TPA - Trabalho 3

Edson Boldrini

IFES Campus Serra - 2019/2

1 Introdução

Antes de começar a falar sobre o relatório gostaria de informar que minha dupla nesse trabalho era o aluno Caio Kinupp, porém ele desistiu da matéria e não contribuiu em nada com esse trabalho, por isso apenas o meu nome (Edson Boldrini) consta como autor desse documento e autor das soluções dos problemas.

Este documento tem como objetivo apresentar um relatório sobre as soluções de problemas escolhidos dentro do UVA Online Judge. Estas soluções compreendem o uso e aprendizado de várias estruturas de dados abordadas durante o segundo semestre (2019/2) da disciplina de Técnicas de Programação Avançada ministrada pelo professor Jefferson Oliveira Andrade no IFES Campus Serra.

2 UVA Submissions

My Submissions

#		Problem	Verdict	Language	Run Time	Submission Date
24227792	1062	Containers	Accepted	PYTH3	0.010	2019-11-21 15:12:17
24226320	514	Rails	Accepted	PYTH3	0.420	2019-11-21 09:19:55
24217892	11988	Broken Keyboard (a.k.a. Beiju Text)	Time limit exceeded	PYTH3	1.000	2019-11-19 15:58:30
24212553	11926	Multitasking	Time limit exceeded	PYTH3	1.000	2019-11-18 14:22:25
24208980	10264	The Most Potent Corner	Accepted	PYTH3	0.430	2019-11-17 23:47:34
24194642	10258	Contest Scoreboard	Accepted	PYTH3	0.010	2019-11-14 13:34:04
24194473	10258	Contest Scoreboard	Presentation error	PYTH3	0.010	2019-11-14 13:05:04
24189888	10107	What is the Median?	Accepted	PYTH3	0.160	2019-11-13 12:19:29
24189542	11581	Grid Successors	Accepted	PYTH3	0.080	2019-11-13 11:27:11
24187021	10920	Spiral Tap	Time limit exceeded	PYTH3	3.000	2019-11-13 00:03:49
24153509	11340	Newspaper	Time limit exceeded	PYTH3	1.000	2019-11-05 16:03:52
24153466	11340	Newspaper	Time limit exceeded	PYTH3	1.000	2019-11-05 15:51:11
24152659	10038	Jolly Jumpers	Accepted	PYTH3	0.170	2019-11-05 12:55:19
24152317	10038	Jolly Jumpers	Wrong answer	PYTH3	0.140	2019-11-05 11:45:05
24152267	10038	Jolly Jumpers	Wrong answer	PYTH3	0.130	2019-11-05 11:35:24
24150652	10038	Jolly Jumpers	Wrong answer	PYTH3	0.120	2019-11-05 03:33:05
24150641	10038	Jolly Jumpers	Runtime error	PYTH3	0.000	2019-11-05 03:27:33
24113383	12247	Jollo	Accepted	PYTH3	0.010	2019-10-27 19:23:40
24113352	12247	Jollo	Runtime error	PYTH3	0.000	2019-10-27 19:15:04

<< Start < Prev 1 Next > End >>

Display # 50 Results 1 - 19 of 19

Figure 1: Submissions

Para os problemas 11340, 10920, 11926, 11988 o tempo limite de execução foi excedido, porém arquivos de testes com dados de testes do debug do UVa foram adicionados aos arquivos do trabalho comprovando a eficácia dos algorítmos.

3 Estruturas de dados

3.1 Warm up

3.1.1 UVa 12247 – Jollo

Este problema tem o objetivo de determinar qual a última carta que o príncipe necessita receber para ganhar da princesa não importando quão mal ele jogue. Para chegar a essa solução, desenvolvi um algoritmo que sempre faz com que o príncipe execute a pior jogada possível, ou seja, jogue sempre a maior carta dele menor que a carta que a princesa jogue. Com isso, consigo saber se é possível ele chegar a última rodada tendo ganhado ao menos um dos games (já que estamos numa melhor de 3), caso ele não ganhe nenhum a resposta é -1 pois já terá perdido o jogo, para o caso dele ter ganho ao menos um dos games, a carta que ele necessita receber para garantir a vitória é a carta com o valor da última que sobrou da princesa + 1, verificando se a carta já não saiu antes, caso já tenha saído, incremente até encontrar uma carta que não tenha saído e seja menor que 52.

```python
# Problem: UVA 12247 - Jollo
# Author(s): Edson Boldrini

import sys


def solve(princessCards, princeCards):
    princessCards.sort(reverse=True)
    princeCards.sort(reverse=True)
    if (princessCards[0] > princeCards[0] and princessCards[1] > princeCards[1]):
        n = -1
    elif(princessCards[0] < princeCards[0] and princessCards[0] < princeCards[1]):
        n = 1
        while n in princessCards or n in princeCards:
            n += 1
    else:
        n = princessCards[1]+1
        while n in princessCards or n in princeCards:
            n += 1
        if (princessCards[0] > n and princessCards[1] > princeCards[1]):
            n = princessCards[0]+1
            while n in princessCards or n in princeCards:
                n += 1

    if (n > 52):
        n = -1

    print(n)
```

```
    # print("---")


def main():
    princess = []
    prince = []
    try:
        line = input()
        while line:
            cards = line.replace('\n', '').split(' ')
            if(cards == ['0', '0', '0', '0', '0']):
                break
            for i in range(len(cards)):
                cards[i] = int(cards[i])
            princess = cards[:3]
            prince = cards[3:]
            solve(princess, prince)
            try:
                line = input()
            except EOFError:
                break
    except EOFError:
        print("No lines")


if __name__ == "__main__":
    main()
```

3.2 Vetores

3.2.1 UVa 10038 – Jolly Jumpers

Este problema tem objetivo de encontrar sequencias do tipo jolly jumpers. Para encontrar essas sequencias meu raciocínio foi primeiramente encontrar as diferenças permitidas (allowedDifferences) para aquela sequencia de entrada e depois comparei se as diferenças encontradas eram permitidas, caso fossem eu printava "Jolly" e se não fossem printava "Not jolly" assim como a especificação determina.

```
# Problem: UVA 10038 - Jolly Jumpers
# Author(s): Edson Boldrini

import math


```

```
def solve(numbers):
    differences = []
    if (numbers[0] == 1 or len(numbers) == 2):
        print("Jolly")
    else:
        for i in range(1, len(numbers)-1):
            candidate = abs(numbers[i]-numbers[i+1])
            differences.append(candidate)
        allowedDifferences = [i for i in range(1, numbers[0])]
        jolly = True
        if(0 not in differences):
            for j in allowedDifferences:
                if (j not in differences):
                    jolly = False
                    break
            if (jolly):
                print("Jolly")
            else:
                print("Not jolly")
        else:
            print("Not jolly")


def main():
    try:
        line = input()
        while line:
            numbers = line.split(' ')
            for i in range(len(numbers)):
                numbers[i] = int(numbers[i])
            solve(numbers)
            try:
                line = input()
            except EOFError:
                break
    except EOFError:
        print("No lines")


if __name__ == "__main__":
    main()
```

3.2.2 UVa 11340 – Newspaper

Este problema tem o objetivo de encontrar o preço das letras a ser cobrado pela publicação no jornal. Para essa solução, li os dados de entrada sobre os custos de cada caractere e os guardei num dicionário (chave, valor) e após isso caminhei por todo o texto analisando caractere a caractere buscando seu preço definido pelo dicionário anteriormente descrito

```
# Problem: UVA 11340 - Newspaper
# Author(s): Edson Boldrini


def solve(charactersCost, charactersList):
    price = 0.0
    for c in charactersList:
        if(c in charactersCost.keys()):
            price += charactersCost[c]
    finalCost = str(round(price, 2)).split('.')
    if(len(finalCost[1]) == 1):
        completeZero = finalCost[1].ljust(2, '0')
        finalCost.pop(1)
        finalCost.append('.')
        finalCost.append(completeZero)
        finalCost = ''.join(finalCost)
        print(finalCost+'$')
    else:
        print(str(round(price, 2))+'$')


def main():
    try:
        line = input()
        numberOfTests = int(line)
        # print(numberOfTests)
        for i in range(numberOfTests):
            try:
                line = input()
                numberOfPaidCharacters = int(line)
                # print(numberOfPaidCharacters)
                charactersCost = {}
                for j in range(numberOfPaidCharacters):
                    try:
                        line = input()
                        data = line.split(' ')
                        charactersCost[data[0]] = float(data[1])/100
                    except EOFError:
                        break
                # print(charactersCost)
```

```
                try:
                    line = input()
                    numberOfTextLines = int(line)
                    charactersList = []
                    for k in range(numberOfTextLines):
                        try:
                            line = input()
                            data = [char for char in line]
                            for c in data:
                                charactersList.append(c)
                        except EOFError:
                            break
                    # print(charactersList)
                    solve(charactersCost, charactersList)
                except EOFError:
                    break
            except EOFError:
                break
    except EOFError:
        print("No lines")


if __name__ == "__main__":
    main()
```

3.3 Matrizes

3.3.1 UVa 10920 – Spiral Tap

Este problema tem o objetivo de saber a posição exata de um determinado número dentro de um tabuleiro espiral. Ao receber o tamanho do tabuleiro e o número para ser encontrado, é determinado em qual quadrante ele está ou seja, em qual dos circulos internos ao tabuleiro ele se encontra. Após determinar o quadrante em que ele se encontra o número é procurado pela linha e coluna utilizando duas variáveis auxiliáres (lb = lowerbound e pad = padding) para encontrar o limite inferior de busca e a quantidade de quadrantes fora dali, com essas duas variáveis é possível determinar a linha e a coluna do número encontrado utilizando elas para caminhar entre as posições possiveis.

```
# Problem: UVA 10920 - Spiral Tap
# Author(s): Edson Boldrini

import math


```

```
def solve(numbers):
    borderSize = int(numbers[0])
    numberToFind = int(numbers[1])

    center = (borderSize//2)+1

    if(numberToFind == 1):
        print("Line = %d, column = %d." % (center, center))
    else:
        i = 3
        while(numberToFind > i*i):
            i += 2
        lb = (i-2)*(i-2)
        pad = (borderSize-i)/2

        column = center
        line = center

        if(numberToFind <= lb + i-1):
            column = lb+i-numberToFind+pad
            line = borderSize-pad
        elif(lb + i-1 < numberToFind and numberToFind <= lb + 2*(i-1)):
            column = pad+1
            line = lb+(i-1)*2+1-numberToFind+pad
        elif(lb + 2*(i-1) < numberToFind and numberToFind <= lb + 3*(i-1)):
            column = numberToFind-(lb+2*(i-1))+pad+1
            line = pad+1
        else:
            column = borderSize-pad
            line = numberToFind-(lb+3*(i-1))+pad+1

        print("Line = %d, column = %d." % (line, column))


def main():
    try:
        line = input()
        while line:
            numbers = line.split(' ')
            if(numbers[0] == '0' and numbers[1] == '0'):
                break
            solve(numbers)
            try:
                line = input()
            except EOFError:
                break
    except EOFError:
```

```
54          print("No lines")
55
56
57  if __name__ == "__main__":
58      main()
```

3.3.2 UVa 11581 – Grid successors

Este problema tem o objetivo de informar qual o índice que permite que a função aplicada a matriz retorna uma matriz com 0 em todas as posições. Para achar a solução deste problema é definido dois arrays auxiliares para comparações, dx e dy. Com esses arrays auxiliares é possível convertendo a matriz em cada execução, quando ela chega a zero, o programa termina.

```
1   # Problem: UVA 11581 - Grid Successors
2   # Author(s): Edson Boldrini
3
4   dx = [1, -1, 0, 0]
5   dy = [0, 0, -1, 1]
6
7
8   def check(x, y):
9       return x >= 0 and x < 3 and y >= 0 and y < 3
10
11
12  def solve(firstLine, secondLine, thirdLine):
13      response = -1
14      if(not (firstLine == "000" and secondLine == "000" and thirdLine == "000")):
15          grid = []
16          f = []
17          s = []
18          t = []
19          f.append(int(firstLine[0]))
20          f.append(int(firstLine[1]))
21          f.append(int(firstLine[2]))
22          s.append(int(secondLine[0]))
23          s.append(int(secondLine[1]))
24          s.append(int(secondLine[2]))
25          t.append(int(thirdLine[0]))
26          t.append(int(thirdLine[1]))
27          t.append(int(thirdLine[2]))
28          grid.append(f)
29          grid.append(s)
30          grid.append(t)
31          newGrid = [[None, None, None], [None, None, None], [None, None, None]]
```

```
        while (True):
            allZeros = True
            for i in range(len(grid)):
                for j in range(len(grid)):
                    if (grid[i][j] != 0):
                        allZeros = False

            if (allZeros):
                break

            count = 0
            for i in range(len(grid)):
                for j in range(len(grid)):
                    count = 0
                    for m in range(len(dy)):
                        if (check(i + dx[m], j + dy[m])):
                            count += grid[i + dx[m]][j + dy[m]]
                    newGrid[i][j] = count % 2

            response += 1
            for i in range(len(grid)):
                for j in range(len(grid)):
                    grid[i][j] = newGrid[i][j]

    print(response)


def main():
    try:
        line = input()
        numberOfTests = int(line)
        for i in range(numberOfTests):
            try:
                blankLine = input()
                firstLine = input()
                secondLine = input()
                thirdLine = input()
                solve(firstLine, secondLine, thirdLine)
            except EOFError:
                break
    except EOFError:
        print("No lines")


if __name__ == "__main__":
    main()
```

3.4 Ordenação

3.4.1 UVa 10107 – What is the Median?

Este problema tem objetivo de encontrar a mediana de uma cadeia de números a medida que ela cresce. Para resolver esse problema o array foi ordenado a cada inserção e depois o tamanho dele foi usado para descobrir a posição da mediana. Caso o tamanho do array seja ímpar, a posição da mediana é exatamente o meio do array, caso seja par, a mediana é a média entre as duas posições mais ao centro do array.

```
1  # Problem: UVA 10107 - What is the Median?
2  # Author(s): Edson Boldrini
3
4
5  def solve(acc):
6      acc.sort()
7      median = 0
8      if(len(acc) % 2 == 1):
9          median = int(acc[(len(acc)//2)])
10     else:
11         median = (int(acc[(len(acc)//2)-1]) + int(acc[len(acc)//2]))//2
12     print(median)
13
14
15 def main():
16     try:
17         line = input()
18         acc = []
19         while line:
20             acc.append(int(line))
21             solve(acc)
22             try:
23                 line = input()
24             except EOFError:
25                 break
26     except EOFError:
27         print("No lines")
28
29
30 if __name__ == "__main__":
31     main()
```

3.4.2 UVa 10258 – Contest Scoreboard

Este problema tem objetivo de encontrar o score final de um torneio de programação. Para encontrar essa solução foram utilizados dois dicionários auxiliares, um de submissões corretas e outro submissões incorretas, para assim calcular as pontuações corretas de cada competidor. Cada linha (submissão) é processada unicamente e calcula a pontuação ou penalidade referente a aquela submissão.

```
# Problem: UVA 10258 - Contest Scoreboard
# Author(s): Edson Boldrini

from operator import itemgetter


def solve(submissions):
    correctSubmissions = {}
    incorrectSubmissions = {}
    contestants = {}
    for s in submissions:
        result = s[3]
        contestant = s[0]
        if (result in ["C", "I"]):
            submissionData = (contestant, s[1])
            if (result == "C"):
                if(submissionData not in correctSubmissions):
                    correctSubmissions[submissionData] = s[2]
                    if(submissionData in incorrectSubmissions):
                        if(contestant not in contestants):
                            contestants[contestant] = [
                                int(contestant), 1, int(s[2])+20]
                        else:
                            contestants[contestant] = [int(contestant), int(
                                contestants[contestant][1])+1, int(contestants[contestant][2])+int(s[2])+20]
                    else:
                        if(contestant not in contestants):
                            contestants[contestant] = [
                                int(contestant), 1, int(s[2])]
                        else:
                            contestants[contestant] = [int(contestant), int(
                                contestants[contestant][1])+1, int(contestants[contestant][2])+int(s[2])]
            else:
                if(submissionData not in incorrectSubmissions):
                    incorrectSubmissions[submissionData] = s[2]

                if(contestant not in contestants):
                    contestants[contestant] = [int(contestant), 0, 0]
        else:
```

```
40              if(contestant not in contestants):
41                  contestants[contestant] = [int(contestant), 0, 0]
42
43      contestants = [v for v in contestants.values()]
44      contestants = sorted(contestants, key=itemgetter(0))
45      contestants = sorted(contestants, key=itemgetter(2))
46      contestants = sorted(contestants, key=itemgetter(1), reverse=True)
47
48      for c in contestants:
49          print(c[0], c[1], c[2])
50
51
52  def main():
53      try:
54          line = input()
55          numberOfTests = int(line)
56          blankLine = input()
57          for i in range(numberOfTests):
58              submissions = []
59              line = input()
60              while(line != ""):
61                  try:
62                      submissions.append(line.split(' '))
63                      line = input()
64                  except EOFError:
65                      break
66              solve(submissions)
67              if (i != numberOfTests - 1):
68                  print("")
69      except EOFError:
70          print("No lines")
71
72
73  if __name__ == "__main__":
74      main()
```

3.5 Manipulação de bits

3.5.1 UVa 10264 – The Most Potent Corner

Este problema tem o objetivo de encontrar a vizinhança (aresta) do cubo que tem a potência mais alta. Para encontrar essa aresta mais potente são calculadas as potências de todos os vértices e a partir daí encontrar dentre essas potências, a aresta mais potente (os dois vértices vizinhos que somados tem a maior potência). Após ser encontrado, esse valor da potência é printado.

```python
# Problem: UVA 10264 - The Most Potent Corner
# Author(s): Edson Boldrini


def solve(numberOfDimensions, weights):
    potencies = {}
    for w in weights:
        acc = 0
        for i in range(len(w)):
            if (w[i] == '0'):
                neighbour = w[:i] + '1' + w[i + 1:]
            elif(w[i] == '1'):
                neighbour = w[:i] + '0' + w[i + 1:]
            acc += weights[neighbour]
        potencies[w] = acc
    maxPotency = 0
    for p in potencies:
        for i in range(len(p)):
            if (p[i] == '0'):
                neighbour = p[:i] + '1' + p[i + 1:]
            elif(p[i] == '1'):
                neighbour = p[:i] + '0' + p[i + 1:]
            neighbouringPotency = potencies[p] + potencies[neighbour]
            if(neighbouringPotency > maxPotency):
                maxPotency = neighbouringPotency
    print(maxPotency)


def main():
    try:
        line = input()
        numberOfDimensions = int(line)
        while(line != ""):
            try:
                weights = {}
                for i in range(2**numberOfDimensions):
                    try:
                        line = input()
                        weights[(bin(i).replace('0b', '')).rjust(
                            numberOfDimensions, '0')] = int(line)
                    except EOFError:
                        break
                solve(numberOfDimensions, weights)
                line = input()
                numberOfDimensions = int(line)
            except EOFError:
                break
```

```
48      except EOFError:
49          print("No lines")
50
51
52  if __name__ == "__main__":
53      main()
```

3.5.2 UVa 11926 – Multitasking

Este problema tem objetivo determinar se as tarefas planejadas de uma pessoa tem conflito de horário ou se não tem. Para resolver esse problema transformei as tarefas repetitivas em tarefas de execução única, trazendo as tarefas repetitivas para a linha to tempo de tarefas de execução única definindo como elas se comportariam no contexto de tarefas de execução única. Após isso, é verificado se na linha do tempo gerada as tarefas tem conflito ou não, inserindo uma a uma até a finalização da lista de tarefas.

```
1   # Problem: UVA 11926 - Multitasking
2   # Author(s): Edson Boldrini
3
4
5   def solve(oneTimeTasks, repeatingTasks):
6       for rt in repeatingTasks:
7           start = int(rt.split(" ")[0])
8           end = int(rt.split(" ")[1])
9           repeat = int(rt.split(" ")[2])
10          i = 0
11          while(i + start + repeat <= 1000000 + repeat):
12              oneTimeTasks.append(str(start + i) + " " + str(end + i))
13              i += repeat
14
15      conflict = False
16      tasksDone = []
17      for ott in oneTimeTasks:
18          start = int(ott.split(" ")[0])
19          end = int(ott.split(" ")[1])
20          for td in tasksDone:
21              if((start < td[0] and end > td[1]) or (start >= td[0] and end <= td[1]) or (start > td[0] and start <
22                  conflict = True
23                  break
24          if (not conflict):
25              tasksDone.append((start, end))
26          else:
27              break
28
```

```
    if(conflict):
        print("CONFLICT")
    else:
        print("NO CONFLICT")


def main():
    try:
        line = input()
        while(line != "0 0"):
            try:
                oneTimeTasksCount = int(line.split(" ")[0])
                repeatingTasksCount = int(line.split(" ")[1])
                oneTimeTasks = []
                repeatingTasks = []
                for i in range(oneTimeTasksCount):
                    line = input()
                    oneTimeTasks.append(line)
                for j in range(repeatingTasksCount):
                    line = input()
                    repeatingTasks.append(line)
                solve(oneTimeTasks, repeatingTasks)
                line = input()
            except EOFError:
                break
    except EOFError:
        print("No lines")


if __name__ == "__main__":
    main()
```

3.6 Lista Encadeada

3.6.1 UVa 11988 – Broken Keyboard (a.k.a. Beiju Text)

Este problema tem objetivo de mostrar qual é o texto final gerado pela digitação de um usuário com problema no teclado. Para essa solução foi pensado uma variável chamada "cursor" que faz o papel de escrever o caractere que veio da lista de entrada no novo texto na posição que a variável cursor informa. Cursor é incrementado em 1 para que o texto possa seguir o fluxo em frente quando não há erros do teclado, representados pelos caracteres "[" (keyboard home button) e "]" (keyboard end button).

```python
# Problem: UVA 11988 - Broken Keyboard (a.k.a. Beiju Text)
# Author(s): Edson Boldrini


def solve(sentence):
    resultSentence = ""
    cursor = len(resultSentence)
    for c in sentence:
        if(c == "["):
            cursor = 0
        elif(c == "]"):
            cursor = len(resultSentence)
        else:
            if (cursor == 0):
                resultSentence = c + resultSentence
            elif (cursor > 0 and cursor < len(resultSentence)):
                resultSentence = resultSentence[:cursor] + \
                    c + resultSentence[cursor:]
            elif (cursor == len(resultSentence)):
                resultSentence = resultSentence + c
            cursor += 1

    print(resultSentence)


def main():
    try:
        line = input()
        while(line != ""):
            try:
                solve(line)
                line = input()
            except EOFError:
                break
    except EOFError:
        print("No lines")


if __name__ == "__main__":
    main()
```

3.7 Pilhas

3.7.1 UVa 00514 – Rails

Este problema tem objetivo de informar se é possível reorganizar os vagões da estação de trem. É um programa simples de pilha onde caso a pilha de trens na estação esteja vazia ao fim das organizações significa que é possível (print "Yes", do contrário, não é possivel (print "No").

```
# Problem: UVA 514 - Rails
# Author(s): Edson Boldrini


def solve(N, cars):
    stack = []

    while(len(stack) > 0):
        stack.pop()
    j = 0
    for i in range(N):
        c = cars[i]
        if(c == 0):
            break

        while(j < N and j != c):
            if(len(stack) > 0 and stack[-1] == c):
                break
            j += 1
            stack.append(j)
        if(len(stack) > 0 and stack[-1] == c):
            stack.pop()
    if(len(stack) == 0):
        print("Yes")
    else:
        print("No")


def main():
    try:
        line = input()
        while(line != "0"):
            try:
                N = int(line)
                cars = []
                line = input()
                while(line != "0"):
                    try:
                        cars = [int(i) for i in line.split(" ")]
```

```
                        solve(N, cars)
                        line = input()
                    except EOFError:
                        break
                print("")
                line = input()
            except EOFError:
                break
    except EOFError:
        print("No lines")


if __name__ == "__main__":
    main()
```

3.7.2 UVa 01062 – Containers

Este problema visa calcular quantas pilhas são necessárias para poder carregar as embarcações por ordem alfabética. É um outro programa simples de pilha que aloca uma pilha nova cada vez que uma letra nova que ainda não tem sua pilha alocada aparece na sequência.

```
# Problem: UVA 1062 - Containers
# Author(s): Edson Boldrini


def solve(containers):
    stacks = []
    for c in containers:
        pushed = False
        for j in range(len(stacks)):
            if (stacks[j][-1] >= c):
                pushed = True
                stacks[j].append(c)
                break
        if (not pushed):
            stacks.append([c])

    return len(stacks)


def main():
    try:
        i = 1
        stacks = 0
```

```
24          line = input()
25          while(line != "end"):
26              try:
27                  containers = line
28                  stacks = solve(containers)
29                  print("Case %d: %d" % (i, stacks))
30                  i += 1
31                  line = input()
32              except EOFError:
33                  break
34      except EOFError:
35          print("No lines")
36
37
38  if __name__ == "__main__":
39      main()
```