Pioneer (that's you) to assist PiP on this dilemma. Please help PiP out so he will not be locked in the tickle-torture cellar!
Input:

- `word`: the input string

Output:

- A boolean indicating if the word was a palindrome or not

Examples:

```
>>> is_pal("cars")
False
>>> is_pal("tacocat")
True
>>> is_pal("aa")
True
>>> is_pal("abaa")
False
>>> is_pal("tacobcat")
False
```

Adding Up Evens

```
def add_evens(nums):
```

The aliens now have a list of each of their favorite numbers and want PiP to add them all up and get their collective favorite number. However, PiP is deathly afraid of odd numbers and only wants to add up the even indices of this list. Recall that arrays are indexed starting with 0, and 0 is an even number.
Input:

- `nums`: the input integer

```
>>> add_evens([5])
      5

 >>> add_evens([1, 2, 3, 4, 5])
      9

 >>> add_evens([-4, 3, 0, 2])
      -4
```

- An integer indicating the sum of the integer's even-index digits

Adding Up Evens Follow-up

```
def add_evens(nums):
```

The aliens, impressed with PiP, now give him a single number instead of an array. PiP is still afraid of odd numbers, so now they has to add up the even index digits of this input number (instead of an array). Note, for a number the rightmost digit is at the 0th index.

For example, given the number 123: 3 is in index 0, 2 is index 1, and 1 is in index 3.

```
>>> add_evens(5)
  5
  >>> add_evens(12345)
  9
  >>> add_evens(4302])
  5
```

Level 2 (3 pts each)

Cheese

```
def can_cheese(small: int, big: int, goal: int) -> bool:
```

One day, Sam decides to bathe in cheese. For the amount of cheese necessary, we want to make a bucket of cheese that is `goal` kilograms. We have a number of `small` cheese (1 kilo each) and `big` cheese (5 kilo each). Return `True` if it is possible to make the goal by choosing from the given cheeses and otherwise `False`. This is a little harder than it looks and can be done without any loops.

Input:

- `small`: number of `small` cheese
- `big`: number of `large` cheese
- `goal`: the target weight

Output:

- A boolean whether we can make the target weight or not

```
>>> can_cheese(3, 1, 8)
True
>>> can_cheese(3, 1, 9)
False
>>> can_cheese(3, 2, 10)
True
>>> can_cheese(1, 3, 12)
False
>>> can_cheese(13, 1, 13)
True
>>> can_cheese(14, 0, 13)
True
```

Cheese Followup

```
def can_cheese(small, small_size, big, big_size, goal) -> bool:
```

Consider the same problem as before but now the size of the small cheese and the size of the big cheese are now inputs to the function.

```
 >>> can_cheese(3, 3, 5, 5, 27)
   True

>>> can_cheese(3, 3, 5, 5, 28)
   True

>>> can_cheese(3, 3, 5, 5, 32)
   False
```

Min of the Maxes

```
def min_of_maxes(nums: int, k: int) -> int:
```

PiP is exploring the moon and finds martians holding hands in a line. They cannot let go of the hands they are holding and the line must be kept perfectly straight as per tradition. Each Martian holds in their pockets some number of marbles. PiP checks `k` sequential Martians in a row at a time. In this group of `k`, he finds the martian with the most number of marbles and writes that number down in his notepad. PiP checks all possible groups of `k` sequential martians from the line of martians. Back at base, PiP looks at all the numbers he wrote down and reports the smallest number out of those to his boss, BiP.

Input:

- `nums`: array of numbers
- `k`: the size of the sequential martians PiP can check in `nums`

Output:

- the smallest value out of all the largest values of the `k`-sized subsets of martians

```
>>> min_of_maxes([1,3,-1,-3,5,3,6,7], 3)
  3

>>> min_of_maxes([1,3,-1,-3,5,3,6,7], 1)
  -3
```

Level 3 (4 pts each)

Astronaut Acquaintance

```
def space_acquaintance(space_from: list, space_to: list) -> int:
```

Astronauts in space want to gather together because they are lonely. Given two lists of equal length where astronauts in corresponding indices know each other (ie: [A, B], [C, D] where A and C know each other and B and D know each other), return the min acquaintance sum from all trios. If no trio exists, return -1.
Definitions:
Trio: a group of three astronauts where each astronaut knows each other.
Acquaintance sum: the number of astronauts a particular trio knows outside of the current trio. We count a trio as knowing an astronaut if one or more members of the trio know the astronaut.
Input:

- `space_from`: first list of astronauts where each astronaut knows the astronaut in the corresponding index in `space_to`
- `space_to`: second list of astronaut where each astronaut knows the astronaut in the corresponding index in `space_from`

Output:

- the min acquaintance sum of all astronaut trios

```
>>> space_acquaintance([1, 2, 2, 3, 4, 5], [2, 4, 5, 5, 5, 6])
  3

>>> space_acquaintance([1, 4, 4, 2, 5, 6, 7, 2], [3, 1, 3, 5, 6, 7, 5, 6])
  0
```

Shortest Path

```
def shortest_path(graph: dict, A: int, B: int) -> int:
```

Alex is in space station A, and needs to get to a robotics competition at space station B. Unfortunately, Google Space Maps TM is giving incorrect paths due to malicious people on the way. Luckily, each station is connected to each other by a series of other space stations. Find the shortest path for Alex assuming the cost from traveling from adjacent space stations is one.

- there will always be a path from space station A to space station B
- the cost of traveling to adjacent space stations is 1

In this question, the space station graph will be represented by a dictionary where the key is a particular space station, and the value will be a list of other space stations which there is a path to.
Input:

- `graph`: the dictionary representation of the graph
- `A`: start node
- `B`: end node

Output:

- the length of the shortest path from `A` to `B`

```
>>> shortest_path({1:[2], 2:[]}, 1,2)
  1

>>> shortest_path({1: [2, 3], 2: [1, 3, 8], 3: [1, 2, 4, 8], 5: [3, 6], 6: [5, 7], 7: [6, 8], 8: [2, 7]}, 1, 1)
  0

>>> shortest_path({1: [2, 3], 2: [1, 3, 8], 3: [1, 2, 4, 8], 5: [3, 6], 6: [5, 7], 7: [6, 8], 8: [2, 7]}, 1, 8)
  2

>>> shortest_path({1: [2, 3], 2: [1, 3, 8], 3: [1, 2, 4, 8], 5: [3, 6], 6: [5, 7], 7: [6, 8], 8: [2, 7]}, 4, 7)
  3
```

Uppercasing

```
def longest_uppercase(input, k):
```

Terrie has a string of characters. She has the ability to turn uppercase `k` of the letter types. What is the length of the longest substring of uppercase characters that Terrie can create?
Example:
`k` = 2
input string: “aabbakaa”
To make the max uppercase substring I choose to capitalize the letter types “a” and “b”.
This results in the string “AABBAkAA”.
The longest uppercase substring is therefore “AABBA” which is of length 5.

Input:

- `input`: the input string
- `k`: the number of letter types that Terrie can turn uppercase

Output:

- the length of the longest substring of uppercase characters

```
>>> longest_uppercase("aaabbcajnnaddgfjn", 2)
  5

>>> longest_uppercase("aaabbbcajnnnnaddgfjn", 1)
  4

>>> longest_uppercase("aaabbbcajnnnnadddgfjn", 4)
  9
```

Extras (0 pts each)

Eating Ice Cream

```
def max_tastiness(ice_creams: list) -> float:
```

PiP's 2nd favorite food is ice cream, so on long trips, he always brings 1 of each of his favorite flavors with him. However, PiP forgot to bring a freezer this time. The ice cream immediately starts to melt, so PiP needs to hurry and enjoy as much as possible before it's all gone.

Each ice cream the PiP brought has an associated tastiness (t) and melting factor (f); when PiP eats one, PiP enjoys t tastiness, but all the other ice creams melt a bit (reducing their tastiness by a factor of f). PiP can eat the ice creams in any order; if PiP eats the ice creams in the optimal order, when is the maximum total tastiness that PiP can enjoy?

Each tastiness will be in the range [0,100] and each melting factor will be in the range [0,1]. All outputs will be rounded to 5 decimal places.
Example: [(4, .9), (6, .5)] - If PiP eats 4 first, then the tastiness will be 4 + 6*0.9 = 9.4. If PiP eats 6 first, then the tastiness will be 6 + 4*0.5 = 8. The max tastiness is therefore 9.4.

Input:

- `ice_creams`: A list of pairs (tastiness, melting factor).

Output:

- the maximum achievable total tastiness. Only needs to be accurate up to 5 decimal places.

```
>>> round(max_tastiness([(4,.9), (6,.5)]), 5)
  9.4

>>> round(max_tastiness([(10,0.1),(5,0.1),(4,0.2),(2,0.6)]), 5)
  10.544

>>> round(max_tastiness([(0,1),(1,1),(0,0),(1,0),(5,0.4),(4,0.5),(2,0.7)]), 5)
  8.14
```

```
>>> ar = [((0.003*x*x*x + 0.001*x*x + 4.159)%100, (0.00265*x*x)%1) for x in range(10000)]
>>> round(max_tastiness(ar), 5)
   7534.58121
```

Lifeguard Budget

```
def lifeguard_budget(intervals: list) -> int:
```

Due to terrible work conditions, all the current lifeguards have quit. The owner, trying to save the business, has employed a number of lifeguards. There exists a list of underpaid potential lifeguards, and the times they can be on duty, represented as a tuple (start_minute, end_minute). If we hire a lifeguard, we need to pay them $1/minute for the lifeguard's whole interval (no partial hires). What is the least amount of money we can spend to cover the interval (0, 1440)?

- all intervals include start but exclude end. For example, (0,5) means 0,1,2,3,4.
- start_minute ≤ end_minute for each lifeguard
- all input times are integers
- there will always be a possible solution

Input:

- `intervals`: A list of pairs (start_minute, end_minute) for each lifeguard

Output:

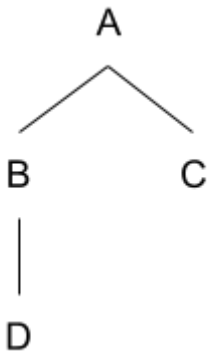
- the minimum amount (in dollars) that the pool owner can spend

```
  >>> lifeguard_budget([(0,1000),(0,500),(500,1440)])
1440

  >>> lifeguard_budget([(0,1000),(0,500),(501,1440)])
1939

  >>> lifeguard_budget([(a*5-7,a*5) for a in range(400)])
2016

>>> lifeguard_budget([(a*6-10,a*6-2) for a in range(400)][::-1])
1928
```

Tree Validation

```
def largest_valid_tree(edge_string: str) -> int:
```

Jon made the alphabet into a tree! At least, that's what he claims. To prove it, Jon will give you the edges of the tree as a long string. For example, for this tree:



Jon would give you the string "AB AC BD". Each pair of letters is a directed edge, and the string is space-delineated. Not all the letters of the alphabet will be included in the tree, just the ones that Jon remembers.
If Jon's list of edges is a valid tree, return the number of edges. Otherwise, return the size of the largest subset of edges that does form a valid tree. In a valid tree:

- there is only one root
- all nodes beside the root have one parent
- there are no cycles

Input:

- `edge_string`: the input string

Output:

- the number of edges in the largest valid subtree

```
>>> largest_valid_tree("AB AC BD")
  3

>>> largest_valid_tree("AB BC CA")
  2
```