hackerrank_regex

August 28, 2023

[2]:
```
from itertools import product,starmap
from operator import mul
from collections import Counter
import re
import numpy as np
from fractions import Fraction
from functools import reduce
```

[1]:
```
k, m = map(int,['1','1000'])
# k, m => 1, 1000
print(k, m)

# *map(int,['1','1000']) => 1 1000
```

```
1 1000
1 1000
```

[18]:
```
# Enter => 3[space]1000
type(input())
```

```
3 1000
```

[18]:
```
str
```

[19]:
```
# Enter => 2[space]5[space]4

# str.split(sep=None, maxsplit=-1) => if maxsplit is given, at most maxsplit␣
 ↪splits are done (thus, the list will -
# - have at most maxsplit+1 elements).
input().split(" ",1)
```

```
2 5 4
```

[19]:
```
['2', '5 4']
```

[36]:
```
vec1, vec2 = (5, 7, 5), (5, 7, 5)
for i in zip(vec1, vec2):
    print(i)
```

```
    break
```

```
(5, 5)
```

[44]:
```
([5, 4], [7, 8, 9])
```

[44]:
```
([5, 4], [7, 8, 9])
```

[51]:
```
# a = [[5, 4], [7, 8, 9], [5, 7, 8, 9, 10]]
# for _ in product(*a):
#     print(_)

# (5, 7, 5)
# (5, 7, 7)
# (5, 7, 8)
# ...
# (5, 9, 8)
# (5, 9, 9)
# (5, 9, 10)
# ...
# (4, 9, 7)
# (4, 9, 8)
# (4, 9, 9)
# (4, 9, 10)
```

[47]:
```
def dotproduct(vec1, vec2):
    '''
    np.dot(np.array([5, 7, 5]), np.array([5, 7, 5])) => 99
    '''
    # vec1, vec2 => (5, 7, 5), (5, 7, 5)

    # starmap(function, iterable) => make an iterator that computes the␣
 ↪function using arguments obtained from the iterable.
    # starmap(mul, zip(vec1, vec2)) => <itertools.starmap object at␣
 ↪0x7f1b7c26d510>
    # sum(starmap(mul, zip(vec1, vec2))) => 5*5 + 7*7 + 5*5 (99)
    return sum(starmap(mul, zip(vec1, vec2)))

# input() => 3[space]1000
# input().split() => ['3','1000']
K, M = map(int,input().split())
N_list = []

for _ in range(K):
    N_i, N = input().split(" ",1)
    # N_i, N => '2', '5 4'
    N_list.append( list(map(int, N.split())) )
```

```
# N_list => [[5, 4], [7, 8, 9], [5, 7, 8, 9, 10]]
# *N_list => [5, 4] [7, 8, 9] [5, 7, 8, 9, 10]
# [5,4], [7, 8, 9] => ([5, 4], [7, 8, 9])

max_S = 0
# itertools.product(*iterables, repeat=1) => cartesian product of input␣
 ↪iterables.
for x in product(*N_list):
    S = dotproduct(x,x)%M
    if S > max_S:
        max_S = S
    break


'''
You are given a function f(X) = X^2. You are also given K lists. The ith list␣
 ↪consists of N elements.
You have to pick one element from each list so that the value from the equation␣
 ↪below is maximized:
S = (X1^2 + X2^2 + ... + XK^2)%M
Xi denotes the element picked from the ith list.
Note that you need to take exactly one element from each list.


# 3 1000 => 3 lists, 1000 is M
# 2 5 4  => 2 element in the first list and elements are [5,4]
# 3 7 8 9 => => 3 element in the second list and elements are [5,4]
# 5 5 7 8 9 10
'''
print(max_S)
```

```
3 1000
2 5 4
3 7 8 9
5 5 7 8 9 10
99
```

[ ]:

[53]:
```
################################################################################

x = int(input())
y = Counter(map(int, input().split()))
# y => Counter({5: 2, 6: 2, 2: 1, 3: 1, 4: 1, 8: 1, 7: 1, 18: 1})
z = int(input())
```

```
total = 0
for i in range(z):
    size, rate = map(int, input().split())
    if y[size]:
        y[size] -= 1
        total += rate
'''
Raghu is a shoe shop owner. His shop has X number of shoes (problem from␣
 ↪hackerrank).
10  =>  first line contains x, the number of shoes.
2 3 4 5 6 8 7 6 5 18 => second line contains the space separated list of all␣
 ↪the shoe sizes in the shop.
6 => third line contains z, the number of customers.
6 55 => shoe size and price
6 45
6 55
4 40
18 60
10 50

Your task is to compute how much money earned.
'''
print(total)
```

```
10
2 3 4 5 6 8 7 6 5 18
6
6 55
6 45
6 55
4 40
18 6
10 50
146
```

[8]:
```
#########################################################
# ^: Asserts the start of the string.
# [1-9]: Matches a single digit between 1 and 9.
# [0-9]{5}: Matches exactly five digits (0-9).
# $: Asserts the end of the string.
regex_integer_in_range = r"^[1-9][0-9]{5}$" # [0-9] => 0,1,2,3 ..., 9

# (\d) => This part captures a single digit (0-9) and stores it in a capturing␣
 ↪group. The parentheses ( ) denote a -
#     - capturing group, and \d represents any digit.
# (?=...) => This is a positive lookahead assertion. It specifies a condition␣
 ↪that must be satisfied for a match to -
```

```
#     - occur, but it doesn't consume characters in the string.
# ...(?=\d) => It checks if there is a digit followed the content of the␣
 ↪capturing group.
#  ...(?=\d\1) => It checks if there is a "digit" and then a "same number (as␣
 ↪the content of capturing group)", -
#     - followed by the content of the capturing group.

# Isaac (?=Asimov) => will match 'Isaac ' only if it's followed by 'Asimov'.

# (\d)(?=\d\1) captures a digit and then uses a positive lookahead according to␣
 ↪specified "condition" (...\d\1...).
# re.findall(regex_alternating_repetitive_digit_pair, '10101') => ['1', '0',␣
 ↪'1']
regex_alternating_repetitive_digit_pair = r"(\d)(?=\d\1)"

P = input().strip()

'''
A valid postal code P have to fullfil both below requirements:
    P must be a number in the range from 100000 to 999999 inclusive.
    P must not contain more than one alternating repetitive digit pair.

E.g.,
121426 # Here, 1 is an alternating repetitive digit.
523563 # Here, NO digit is an alternating repetitive digit.
552523 # Here, both 2 and 5 are alternating repetitive digits.
'''

# re.match(regex_integer_in_range, P) => <re.Match object; span=(0, 6),␣
 ↪match='110000'>
# bool(re.match(regex_integer_in_range, P)) => True
# re.findall(regex_alternating_repetitive_digit_pair, '110000') => ['0', '0']
print ( bool(re.match(regex_integer_in_range, P))
and len(re.findall(regex_alternating_repetitive_digit_pair, P)) < 2 )
# False for input 110000.
```

```
110000
False
```

[88]:
```
re.findall(regex_alternating_repetitive_digit_pair, '110001')
```

[88]:
```
['0']
```

[86]:
```
################################################################
import math
import os
import random
```

```python
import re
import sys

# str.rstrip([chars]) => return a copy of the string with trailing characters␣
 ↪removed. If omitted or None, the -
#    - chars argument defaults to removing whitespace.
# '7 3'.rstrip().split() => ['7', '3']
# list(map(int, '7 3'.rstrip().split())) => [7, 3]
n, m = map(int, input().rstrip().split())

matrix = []

for _ in range(n):
    matrix_item = input()
    matrix.append(matrix_item)

# matrix => ['Tsi', 'h%x', 'i #', 'sM ', '$a ', '#t%', 'ir!']
# *matrix => Tsi h%x i # sM  $a  #t% ir!

matrix = list(zip(*matrix))
# matrix => [('T', 'h', 'i', 's', '$', '#', 'i'), ('s', '%', ' ', 'M', 'a',␣
 ↪'t', 'r'), ('i', 'x', '#', ' ', ' ', '%', '!')]

sample = ''

for words in matrix:
    for char in words:
        sample += char
'''
reads the column from top to bottom and starts reading from the leftmost column.
 ↪ if there are symbols or spaces
between two alphanumeric characters of the decoded script, then replace them␣
 ↪with a single space ''. there is
no need to use 'if' conditions for decoding. alphanumeric characters consist of:
 ↪ [A-Z, a-z, and 0-9].

7 3 => the first line contains space-separated integers N (rows) and M␣
 ↪(columns) respectively.
Tsi => 1st row element of matrix.
h%x
i #
sM
$a
#t%
ir!
'''
```

```
# sample => This$#is% Matrix#  %!

# (?<=\w) => positive lookbehind assertion.
# (?=\w) => positive lookahead assertion.
# \w => word character.
# ([^\w\d]+) => capturing group that matches one or more consecutive characters␣
 ↪that are not word characters or digits.
print(re.sub(r'(?<=\w)([^\w\d]+)(?=\w)', ' ', sample))
```

```
7 3
Tsi
h%x
i #
sM
$a
#t%
ir!
This is Matrix#  %!
```

[ ]:

[ ]:

[90]:
```
###############################################################################################
# initiailzing map function
N, M = map(int, input().split())

# taking for rows
rows = [input() for _ in range(N)]
# rows => ['10 2 5', '7 1 0', '9 9 9', '1 23 12', '6 5 9']

# taking input from user
K = int(input())

'''
sort the data based on the kth attribute and print the final resulting table.
if two values for different rows are the same for a attribute, print the row␣
 ↪that appeared first in the input.

5 3 => first line contains rows and columns separated by a space.
10 2 5 => elements of row.
7 1 0
9 9 9
1 23 12
6 5 9
1 => sort over this attribute/column.
```

```
'''

# sorted(iterable, /, *, key=None, reverse=False) =>
# key => key specifies a function of one argument that is used to extract a␣
 ↪comparison key from each element -
#    - in iterable.
# sorted(rows, key=lambda row: int(row.split()[K])) => ['7 1 0', '10 2 5', '6 5␣
 ↪9', '9 9 9', '1 23 12']
for row in sorted(rows, key=lambda row: int(row.split()[K])):
    print(row)
```

```
5 3
10 2 5
7 1 0
9 9 9
1 23 12
6 5 9
1
7 1 0
10 2 5
6 5 9
9 9 9
1 23 12
```

[116]:
```
##################################################################################
# taking the input from user
s = input()

# sorted sorts in ascending i.e., False -> True.
s = sorted(s, key = lambda x: (x.isdigit() and int(x)%2==0, x.isdigit(), x.
 ↪isupper(), x.islower(), x) )
# x = 'S', key => (False, False, True, False, 'S')
# x = 'g', key => (False, False, False, True, 'g')
# x = '1', key => (False, True, False, False, '1')
# x = '4', key => (True, True, False, False, '4')
'''
task is to sort the string  in the following manner:
    All sorted lowercase letters are ahead of uppercase letters.
    All sorted uppercase letters are ahead of digits.
    All sorted odd digits are ahead of sorted even digits.

Sorting1234 => single line of input contains the string s.
'''
# s => ['g', 'i', 'n', 'o', 'r', 't', 'S', '1', '3', '2', '4']
# printing the sorted string
print(*s,sep = '')
```

```
Sorting1234
ginortS1324
```

[120]:
```
##########################################################################################3

def merge_the_tools(string, k):
    l = len(string)//k
    for i in range(l):
        # i = 0, dict.fromkeys(string[i*k:(i*k)+k]) => {'A': None}
        # i = 1, dict.fromkeys(string[i*k:(i*k)+k]) => {'B': None, 'C': None,␣
 ↪'A': None}

        # dict.fromkeys(iterable[, value]) => create a new dictionary with keys␣
 ↪from iterable and values set to value.
        print(''.join(dict.fromkeys(string[i*k:(i*k)+k])))

string, k = input(), int(input())

'''
split string s into n/k substrings i.e., n = length of string. k = factor of n.
then print each substring without any repeat occurrence of a character.

AAABCADDE =>  first line contains a single string.
3 => k, factor of n.
'''
merge_the_tools(string, k)
```

```
AAABCADDE
3
A
BCA
DE
```

[119]:
```
z = {'B': None, 'C': None, 'A': None}
''.join(z)
```

[119]:
```
'BCA'
```

[12]:
```
########################################################################
def minion_game(string: str) -> None:
    """Print the winner of the game and the score."""
    kevin = stuart = 0
    length: int = len(string)

    for i, char in enumerate(string):
        points: int = length - i
        if char in {"A", "E", "I", "O", "U"}:
```

```
            kevin += points
        else:
            stuart += points

    if kevin == stuart:
        print("Draw")
    else:
        print(*("Stuart", stuart) if stuart > kevin else ("Kevin", kevin))

'''
Two players (kevin and stuart) are given the same string, string  will contain␣
 ↪only uppercase letters.
Both have to make substrings using the letters of the string.
kevin has to make words starting with vowels.
stuart has to make words starting with consonants.
The game ends when both players have made all possible substrings.
A player gets +1 point for each occurrence of the substring in the string.

BANANA => a single line of input containing the string.
OOOH
COOL
'''
s = input().strip()
minion_game(s)
```

```
BANANA
Stuart 12
```

[3]:
```
################################################################################

integer1 = int(input())

# str.strip([chars]) => return a copy of the string with the leading and␣
 ↪trailing characters removed.
words = [input().strip() for _ in range(integer1)]

# words => ['bcdef', 'abcdefg', 'bcde', 'bcdef']

# Counter(words).keys() => dict_keys(['bcdef', 'abcdefg', 'bcde'])


val = list(Counter(words).values())
listToStr = ' '.join([f"{elem}" for elem in val])

'''
You are given n words. Some words may repeat. For each word, output its number␣
 ↪of occurrences in the all words.
```

```
The output order should correspond with the input order of appearance of the␣
 ↪word.


4 => The first line contains the integer, n.
bcdef => The next n lines each contain a word.
abcdefg
bcde
bcdef
'''

# output the number of occurrences for each distinct word according to their␣
 ↪appearance in the input.
print(listToStr)
```

```
4
bcdef
abcdefg
bcde
bcdef
3
2 1 1
```

[22]:
```
#########################################################################3

# taking input from user and sorting it
s = input().strip()
# list(s) => ['g', 'o', 'o', 'g', 'l', 'e']
# Counter('google') => Counter({'g': 2, 'o': 2, 'l': 1, 'e': 1})
# Counter(['google', 'apple']) => Counter({'google': 1, 'apple': 1})
# Counter(list(s)) => Counter({'g': 2, 'o': 2, 'l': 1, 'e': 1})

s = sorted(s)
# using counter method to find the frequency of each of the words
freq = Counter(s)

'''
Given a string s in lowercase letters, your task is to find the top three most␣
 ↪common characters in the string.

google => a single line of input containing the string.
'''

# freq.most_common() => [('g', 2), ('o', 2), ('e', 1), ('l', 1)]

# using for loop to print the three words with frequency
for k, v in freq.most_common(3): # 3
```

```
    print(k, v)
```

```
google
g 2
o 2
e 1
```

[18]:
```
'google'.strip()
```

[18]:
```
'google'
```

[19]:
```
list('google')
```

[19]:
```
['g', 'o', 'o', 'g', 'l', 'e']
```

[23]:
```
Counter('google')
```

[23]:
```
Counter({'g': 2, 'o': 2, 'l': 1, 'e': 1})
```

[24]:
```
Counter('google').most_common()
```

[24]:
```
[('g', 2), ('o', 2), ('l', 1), ('e', 1)]
```

[ ]:
```

```

[31]:
```
##############################################################################


# (?!...) =>  negative lookahead assertion. e.g., Isaac (?!Asimov) will match␣
 ↪'Isaac ' only if it's not followed -
#            - by 'Asimov'.
# .* => matches any number of elements/characters (except newline element/
 ↪character).
# (\d)(-?\1) => matches a digit followed by an optional hypen sign and then the␣
 ↪same digit that was captured earlier.
# ..(\d)(-?\1){3}.. => this part is checking for the repeated sequence of the␣
 ↪captured digit (\d).
#                      - captured digit (\d) can be repeated three times. i.e.,␣
 ↪..2-22-2.. (invalid),
#                      - ..2-2-2-2.. (invalid), ..2-2-2.. (valid), ..222..␣
 ↪(valid), ..2222.. (invalid), ..2#222.. (valid).
# (?!.*(\d)(-?\1){3}) => checking that there are no repeated sequences of a␣
 ↪digit that are 4 or more characters long in the entire string.

# [456]\d{3} => checking that string must start with a 4, 5 or 6, then a digit␣
 ↪(this digit can appear thrice).
```

```
# (?: ... ) =>  represents a non-capturing group. non-capturing groups are␣
 ↪particularly useful when you want to -
#               - use parentheses ( () ) for grouping but don't need to store␣
 ↪the matched substring for later use.
# (?:-?\d{4}){3} => checking optional hyphen, then exactly four digit,␣
 ↪preceding element (in this case,  -
#                  - the non-capturing group) should appear exactly three times.


# Compile a regular expression pattern into a "regular expression object",␣
 ↪which can be used for matching using -
#    - its match(), search() and other methods.
pattern = re.compile(
    r'^'
    r'(?!.*(\d)(-?\1){3})'
    r'[456]\d{3}'
    r'(?:-?\d{4}){3}'
    r'$')

'''
A valid credit card from ABCD Bank has the following characteristics:
 It must start with a 4, 5 or 6.
 It must contain exactly 16 digits.
 It must only consist of digits (0-9).
 It may have digits in groups of 4, separated by one hyphen "-".
 It must NOT use any other separator like ' ' , '_', etc.
 It must NOT have 4 or more consecutive repeated digits.


5 => first line of input contains an integer N.
4123456789123456 => next N lines contain credit card numbers.
5123-4567-8912-3456
61234-567-8912-3456
4123356789123456
5133-3367-8912-3456
'''

# using for loop to the input from user
for _ in range(int(input().strip())):

    # pattern.search(input().strip()) =>
    #        <re.Match object; span=(0, 19), match='5123-4567-8912-3456'>   ␣
 ↪(for 5123-4567-8912-3456)
    #        None                                                            ␣
 ↪(for 61234-567-8912-3456)
    print('Valid' if pattern.search(input().strip()) else 'Invalid')
```

```
1
5123-4567-8912-3456
Valid
```

[6]:
```
###############################################################################

def fun(email):
    # r ("raw string literal" used to indicate that the string should be␣
 ↪treated as a raw string, which means that -
    #    - backslashes within the string are treated as literal characters and␣
 ↪not as escape characters).
    #
    # [\\w-]+ => matches one or more word characters (\w, which includes␣
 ↪letters, digits, and underscores) or hyphens.
    # \\. => matches a literal dot character (escaped with backslashes).
    # [a-z]{1,3} => matches one to three lowercase letters.

    #pattern = re.compile("^[\\w-]+@[0-9a-zA-Z]+\\.[a-z]{1,3}$")

    # \w+ => matches one or more word characters at the beginning of the email␣
 ↪address.
    # ([-_]?\\w+)* => allows for hyphens or underscores followed by more word␣
 ↪characters, and this pattern can repeat.
    pattern = re.compile("^\\w+([-_]?\\w+)*@[0-9a-zA-Z]+\\.[a-z]{1,3}$")


    # pattern.match(email) => <re.Match object; span=(0, 19),␣
 ↪match='lara@hackerrank.com'>
    return pattern.match(email)


def filter_mail(emails):
    # filter(function, iterable) => construct an iterator from those elements␣
 ↪of iterable for which function is true.
    # filter(fun, emails) => <filter object at 0x7f89b42a4790>.
    return list(filter(fun, emails))


n = int(input())
emails = []
for _ in range(n):
    emails.append(input().strip())

'''
Valid email addresses must follow these rules:
    It must have the username@websitename.extension format type.
```

```
    The username can only contain letters, digits, dashes and underscores (i.e.
 ↪, [a-z],[A-Z],[0-9],[_-]).
    The website name can only have letters and digits (i.e., [a-z],[A-Z],[0-9]).
    The extension can only contain letters (i.e., [a-z],[A-Z]).
    The maximum length of the extension is 3.

3 => the first line of input is the integer N.
lara@hackerrank.com => N lines next, each containing a string.
brian-23@hackerrank.com
harsh@gmail
'''

filtered_emails = filter_mail(emails)
filtered_emails.sort()
print(filtered_emails)
```

```
1
brian-23@hackerrank.com
['brian-23@hackerrank.com']
```

[ ]:

[54]:
```
##########################################################################################

class EvenStream(object):
    def __init__(self):
        self.current = 0

    def get_next(self):
        to_return = self.current
        self.current += 2
        return to_return

class OddStream(object):
    def __init__(self):
        self.current = 1

    def get_next(self):
        to_return = self.current
        self.current += 2
        return to_return

def print_from_stream(n, stream=None):
    if stream is None:
        stream = EvenStream()
    for _ in range(n):
        print(stream.get_next())
```

```
queries = int(input())

'''
def print_from_stream(n, stream) => function should print the first n values␣
 ↪returned by "get_next() method" of -
            - "stream object" provided as an argument. Each of these values␣
 ↪should be printed in a separate line.
            whenever the function is called without the stream argument, it␣
 ↪should use an "instance of EvenStream"
            class as the value of stream.

3 => in the first line, there is a single integer q denoting the number of␣
 ↪queries.
odd 2 => next q lines contains a stream_name followed by integer n.
even 3
odd 5
'''
for _ in range(queries):
    stream_name, n = input().split()
    n = int(n)
    if stream_name == "even":
        print_from_stream(n)
    else:
        print_from_stream(n, OddStream())
```

```
3
odd 2
1
3
even 3
0
2
4
odd 5
1
3
5
7
9
```

[7]:
```
################################3
'''
modify && and || symbols to the following:
&& → and
|| → or
```

```
Note do not change &&& or ||| or & or |.
Only change those '&&' which have space on both sides.
Only change those '||' which have space on both sides.

4
if a + b > 0 && a - b < 0 || a*b=2:
    start()
elif a*b > 10 || a/b < 1 && a*b=2:
    stop()
'''

def lambda_extended(x):
    # when input()='if a + b > 0 && a - b < 0 || a*b=2:' this function is called twice.
    # x (on first call of this func)=>
    # <re.Match object; span=(13, 15), match='&&'>
    # x (on second call of this func)=>
    # <re.Match object; span=(26, 28), match='||'>


    # x.group(), x.group(0), x.group(1) => &&, &&, &&      (on first call of this func)
    # x.group(2) => no such group            (on first call of this func)
    if x.group() == '&&':
        return 'and'
    else: return 'or'

for i in range(int(input())):

    # re.sub(pattern, repl, string, ...) => return the string obtained by replacing the leftmost non-overlapping -
    #    - occurrences of pattern in string by the replacement repl. if repl is a function, it is called for every -
    #    - non-overlapping occurrence of pattern. the function takes a single match object argument, and returns -
    #    - the replacement string.
    # (?<=...) => positive lookbehind assertion.
    # (&&|\|\|) => matches either && (logical AND) or || (logical OR) operator.
    # (?= ) => positive lookahead assertion.

    #print(re.sub(r'(?<= )(&&|\|\|)(?= )', lambda x: 'and' if x.group() == '&&' else 'or', input()))
    print(re.sub(r'(?<= )(&&|\|\|)(?= )', lambda_extended, input()))
```

```
1
if a + b > 0 && a - b < 0 || a*b=2:
if a + b > 0 and a - b < 0 or a*b=2:
```

[77]:
```
###############################################################
def product(fracs):
    # reduce(function, iterable) => apply function of two arguments␣
 ↪cumulatively to the items of iterable, from -
    #     - left to right, so as to reduce the iterable to a single value

    # reduce(lambda x, y: x * y, fracs) => 5/8
    t = reduce(lambda x, y: x * y, fracs)

    return t.numerator, t.denominator

'''
Given a list of rational numbers, find their product.

3 => first line contains n, the number of rational numbers.
1 2 => next n lines contain two integers, the numerator and denominator of␣
 ↪rational number.
3 4
10 6

5 8 => numerator and denominator of output.
'''

fracs = []
for _ in range(int(input())):
    # *map(int, input().split()) => 1 2        (for first input 1 2)
    # Fraction(*map(int, input().split())) => 1/2     (for first input 1 2)

    fracs.append( Fraction(*map(int, input().split())) )

# fracs => [Fraction(1, 2), Fraction(3, 4), Fraction(5, 3)]
# fracs[0] => 1/2

result = product(fracs)
print(*result)
```

```
3
1 2
3 4
10 6
5 8
```

[ ]: