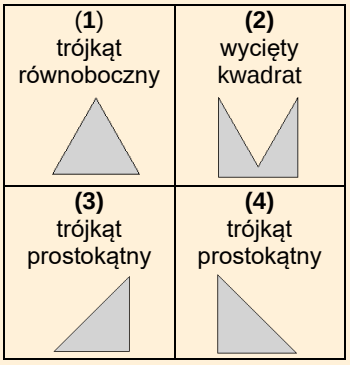
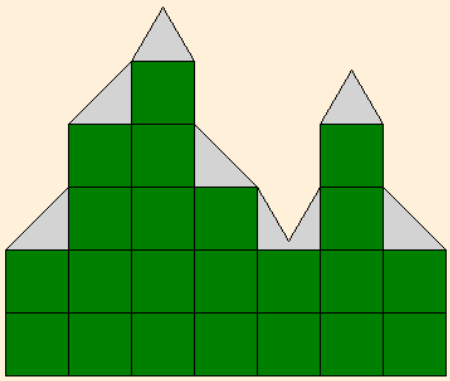
Zadanie góry – LOGIA 21 (2020/21), etap 2

Treść zadania

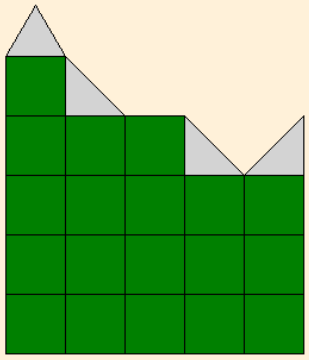
Janek chodzi po górach i oznacza liczbami naturalnymi od 1 do 9 kolejno odwiedzane wierzchołki proporcjonalnie do ich wysokości. Potem rysuje schemat, którym ilustruje zanotowane dane. Jeśli wierzchołek jest oznaczony liczbą **n**, to rysuje kolumnę **n** zielonych kwadratów o długości boku **50**. Ponadto dodaje szare zakończenie: trójkąt równoboczny (**1**), gdy sąsiednie wysokości są niższe, kwadrat z wyciętym trójkątem równobocznym (**2**), gdy są wyższe, trójkąt prostokątny (**3**), gdy tylko następna wysokość jest wyższa i trójkąt prostokątny (**4**), gdy tylko poprzednia wysokość jest wyższa. Jeśli żaden z tych warunków nie zachodzi, to nie rysuje szarego zakończenia. Trasę Janek może pokonywać wielokrotnie, więc „lewym sąsiadem" dla pierwszego wierzchołka jest ostatni, a „prawym sąsiadem" dla ostatniego pierwszy.



Pomóż Jankowi i napisz funkcję **schemat(lista)** rysującą schemat według opisanych zasad. Parametrem funkcji jest lista o długości od 3 do 20 liczb w zakresie od 1 do 9. Rysunek powinien być jednakowo oddalony od lewej i prawej krawędzi ekranu.



schemat([2, 4, 5, 3, 2, 4, 2])



schemat([5, 4, 4, 3, 3])

Omówienie rozwiązania

Rozwiązanie zadanie polega na przejrzeniu wszystkich elementów parametru **lista** i narysowaniu wskazanej liczby kwadratów wraz z zakończeniem. Liczbę kwadratów w danej kolumnie ustalamy na podstawie wartości rozpatrywanego elementu listy, a zakończenie również na podstawie wartości sąsiednich elementów. Należy zadbać o rozpatrywanie warunków w kolejności zgodnej z podaną w treści zadania: obaj sąsiedzi niżsi, obaj wyżsi, gdy tylko następna wysokość jest wyższa i gdy tylko poprzednia wysokość jest wyższa. Jeśli żaden z tych warunków nie zachodzi, to nie rysujemy zakończenia.

Pewną trudność może też stanowić pierwszy i ostatni element listy, gdyż należy właściwie określić ich sąsiadów. Warto skorzystać z operacji modulo. Dokładniej sąsiadami dla elementu **i** poza pierwszym i ostatnim jest element **i-1** oraz **i+1**. Natomiast dla elementu **0** jest to **n-1**, gdzie **n** oznacza długość listy oraz **i+1**, a dla elementu **n-1**, to **n-2** oraz **0**. Można ogólnie zapisać, że sąsiadami dla elementu **i** są odpowiednio elementy **(i-1)% n** oraz **(i+1)% n**.

Rozwiązanie w języku Python

Przy definiowaniu głównej funkcji **schemat(lista)** warto skorzystać z funkcji pomocniczych: **kwadrat(bok)** oraz **elt1(bok)**, … , **elt4(bok)**, które będą rysowały kwadrat oraz cztery rodzaje elementów kończących. Do prawidłowego działania skryptu należy zaimportować moduły **turtle** i funkcję **sqrt** z modułu **math**. Jeśli zaczynamy rysowanie figury, która będzie zamalowana należy skorzystać z polecenia **begin_fill()**, gdy kończymy **end_fill()**. Wcześniej jednak ustawiamy kolor obwódki i kolor zamalowania.

Rysunek powinien być jednakowo oddalony od lewej i prawej krawędzi ekranu. Skoro zaczynamy od lewego dolnego rogu, to na początku należy przesunąć żółwia o połowę szerokości rysunku, czyli **n*50/2**. W rozwiązaniu zastosowana została instrukcja **if … elif**, która pozwala wygodnie zapisać kilka warunków.

```
1.  from turtle import *
2.  from math import sqrt
3.
4.  def kwadrat(bok):
5.      begin_fill()
6.      for i in range(4):
7.          fd(bok); rt(90)
8.      end_fill()
9.
10. def elt1(bok):
11.     begin_fill()
12.     rt(30)
13.     for i in range(3):
14.         fd(bok); rt(120)
15.     lt(30)
16.     end_fill()
17.
18. def elt2(bok):
19.     begin_fill()
20.     fd(bok); rt(150)
21.     fd(bok); lt(120)
22.     fd(bok); rt(150)
23.     fd(bok); rt(90)
24.     fd(bok); rt(90)
25.     end_fill()
26.
```

```
27. def elt3(bok):
28.     begin_fill()
29.     rt(45)
30.     fd(bok * sqrt(2)); rt(135)
31.     fd(bok); rt(90)
32.     fd(bok); rt(90)
33.     end_fill()
34.
35. def elt4(bok):
36.     begin_fill()
37.     fd(bok); rt(135)
38.     fd(bok * sqrt(2)); rt(135)
39.     fd(bok); rt(90)
40.     end_fill()
41.
42. def schemat(lista):
43.     a = 50
44.     n = len(lista)
45.     # wyśrodkowanie rysunku
46.     pu(); bk(n * 50 / 2); lt(90); bk(200); rt(90); pd()
47.     for i in range(n):
48.         lt(90)
49.         color("black","green")
50.         for j in range(lista[i]):
51.             kwadrat(a)
52.             fd(a)
53.         color("black","lightgray")
54.         if lista[(i - 1) % n] < lista[i] > lista[(i + 1) % n]:
55.             elt1(a)
56.         elif lista[(i - 1) % n] > lista[i] < lista[(i + 1) % n]:
57.             elt2(a)
58.         elif lista[i] < lista[(i + 1) % n]:
59.             elt3(a)
60.         elif lista[i] < lista[(i - 1) % n]:
61.             elt4(a)
62.         fd(-a * (lista[i]))
63.         rt(90); fd(a)
```

Testy

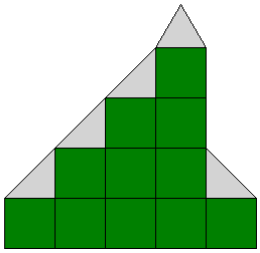
Program testujemy wydając sekwencję poleceń:

```
tracer(0); schemat(lista); update()
```

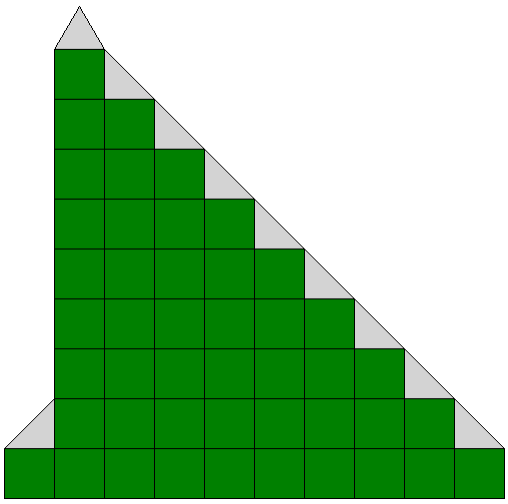
gdzie parametr **`lista`** będzie przyjmować różne wartości.

Przykładowe testy powinny obejmować różne długości tras, a w ramach jednej trasy różne przypadki zakończeń elementów wewnątrz listy i na jej brzegach. Poniżej przykładowe wywołania.

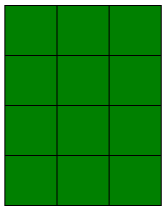
```
schemat([1, 2, 3, 4, 1])
```



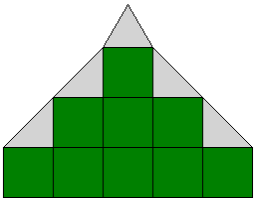
```
schemat ([1, 9, 8, 7, 6, 5, 4, 3, 2, 1])
```



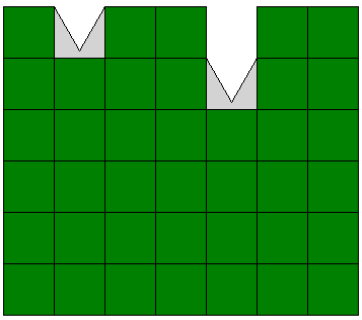
```
schemat([4, 4, 4])
```



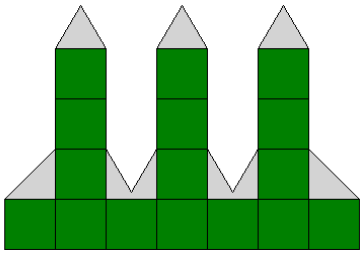
```
schemat([1, 2, 3, 2, 1])
```



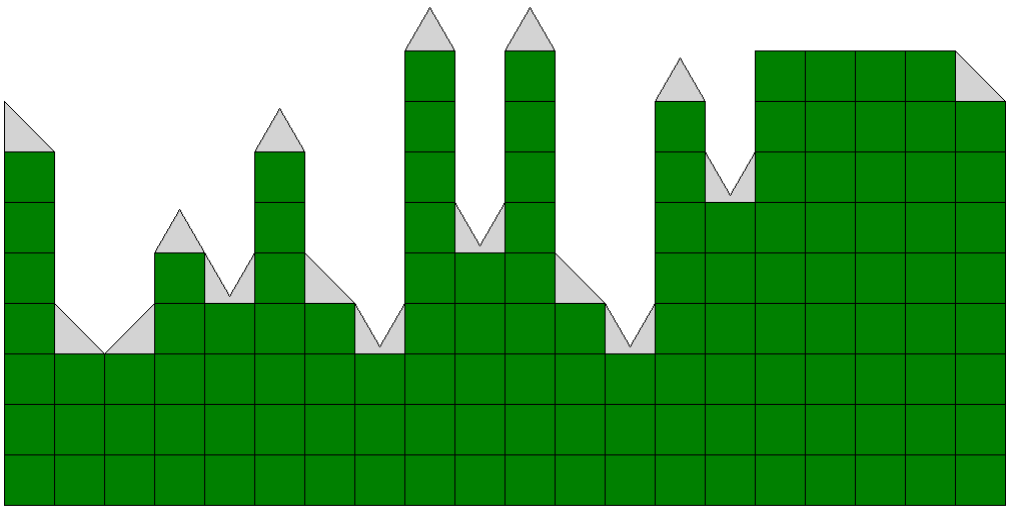
```
schemat([6, 5, 6, 6, 4, 6, 6])
```



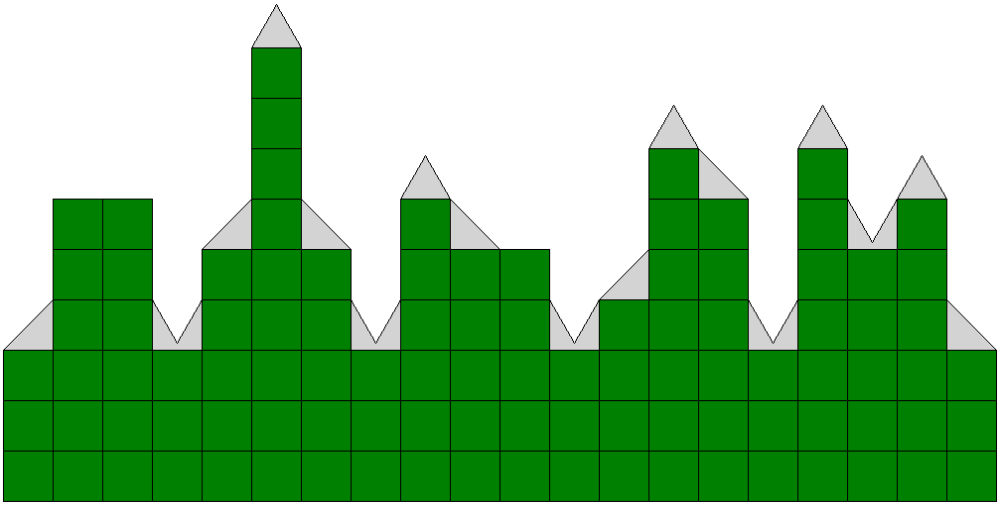
```
schemat([1, 4, 1, 4, 1, 4, 1])
```



```
schemat([7, 3, 3, 5, 4, 7, 4, 3, 9, 5, 9, 4, 3, 8, 6, 9, 9, 9, 9, 8])
```



```
schemat([3, 6, 6, 3, 5, 9, 5, 3, 6, 5, 5, 3, 4, 7, 6, 3, 7, 5, 6, 3])
```



Zadanie Papadam – LOGIA 21 (2020/21), etap 2

Treść zadania

Papadam to chrupki naleśnik indyjski. Bajtek lubi takie naleśniki, dlatego przed przyjazdem do mamy wysyła jej zakodowany napis złożony z małych liter *p*, *a*, *d* i *m* z prośbą o przygotowanie tylu naleśników, ile ukrytych słów *papadam* znajduje się w napisie. Na przykład w napisie: *ppapadaaaamaaapaappaddmppaapadaaaampppaa* są ukryte dwa słowa *papadam* (zaznaczone kolorem zielonym). Napisz program, który pomoże mamie obliczyć, o ile naleśników prosi Bajtek.

Wejście:

Niepusty napis składający się z małych liter *p*, *a*, *d* i *m* o długości nie większej niż 10 000.

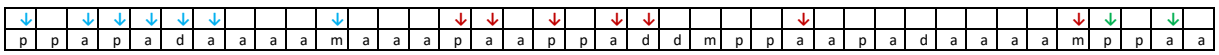
Wyjście:

Liczba określająca, ile razy w podanym napisie można wyodrębnić słowo *papadam*.

	Przykład 1	
Wejście	ppapadaaaamaaapaappaddmppaapadaaaamppaa	
Wyjście	2	
	Przykład 2	*Przykład 3*
Wejście	pppppapaddaaaam	papaadm
Wyjście	1	0

Omówienie rozwiązania

Zastanówmy się jak najprościej rozwiązać ten problem. Na pewno musimy przejrzeć cały napis podany na wejściu. Zmienna **ile**, która będzie zawierała liczbę znalezionych słów 'papadam' w napisie przyjmie na początku wartość 0. Przeglądając napis, jednocześnie szukamy pierwszej litery słowa 'papadam', czyli 'p' w napisie, po znalezieniu jej szukamy kolejnej, czyli 'a', następnie znowu 'p' itd., po znalezieniu wszystkich siedmiu liter podanego słowa 'papadam' możemy zwiększyć wartość zmiennej **ile** o **1**, ponieważ zostało znalezione całe słowo w podanym napisie. Wtedy kontynuując przeglądanie napisu szukamy ponownie pierwszej litery słowa i dalej postępujemy identycznie jak poprzednio.



W tabeli zobrazowano przykład z treści zadania. Niebieska strzałka wskazuje kolejne znalezione litery pierwszego słowa 'papadam', strzałka czerwona drugiego, dwie zielone strzałki wskazują na dwie pierwsze litery 'p' i 'a' znalezione w podanym słowie. Tym razem wartość zmiennej **ile** nie zostanie zwiększona, ponieważ nie zostało znalezione całe słowo 'papadam'.

Rozwiązanie ma złożoność liniową, ponieważ wczytane słowo **s** zostało jeden raz przejrzane.

Rozwiązanie w języku Python

```
1.  s = input()
2.  ile = 0
3.  k = 0
4.  na = 'papadam'
5.  for i in range(len(s)):
6.      if s[i] == na[k]:
7.          k += 1
8.          if k == 7:
9.              k = 0
10.             ile += 1
11. print(ile)
```

Testy

Sprawdzamy działanie napisanego programu dla danych z treści zadania oraz innych danych.

Na przykład:

wejście → ppapadamm
wyjście → 1

wejście →
mmmmmmmmmmmmmmmmmmmddddddddddddddddddddddddddddddddddddddddddddaaaaaaa
aaaaaaaaaaaaaaaaammmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmm
wyjście → 0

wejście → ppaaapapdaaampppappaaadapmpapaapadpamaa
wyjście → 3

wejście → papadampapadampapadampapadam
wyjście → 4

Zadanie Wagi napisów – LOGIA 21 (2020/21), etap 2

Treść zadania

Tolek każdej literze w słowie przypisuje liczbę całkowitą nieujemną, którą nazywa wagą litery. Pierwsza i ostatnia litera słowa ma wagę **1**, druga i przedostatnia wagę **2**, trzecia od początku i końca wagę **3**, itd. Następnie wszystkim samogłoskom redukuje wagę o jeden, a spółgłoskom zwiększa wagę o jeden. Wagę słowa oblicza sumując wszystkie wagi liter. Napisz program, który ułoży słowa według ich wag od najmniejszej do największej. Jeśli dwa słowa mają taką samą wagę, to występują wtedy w kolejności alfabetycznej.

Wejście:

W pierwszym i jedynym wierszu znajduje się od 3 do 1000 słów oddzielonych pojedynczymi odstępami, o długości od 2 do 30, składających się z małych liter alfabetu łacińskiego.

Wyjście:

Jeden wiersz zawierający słowa uporządkowane według reguł opisanych w treści zadania. Słowa oddzielone są pojedynczymi odstępami.

	Przykład 1	*Przykład 2*	*Przykład 3*
Wejście	**ala ma kota**	**wykonaj zadanie waga slowa**	**aaa cc am la**
Wyjście	**ma ala kota**	**waga slowa zadanie wykonaj**	**aaa am la cc**

Wyjaśnienie do przykładu 1:

	ala	**ma**	**kota**
wagi na podstawie położenia	1+2+1	1+1	1+2+2+1
wagi po uwzględnieniu typu litery	0+3+0	2+0	2+1+3+0
waga słowa	3	2	6

Omówienie rozwiązania

W zadaniu należało posortować dane według dwóch kryteriów. Pierwszym z nich była waga opisana w treści zadania, drugim kolejność alfabetyczna.

Waga słowa zależy od długości słowa oraz liczby samogłosek i spółgłosek w nim zawartych. Nie ma natomiast znaczenia położenie liter w ciągu znaków. Na przykład dla słów *kot* i *kto* waga wynosi 5.

Do obliczenia wagi na podstawie położenia możemy skorzystać ze wzoru na sumę ciągu arytmetycznego lub zastosować iterację. Niezależnie od wybranego sposobu należy rozpatrzyć dwa przypadki – dla słów parzystej długości oraz nieparzystej długości.

Algorytm iteracyjny obliczający wagę bez uwzględniania typu liter:

```
1. suma ← 0
2. dla każdego i od 1 do (długość słowa) // 2
3.     suma ← suma + 2 * i
4. jeśli (długość słowa) % 2 == 1
5.     suma ← suma + (długość słowa) // 2 + 1
6. wynik suma
```

W wersji bez iteracji:

```
1. suma ← (1 + (długość słowa) // 2) * ((długość słowa) // 2)
2. jeśli (długość słowa) % 2 == 1
3.     suma ← suma + (długość słowa) // 2 + 1
4. wynik suma
```

Wystarczy jeszcze zliczyć liczbę samogłosek w słowie. Liczbę spółgłosek otrzymamy odejmując od długości słowa liczbę samogłosek. Algorytm zliczający samogłoski może wyglądać następująco:

```
1. l_samoglosek ← 0
2. dla każdego znaku w słowie
3.     jeśli znak jest samogłoską, to
4.         l_samoglosek ← l_samoglosek + 1
5. wynik l_samogłosek
```

Waga uwzględniająca typ liter obliczona zostanie ze wzoru:

waga bez uwzględnienia typu liter – l_samogłosek + (długość słowa – l_samogłosek)

czyli

*waga bez uwzględnienia typu liter + długość słowa – 2 * l_samogłosek*

Po wyznaczeniu wag możemy zaimplementować dowolny algorytm sortowania, np. bąbelkowy.

Rozwiązanie w języku Python

Funkcja zliczająca samogłoski:

```
1. def ile(slowo):
2.     l_samoglosek = 0
3.     for zn in slowo:
4.         if zn in "aeiouy":
5.             l_samoglosek += 1
6.     return l_samoglosek
```

Można ją zastąpić listą składaną:

```
1. sum([1 for zn in slowo if zn in "aeiouy"])
```

Funkcja obliczająca wagę słowa:

```
1. def waga(slowo):
2.     ile = sum([1 for zn in slowo if zn in "aeiouy"])
3.     suma = (1 + len(slowo) // 2) * (len(slowo) // 2)
4.     if len(slowo) % 2 == 1:
5.         suma = suma + len(slowo) // 2 + 1
6.     return suma + len(slowo) - 2 * ile
```

Rozwiązanie zadania w oparciu o sortowanie bąbelkowe:

```
def sortuj(lista):
    for i in range(0, len(lista) - 1):
        for j in range(len(lista) - 1 - i):
            if waga(lista[j]) > waga(lista[j + 1]):
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
            elif waga(lista[j]) == waga(lista[j + 1]) and lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista

dane = input().split()
wynik = sortuj(dane)
for ele in wynik:
    print(ele, end=" ")
```

Testy

Testy zostały podzielone na 5 grup. Pierwsze trzy grupy zwierały słowa o różnych wagach, a dwie ostatnie słowa o powtarzających się wagach w celu sprawdzenia poprawności rozwiązania (uwzględnienie dwóch kryteriów sortowania). Ponadto testy uwzględniały ciągi znaków różnej długości oraz różną liczbę słów.

Grupa testów	Różne wagi	Długość słowa	Liczba słów
I	tak	od 2 do 4	3
II	tak	od 4 do 9	od 8 do 10
III	tak	od 2 do 15	50
IV	nie	wszystkie słowa tej samej długości	500
V	nie	od 2 do 30	1000

Zadanie Liczby czterocyfrowe – LOGIA 21 (2020/21), etap 2

Treść zadania

Basia analizuje liczby czterocyfrowe. Dla danej liczby porządkuje jej cyfry od największej do najmniejszej i od najmniejszej do największej. Potem odejmuje otrzymane liczby od siebie (od większej mniejszą). Z wynikiem postępuje tak samo, aż do momentu, gdy zauważy, że liczba się nie zmienia.

Na przykład, gdy zacznie od liczby 1234 otrzymuje kolejne wyniki odejmowania 4321 – 1234 = 3087; 8730 – 378 = 8352; 8532 – 2358 = 6174. Dalej zauważa, że już liczba się nie zmienia: 7641 – 1467 = 6174. Napisz program, który policzy, po ilu krokach Basia otrzyma stałą wartość.

Wejście:

Liczba całkowita z zakresu od 1000 do 9999.

Wyjście:

Liczba określająca, po ilu działaniach odejmowania Basia otrzyma stałą wartość zgodnie z regułami opisanymi w treści zadania.

	Przykład 1	*Przykład 2*	*Przykład 3*
Wejście	**1234**	**2222**	**1224**
Wyjście	**3**	**1**	**6**

Omówienie rozwiązania

Zadanie można rozwiązać według opisu podanego w treści – dla danej liczby porządkujemy jej cyfry malejąco i rosnąco, znajdujemy różnicę liczb i sprawdzamy, czy ona się zmienia. Jeśli się nie zmienia, to kończymy program, gdy się zmienia zwiększamy licznik iteracji o 1 i do dalszego rozważania bierzemy nową liczbę. Postępujemy tak długo, aż nie otrzymamy wartości stałej.

Możemy zauważyć, że są dwie klasy danych wejściowych. Pierwsza klasa to liczby o jednakowych cyfrach np. 1111, wtedy różnica wynosi 0. Druga klasa to liczby, w których występują co najmniej dwie różne cyfry, wtedy w co najwyżej kilku krokach dochodzimy do liczby 6174.

Rozwiązanie w języku Python

Wynikiem zadania jest wartość funkcji **`petla(x)`** dla liczby odczytanej z wejścia – **`print(petla(int(input())))`**. W rozwiązaniu wykorzystana jest funkcja **`nast(liczba)`** generująca kolejny element. Dodatkowo zdefiniowana została funkcja **`naliste(x)`**, która zamienia liczbę na listę cyfr i funkcja odwrotna **`naliczbe(x)`**, zamieniająca listę na liczbę. Zamiany dokonujemy po to, by skorzystać z wbudowanej funkcji **`sorted()`**, która sortuje listę liczb. Zamianę liczby na listę dokonujemy poprzez wyliczenie i zapamiętanie ostatniej cyfry bieżącej liczby – pomocna jest tu operacja znajdowania reszty z dzielenia `%`. Przy operacji odwrotnej mnożymy przez 10 i dodajemy kolejną cyfrę.

Zapis `x[::-1]` powoduje odwrócenie listy, czyli z listy posortowanej rosnąco otrzymujemy listę posortowaną malejąco.

```
1.  # zamiana liczby na listę cyfr
2.  def naliste(x):
3.      pom = []
4.      while x > 0:
5.          pom = [x % 10] + pom
6.          x = x // 10
7.      return pom
8.
9.  # zamiana listy cyfr na liczbę
10. def naliczbe(x):
11.     pom = 0
12.     for i in range(len(x)):
13.         pom = pom * 10 + x[i]
14.     return pom
15.
16. # generowanie następnego elementu
17. def nast(liczba):
18.     x = naliste(liczba)
19.     x = sorted(x)
20.     a = naliczbe(x)
21.     b = naliczbe(x[::-1])
22.     return b - a
23.
24. # generowanie, aż do stałego elementu
25. def petla(x):
26.     ile = 0
27.     y = nast(x)
28.     while x != y:
29.         x = y
30.         y = nast(x)
31.         ile += 1
32.     return ile
33.
34. print(petla(int(input())))
```

Testy

Przy układaniu testów warto uwzględnić przypadek, gdy wszystkie cyfry są jednakowe, gdy na wejściu jest liczba 6174 oraz pozostałe przypadki. Wywołujemy program dla następujących testów.

Wejście	Wyjście
1111	1
9834	4
5689	2
7491	5
9998	2
3434	4
6174	0
4536	3
2887	7
2888	6