Balchunas

AP CSP: Problem Set 2

Specification

Submit a .py file named `your_name_pset2.py`. Your file should **NOT** execute any functions. You are just filling them in. When I run your program there should be no errors and no output. You may choose to test your functions as you go, but you may delete the function calls after you test or comment them out. Comment your problems with the problem number above the function header. Remember to import the random module to use the library's functions.

Problem 1

Given a list, write a function that returns a reversed version of that list. Do **NOT** use the `reverse()` method provided by Python.

```
reverse_list([1,2,3]) # returns [3,2,1]
```

Problem 2

Linear search, or *sequential search*, is an algorithm that searches through a list for a specific key. The index of the first discovered instance of the key in the list is returned, and -1 is returned if the key is not found. Given a list and a key, implement your own version of a linear search. Do **NOT** use the `index()` method provided by Python.

```
lin_search([1, 2, 3], 4)    # returns -1
lin_search([1, 2, 3], 3)    # returns 2
lin_search([1, 2, 3, 3], 3) # returns 2
```

Problem 3

Binary search is an algorithm that searches through a **SORTED** list for a specific key. The index of the first discovered instance of the key in the list is returned, and -1 is returned if the key is not found. A binary search is much more efficient than a linear search. The binary search algorithm looks at the middle element of a list. If the element is the key, that index value is returned, otherwise you look at the first half or second half of the list, depending on the location of the key in relation to the element you just looked at. You then look at the middle of that list and repeat until you run either find the key, or you run out of elements to check. Do **NOT** just blindly copy an implementation from the internet or from class.

```
bin_search([1, 2, 3, 4, 5], 2)      # returns 1
bin_search([99, 101, 111, 120], 88) # returns -1
```

Problem 4

Selection sort is one of the most fundamental sorting algorithms. The algorithm is at follows. Start at the first element of a list and search the whole list for the smallest element. Swap that element with the element at the 0-th index. Move to the second element and search for the smallest element between that and the end of the list. This should be the second-smallest element in the list, so swap that with the second element. Repeat until all elements have been sorted. Given a list, sort the list in-place (do not return a value, the function itself will change the list since lists are *mutable* in Python).

```
selection_sort([3, 4, 1, 2]) # list becomes [1, 2, 3, 4]
```

Problem 5

Bubble sort is one of the most fundamental sorting alogrithms. The algorithm is as follows. Loop through the list element-by-element comparing the current element to the one in front of it, swapping their values if needed. These passes are completed until there are no swaps have to be performed during a pass, meaning the list has become fully sorted. Add the optimization that the i-th iteration of the comparison loop does not need to look at the last i elements. Given a list, sort the list in-place (do not return a value, the function itself will change the list since lists are *mutable* in Python).

```
bubble_sort([3, 4, 1, 2]) # list becomes [1, 2, 3, 4]
```

Problem 6

Insertion sort is one of the most fundamental sorting algorithms and in some cases can be much much faster than selection sort and a poorly written bubble sort. The algorithm is as follows. Starting at the k-index element, you will swap the element forwards until the list is appropriately sorted up to the k-th index. Start at the second element (k = 1) and stop when the entire list is sorted. In other words, you insert the specific element where it goes within the first k elements of the list. Eventually you perform this swapping algorithm with the final element (k = len(list) - 1) and every element must be in its proper location. Given a list, sort the list in-place (do not return a value, the function itself will change the list since lists are *mutable* in Python).

```
insertion_sort([3, 4, 1, 2]) #returns [1, 2, 3, 4]
```

Problem 7

Given a list of integers, return the number of integers greater than or equal to some threshold.

```
count_ints([1, 2, 3, 4, 5], 3) # returns 3
```

Problem 8

Given a list of numbers and a threshold, return a list that is full of asterisks and empty spaces where an asterisk means the number in the original list was smaller than the threshold and the space means the number in the original list was larger than (or equal to) the threshold.

```
list_mask([1, 2, 2, 4, 1, 3], 2) # returns ["*", "", "", "", "*", ""]
```

Problem 9

Masking is an important problem in computer vision applications where a program analyzes images. Here we'll create an *and* map. A mask will have either asterisks or empty strings as its elements. Given a list of integers and a mask, return a new list of integers where the new list's elements are equal to the original list's elements where the mask is a "*" and 0 otherwise.

```
img_mask([1, 2, 3, 4], ["*", "", "", "*"]) # returns [1, 0, 0, 4]
```

Problem 10

Write a function that takes a list of integers and returns a tuple with 2 elements. The first element is the minimum value in the input list and the second element is the maximum value in the input list. Do *NOT* sort the list.

```
min_max([1, 10, 42, 2]) # returns (1, 42)
```

Problem 11

Given a list of numbers, return the average of the numbers. The average of a list of numbers is calculated by adding all of the elements together and dividing by the total number of elements. The fancy *summation notation* for this algorithm is: $\frac{1}{N}\sum x_i$ where x_i is each number in the list and N is the number of elements in the list.

```
get_avg([1, 2, 3, 4]) # returns 2.5
```

Problem 12

Given a list of integers, return the standard deviation of the numbers in the list. The standard deviation is calculated by taking the square root of the variance. The variance is calculated by finding the average value of the distance of each point from the average value squared. In other words, $\frac{1}{N-1}\sum(x_i - \bar{x})^2$ where x_i is each number in the list, $\bar{x}$ is the average value of the numbers in list x, and N is the total number of numbers in the list x. If you need help understanding this notation, compare it to the previous problem's algorithm with which you are probably already familiar.

```
get_stdev([1, 2, 3, 4]) # returns 1.118
```

Problem 13

Write a function that takes a list and computes the $alternating\ sum$ of all its elements. An alternating sum alternates between adding and subtracting elements. The example returns -2 because $1 - 4 + 9 - 16 + 9 - 7 + 4 - 9 + 11$. Your function should return the alternating sum of a list.

```
alt_sum([1, 4, 9, 16, 9, 7, 4, 9, 11]) # returns -2
```

Problem 14

A *set* is a group of elements which contains no duplicates. Write a function which takes a list and returns a list with all duplicate elements removed.

```
setify([1, 2, 3, 4, 3, 2, 1])        # returns [1,2,3,4]
setify(["abc", "abc", "def", "ghi"]) # returns ["abc", "def", "ghi"]
```

Problem 15

A *set* is a group of elements which contains no duplicates. Write a function which takes a list (that's already guaranteed to be a set) and a value to add to the set. Return a list with the new set. Only add the value if the value is not already in the set.

```
set_add([1, 2, 3, 4], 3) # returns [1, 2, 3, 4]
set_add([1, 2, 3, 4], 5) # returns [1, 2, 3, 4, 5]
```

Problem 16

The *union* between two sets is the resulting set of adding two sets together. Write a function which takes two lists (already guaranteed to be sets) and returns a list which is the union of the two sets.

```
set_union([1, 2, 3, 4], [1, 2, 3, 6]) # returns [1, 2, 3, 4, 6]
set_union([1, 2, 3], [1, 2, 3])       # returns [1, 2, 3]
```

Problem 17

The *intersection* of two sets is the resulting set of the overlap in elements between the two sets. Write a function that takes two lists (already guaranteed to be sets) and returns a list which is the intersection of the two sets.

```
set_intersection([1, 2, 3, 4], [1, 2, 5, 6]) # returns [1, 2]
set_intersection([1, 2, 3], [4, 5, 6])       # returns []
```

Problem 18

A set is a *subset* of a set if every element in the subset is also an element in the set. Given two lists (already guaranteed to be sets), return `True` if the second list is a subset of the first list and `False` otherwise.

```
is_subset([1, 2, 3], [1, 3]) # returns True
is_subset([1, 2, 3], [4, 5]) # returns False
```

Problem 19

The *difference* of two sets `A` and `B` is `A - B`. The result is a set with all the elements of `A` which are not in `B`. Given two lists (already guaranteed to be sets), return a list which is the difference of the first and second sets.

```
set_difference(A, B)
set_difference([1, 2, 3], [1, 2, 4]) # returns [3]
set_difference([1, 2, 3], [4, 5, 6]) # returns [1, 2, 3]
set_difference([1, 2, 3], [1, 5, 6]) # returns [2, 3]
```

Problem 20

Sometimes we want to know the *symmetric difference* between two sets. That is all the elements in `A` and `B` that are not in both `A` and `B`. Given two lists (already guaranteed to be sets), return a list which is the symmetric difference between the two sets.

```
set_symmetric_diff(A, B)
set_symmetric_diff([1, 2, 3], [4, 5, 6]) # returns [1, 2, 3, 4, 5, 6]
set_symmetric_diff([1, 2, 3], [1, 2, 4]) # returns [3, 4]
```

Problem 21

Write a function that shuffles a list. In other words, the function takes a list and returns a list with the same elements, but in a different order. *Note: Shuffling is not a hard problem, but shuffling well is tricky. Bad shuffling algorithms were rampant in the early days of online gambling and led to a lot of exploitation! Read about the Fisher-Yates shuffle to learn more.*

```
shuffle_list(L)
shuffle_list([1,2,3])  # returns [2,3,1] (maybe)
```

Problem 22

You implement your shuffling algorithm to make a playlist for a party containing `N` songs. You want to know what is the probability that the shuffled playlist contains at least one consecutive song (i.e. song 2 is followed by song 3). Create a list of songs labeled `1` to `N`, shuffle the playlist and check to see if any songs are consecutive. The function returns `True` if the shuffled playlist contains such a song, and `False` otherwise.

```
consecutive_song(N)
consecutive_song(3) # returns True  if shuffle is like [2,3,1]
consecutive_song(3) # returns False if shuffle is like [2,1,3]
```

Problem 23

Using the result from the previous problem, write a Monte Carlo simulation that determines the probability of having a consecutive song in your shuffled playlist. The parameter is the number of songs in the playlist. *Investigate how this probability depends on changing* `N`

```
con_song_prob(N) # returns a number between 0 and 1
```

Problem 24

Given an list of integers, return `True` if and only if it is a valid *mountain list*. A list is a mountain list if and only if:

- The length is greater than or equal to 3
- There exists some index i such that li[0] $<$ li[1] ... $<$ li[i]
- And for the same index i, li[i] $>$ li[i + 1] ... li[length - 1]

```
is_mountain_list(L)
is_mountain_list([1, 2, 3, 3, 2, 1]) # returns False
is_mountain_list([0, 3, 2, 1])       # returns True
is_mountain_list([3, 5, 5])          # returns False
is_mountain_list([1, 2, 3, 2, 4, 0]) # returns False
```

Problem 25

There are `N` hotels along the beautiful Lake Erie coast. Each hotel has its own value in dollars. You win an amount of dollars in the lottery and want to buy a sequence of consecutive hotels such that the sum of these consecutive hotels is as great as possible, but not larger than what you've won. Return the greatest possible total value.

```
largest_hotel_value(L, amount)
largest_hotel_value([2, 1, 3, 4, 5], 12) # returns 12
```

```
largest_hotel_value([7, 3, 5, 6], 9)     # returns 8
```

Problem 26

`N` students decide to celebrate finishing a CSP problem set by sneaking into the school after hours to have a party. All `N` students are holding a can of soda. At one point the lights go out and the fire alarm goes off. Every student puts down their drink on a table in the cafeteria. The alarm turns off after a while and the students re-enter the building. Unfortunately the lights are still off and everyone is very thirsty. Everyone throws caution to the wind and grabs a random can! *What are the odds at least one student gets his or her original soda?* Write a program that takes one argument `N` that represents the number of students attending the party, runs 1000 simulations of the event, and returns the fraction of times at least one guest gets his or her original soda. As `N` gets large does this fraction approach 0, 1, or something else?

```
my_soda(N) # returns a number between 0 and 1
```

Projects

Project 1

Write a function that simulates a game of Mastermind. Generate 4 random numbers 0-9, and have a user guess the combination. Numbers guessed correctly and in the correct location should show up, and numbers guessed correctly but in the wrong location should be replaced with an asterisk. Numbers not inside the combination should not produce any output. Give the user 10 guesses before game over and don't let the person make the same guess twice.

```
play_mastermind()
ex. if code is 1234
Code: _ _ _ _
guess: 2542
Code: _ * _ * (because 2 and 4 were guessed but in wrong order)
guess: 1245
Code: 1 2 _ *
guess: 1243
Code: 1 2 * *
guess: 1234
Code: Right!
```

Project 2

Write a function that simulates a game of Wordle. Pick a random word from a word list (represented with a list of strings) that a user has to guess. Give the user 5 guesses to guess the word correctly. Display the letter where the user's guess is correct, and a * for a letter that is in the word, but not in the correct place.

```
play_wordle()
Word: _ _ _
Guess: toe
Word: t _ _
Guess: tea
Word: t _ *
Guess: tap
Word: t a _
Guess: tab
Word: Correct!
```

Audio Files

We learned earlier in the year that audio data can be represented by numerical values in sequence. Here we have a Python module that allows you to load an audio file and manipulate it as if it were a list. Follow the instructions in the README to manipulate some audio files. Submit the code you used for the problems below. Provide the audio files you used inside of a zip folder.

Problem 1

Adjust the volume of just one portion of a song of your choosing. Make it louder in some parts and quieter in others.

Problem 2

Adjust the volume of the song to mute the song for a fixed interval. Do not mute the entire song.

Problem 3

Write an algorithm that plays a portion of a song backwards while leaving the rest of the song untouched.

Problem 4

Write an algorithm that starts a song off very quiet and slowly ramps up the volume to its maximum volume.

Problem 5

Write an algorithm that cuts out a verse or chorus of a particular song.

Problem 6

A *triad* in music is three notes played at the same time. Some of these sound nice to our ear, and some do not. Using the note samples I provided (or make your own!) make an audio file consisting of the notes C, E, G played at the same time.

Problem 7

Repeat the previous problem but with the notes A, C, and E.

Problem 8

Write a program that manipulates audio files in a creative way. You may choose to use the frequency domain for this, for example to remove unwanted frequencies or to boost others. This is not a requirement, but going to the frequency domain allows you to do some cool transformations.