University of Ottawa Computer Science Club

CS Games 2024

Saturday February 10th, 2024

Algorithms Challenge

Group Size: Individual

Languages: Python

Time: 3 hours

Difficulty: Easy to Hard

Introduction

This challenge is a set of small algorithm problems. The problems range from easy to hard. They are to be solved using *Python*, ideally *Python 3*. Many of these problems are designed to help people be introduced to python as a language and to work on fast algorithm design.

The snippets use Python's type annotations. You might need to add the following to your imports section.

```
from typing import List, Dict, Callable, Tuple
```

... or you can remove the type annotations.

Easy Algorithms

Unique Characters

Given a string, determine if the string has all unique characters. Assume the string can include any character from the ASCII set.

Your function should return *True* if all characters are unique, and *False* otherwise.

Goal: $O(n)$ or less

```
def unique_chars(s: str) -> bool:
    pass
```

Sum of Digits

Given a non-negative integer, return the sum of its digits until the sum is a single digit.

Goal: 1 line

```
def sum_of_digits(number: int) -> int:
    pass
```

Example:

```
>>> sum_of_digits(987)  # 9+8+7 = 24 -> 2 + 4 = 6
6
```

Has Prime Factors

Given an integer number and a list of prime numbers, determine if the given number has at least one of the primes as a factor. Your function should return *True* if the given number has it as a prime number and *False* otherwise.

Goal: 1 line

```
def has_prime_factor(number: int, primes: List[int]) -> bool:
    pass
```

N-th of Prime Factors

Given an integer n and a list of prime numbers, determine the n-th number (starting at 1) that has at least one of the given primes as a factor.

Hint: Complete the problem above.

Goal: 2 lines or less

```
def nth_of_prime_factor(n: int, primes: List[int]) -> int:
    pass
```

Find First and Last Position of Element in Sorted Array

Given a sorted array of integers *nums* and a target integer *target*, write a function to find the starting and ending position of a given target value.

If the target is not found in the array, return $[-1, -1]$.

Goal: $O(\log n)$.

```
def search_range(nums: List[int], target: int) -> List[int]:
    pass
```

Example:

```
>>> nums = [5, 7, 7, 8, 8, 10]
>>> target = 8
>>> search_range(nums, target)
[3, 4]
```

Example:

```
>>> nums = [5, 7, 7, 8, 8, 10]
>>> target = 6
>>> search_range(nums, target)
[-1, -1]
```

Find Smallest Invalid

Given a range of integers from 1 to n, where some numbers are *valid* and others are *invalid*, identify the smallest *invalid* number. The rule is that if a number is *valid*, then all numbers before it are also *valid*. In other words, once you encounter the first *invalid* number, all subsequent numbers are guaranteed to be *invalid*.

Unlike the previous problem, here you do not initially know which numbers are valid or invalid. However, you are provided with a 'validator' function: given a number, it returns *True* if the number is valid and *False* if it is invalid.

Your task is to determine the smallest *invalid* number within the range from 1 to n. If all numbers are *valid*, return *None*.

Goal: $O(\log(n))$

```
def find_smallest_invalid(n: int, validator: Callable[[int], bool]) -> Optional[int]:
    pass
```

Note on 'validator': It is a function that you can call with an integer argument. It returns a boolean indicating whether the input is valid.

Here's how you might test your function:

```
>>> n = 20  # Define the range
>>> validator = lambda x: x < 15  # Define validity: numbers less than 15 are valid
>>> find_smallest_invalid(n, validator)
15
```

Pattern Matching

Given a sequence of alphabet letter which represent a *pattern* and a string containing words of 1 or more alphabet letters separated by a space, determine if the general pattern can be found within the string of words. Your function returns *True* if the pattern can be found.

Each letter in the *pattern* is to represent a word within the given *sentence*.

```
def pattern_matching(pattern: str, sentence: str) -> bool:
    pass
```

Example 1:

```
>>> pattern_matching("abca", "apple orange tomato apple")
True
```

Explanation:

```
a = apple
b = orange
c = tomato
abca -> a b c a -> apple orange tomato apple
```

Example 2:

```
>>> pattern_matching("abaac", "blue pink blue pink pink yellow pink")
True
```

Explanation:

```
a = pink
b = blue
c = yellow
abaac -> a b a a c -> pink blue pink pink yellow
                        |    |    |    |      |
                 blue pink blue pink pink yellow pink
```

Prefix Count

You are tasked with designing a class that initializes with a string and provides a method to count how many times a given prefix appears at the beginning of the words in the string. The string will consist of lowercase English letters and spaces separating words. You may assume the string does not change after the object is created.

Goal: The count method is $O(1)$ (constant time).

The class structure should be as follows:

```
class PrefixCount(object):
    def __init__(self, text: str):
        pass

    def count(self, prefix: str) -> int:
        pass
```

Example:

```
>>> pc = PrefixCount("to be or not to be, that is the question")
>>> pc.count("to")  # "to", "to"
2
>>> pc.count("be")  # "be", "be"
2
>>> pc.count("th")  # "that", "the"
2
>>> pc.count("qu")  # "question"
1
```

Queue Extremes

You are tasked with creating an integer queue that can always return its smallest and largest elements instantly. Utilize any built-in types in your language of choice for this implementation.

Returning *None* for most methods when the queue is empty is acceptable.

Goal: The ‘maxValue‘ and ‘minValue‘ methods must operate in $O(1)$ (constant time).

Here’s the general class structure:

```python
class ExtremeQueue(object):
    def __init__(self):
        pass

    def enqueue(self, value: int):
        pass

    def dequeue(self) -> int:
        pass

    def peek(self) -> int:
        pass

    def maxValue(self) -> int:
        pass

    def minValue(self) -> int:
        pass
```

Example usage:

```
>>> eq = ExtremeQueue()
>>> eq.enqueue(5)
>>> eq.enqueue(3)
>>> eq.enqueue(10)
>>> eq.maxValue()  # Should return 10
10
>>> eq.minValue()  # Should return 3
3
>>> eq.dequeue()   # Removes 5 from the queue
5
>>> eq.peek()      # Peeks at the next item, which is 3
3
```

Medium Algorithms

Simplify Ranges

Simplify Range 1

You are given an ordered list of unique integers (positive or negative). You are to summarize the integers based on any consecutive integers with a difference of 1 by their bounds. Your function should return a list of tuples, each tuple represents the lower and upper bound of the range (in that order).

Goal: $O(n)$

```python
def simplify_range_1(numbers: List[int]) -> List[Tuple[int, int]]:
    pass
```

Examples:

```
>>> simplify_range_1([-2, -1, 0, 1, 4, 5, 8, 12, 13, 14, 15])
[(-2, 1), (4, 5), (8, 8), (12, 15)]
```

Simplify Range 2

Similar to above, but with the added requirement that ranges must group all consecutive numbers of the same difference. Elements which belong to more than 1 range are to be included in the range with the smallest difference.

The function must return a list of tuple of 3 numbers. The numbers represent the lower bound, upper bound, and difference (in that order).

Goal: $O(n)$

```
def simplify_range_2(numbers: List[int]) -> List[Tuple[int, int, int]]:
    pass
```

Example:

```
>>> simplify_range_2([-2, 0, 2, 4, 5, 6, 9, 12, 15, 20, 25])
[(-2, 2, 2), (4, 6, 1), (9, 15, 3), (20, 25, 5)]
```

Complexify Range

Make a function that will reproduce the a simplified range given by the *Simplify Range 2* output.

Goal: 1 line

```
def complexify_range(simple_range: List[(int, int, int)]) -> List[int]:
    pass
```

Perimeter Sub Squares

Given the 2D array of integers, find the edge length of the largest square that has a perimeter of all 1s.

Goal: $O(n^3)$

```
def perimeter_sub_square(matrix: List[List[int]]) -> int:
    pass
```

Example:

```
>>> perimeter_sub_square([
...     [0, 1, 0, 1, 1, 0, 0],
...     [0, 1, 1, 1, 1, 0, 0],
...     [0, 1, 1, 0, 1, 1, 0],
...     [0, 1, 0, 1, 1, 0, 0],
...     [1, 1, 1, 1, 1, 0, 0]])
4

0 1 0 1 1 0 0          0 1 0 1 1 0 0
0 1 1 1 1 0 0          0 * * * * 0 0
0 1 1 0 1 1 0   -->    0 * 1 0 * 1 0
0 1 0 1 1 0 0          0 * 0 1 * 0 0
1 1 1 1 1 0 0          1 * * * * 0 0
```

Cornered Sub Squares

Given the 2D array of integers, find the edge length of the largest square that has the corners of all 1s.

Goal: $O(n^4)$

```
def corner_sub_square(matrix: List[List[int]]) -> int:
    pass
```

Example:

```
>>> corner_sub_square([
...     [0, 1, 0, 1, 1, 0, 1],
...     [0, 1, 1, 1, 1, 0, 0],
...     [0, 1, 1, 0, 1, 1, 0],
...     [0, 1, 0, 1, 1, 0, 0],
...     [1, 1, 1, 1, 1, 0, 1]])
5
```

```
0 1 0 1 1 0 1          0 * 0 1 1 0 *
0 1 1 1 1 0 0          0 1 1 1 1 0 0
0 1 1 0 1 1 0   -->    0 1 1 0 1 1 0
0 1 0 1 1 0 0          0 1 0 1 1 0 0
1 1 1 1 1 0 1          1 * 1 1 1 0 *
```

English Ordinal Numbers

Given a positive integer, return its ordinal English representation. The value will be less than 1 billion.

The ordinal numbers should be correctly formed according to standard English rules (e.g., ”first” for 1, ”second” for 2, up to ”twentieth” for 20, then ”twenty-first” for 21, and so on).

Goal: 15 lines or less

```
def english_ordinal_number(number: int) -> str:
    pass
```

Examples:

```
>>> english_ordinal_number(1)
"First"
>>> english_ordinal_number(22)
"Twenty Second"
>>> english_ordinal_number(105)
"One Hundred Fifth"
>>> english_ordinal_number(111)
"One Hundred Eleventh"
>>> english_ordinal_number(121)
"One Hundred Twenty First"
```

Symmetrical Binary Tree

A symmetrical binary-tree is a binary tree-graph that has its two children be reflections of each other. See the example below.

```
      |                             |
      R                             R
    /   \                         /   \
  N       N                     N       N
 / \     / \                   / \     / \
N   N   N   N                 N   N   N   N
 \         /                   \       \
  N       N                     N       N
  Symmetric                     Asymmetric
```

For the purposes of this problem, Nodes are represented as:

```python
class Node(object):
    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right
```

You may assume that all given graphs are valid binary trees (no loops).

Detect Cycle in Directed Graph

Given the root of a directed graph, determine if the graph contains at least one cycle. Return *True* if the graph has a cycle, *False* otherwise.

Assume the graph is represented as an adjacency list, where each node has a list of nodes it points to.

Goal: Worst-case $O(V + E)$ and 10 lines or less

```python
def has_cycle(graph: Dict[int, List[int]]) -> bool:
    pass
```

You may create helper functions if needed. The input graph is represented as a dictionary where keys are node identifiers and values are lists of identifiers representing directed edges from the key node to the nodes in the list.

Examples:

```
>>> has_cycle({1: [2], 2: [3], 3: [1]})
True
```

Explanation: There's a cycle between nodes 1, 2, and 3.

```
>>> has_cycle({1: [2], 2: [3], 3: [4]})
False
```

Explanation: There are no cycles in this graph.

Height of Largest Symmetric Sub-Tree

Given a root of a graph, find the height of the largest symmetric sub-tree of the graph. The height includes the root of the sub-tree.

Goal: Worst-case $O(n \log(n))$

```
def largest_symmetric_height(root: Node) -> int:
    pass
```

Hard Algorithms

Pulling Petals

You may have seen in movies where kids take a flower with large petals and start pulling the petals saying either "He/She loves me" or "He/She loves me not", alternating. Every second petal is removed. Once around the flow, this is repeated with the remaining petals, removing the second remaining petal. This is repeated until there is only 1 petal remaining.

You must create an algorithm, when given n number of petals, that can determine the index (petals are index around the flower) of the final petal to be removed.

The very first petal (index 0, on the first round) is not removed.

Goal: Work for $n \in [2^{32}, 2^{63})$

```
def last_petal(n: int) -> int:
    pass
```

Mastermind Player

Mastermind is a code-breaking game where you try to guess a secret pattern. The pattern consists of four colors, order matters. There are 6 possible colors: white, green, red, blue, yellow, purple. The player makes several guesses on what the pattern is.

Upon each guess, the player will receive two numbers

- Number of colors which match and are in the correct position
- Number of colors which match, but not in the correct position

The secret pattern may contain the same color multiple times.

The pattern and guesses will be represented as a string of four characters. The colors are to be represented by their first letter (w for white, g for green, etc.)

Make The Game - Easy

You are to create a class called Mastermind, which takes in the secret pattern as its only parameter. The object should keep track of the number of guesses made (as an attribute *count*), initializing as 0.

There should also be an attribute *length* which is the length of the secret pattern (presumably 4, but maybe not).

You are to write a method which takes sequences of colors which represent a guess. The function is to return a 2-tuple (a tuple of two elements) of integers. These integers are to represent the accuracy of the guess (as defined above).

The first digit of the tuple is the number of exact matches, and the second number is the number of mispositioned matches.

The secret and pattern will always be the same length. You may assume the length will always be 4 if it makes your solution easier.

Each guess should increment a *count* attribute.

Goal: Make it work for any length of secret-guess pair (meaning potentially more than 4)

```
class Mastermind(object):
    def __init__(self, secret: str):
        pass

    def guess(self, guess: str) -> (int, int):
        pass
```

There are a few corner cases, but they should be clear from the following examples. Keep in mind, that the sum of the two numbers return can never exceed the length of the pattern (if it does, you did something wrong).

```
>>> mm = Mastermind("wbgr")
>>> mm.count
0
>>> mm.guess("wwww")
(1, 0)
>>> mm.guess("wgrr")
(2, 1)
>>> mm.guess("wgrb")
(1, 3)
>>> mm.count
3
>>> mm = Mastermind("wwbb")
>>> mm.guess("bbww")
(0, 4)
>>> mm.guess("wwbb")
(4, 0)
>>> mm.guess("rbrr")
(0, 1)
```

Make The Player - Hard

You are to make an algorithm which will play the game and try guess the secret pattern. The goal is to solve it in as few turns as possible.

Upon completion, your function should return the correct pattern and the number of tries it took to guess.

Goal: Be able to play for any length of pattern (possibly more than 4)

```python
def mastermind_player(mastermind: Mastermind) -> (str, int):
    pass
```

For the case of patterns of length four, you may judge your algorithm based on its *average* number of attempts.

- 6 or less - Mastermind
- 12 or less - Basically Human
- 24 or less - Good
- 50 or less - Decent
- 100 or less - Passable
- more than 1296 (6^4, number of combinations) - Completely Broken.