Zadanie Cukierki – LOGIA 18 (2017/18), etap 2

Treść zadania

Adam ma papierową torebkę. Wrzuca do niej cukierki, a następnie częstuje nimi znajomych. Cukierki są wrzucane i wyciągane pojedynczo. Postępuje tak wielokrotnie: wrzuca i częstuje. Adam zastanawia się, ile razy w torebce było dokładnie **n** cukierków. Napisz dwuparametrową funkcję **ilerazy**, której pierwszym parametrem jest **n** (od 1 do 10000). Drugim parametrem funkcji jest parzystej długości lista dodatnich liczb całkowitych, zawierająca na przemian liczby wrzucanych i wyjmowanych cukierków. Na przykład lista zawierająca 4, 1, 3 i 2 oznacza, że kolejno do torebki zostały wrzucone cztery cukierki, wyjęty jeden, wrzucone trzy i wyjęte dwa. Długość listy wynosi od 2 do 1000. Wynikiem funkcji jest liczba określająca, ile razy w torebce było dokładnie **n** cukierków.

Przykłady:

Python:	Logo:
wynikiem **ilerazy (2, [3, 2])** jest **2**,	wynikiem **ilerazy 2 [3 2]** jest **2**,
wynikiem **ilerazy (4, [5, 3, 5, 3, 2, 1])** jest **4**.	wynikiem **ilerazy 4 [5 3 5 3 2 1]** jest **4**.

W pierwszym przykładzie liczba cukierków w torebce wynosi kolejno: 0, 1, 2, 3, 2, 1.

W drugim przykładzie liczba cukierków w torebce wynosi kolejno: 0, 1, 2, 3, 4, 5, 4, 3, 2, 3, 4, 5, 6, 7, 6, 5, 4, 5, 6, 5.

Omówienie rozwiązania

Zadanie polega na policzeniu, ile razy określona liczba cukierków znajdowała się w torebce. W związku z tym potrzebna będzie zmienna, która będzie to pamiętać (jej wartość początkowa wynosi 0). W drugiej zmiennej będziemy pamiętać aktualną liczbę cukierków znajdujących się w torebce. Jej wartość początkowa także wynosi 0, ponieważ na początku torebka jest pusta. Zwróćmy uwagę, że mimo iż cukierki są wkładane i wyciągane pojedynczo, możemy jeden element listy potraktować całościowo. Jeśli przed włożeniem cukierków ich liczba w torebce była mniejsza niż **n**, a po włożeniu większa lub równa **n**, tzn., że w trakcie wkładania w torebce znajdowało się dokładnie **n** cukierków. A więc licznik należy powiększyć o jeden. Analogicznie, jeśli przed wyjmowaniem cukierków ich liczba w torebce była większa od **n**, a po wyjęciu mniejsza lub równa, to w trakcie wyjmowania znajdowało się dokładnie **n** cukierków. Za każdym razem modyfikujemy liczbę cukierków znajdujących się w torebce. Do rozwiązania zadania wystarczy więc pojedyncza pętla przeglądająca listę i w zależności od parzystości indeksu elementu wykonująca opisane działania.

Rozwiązanie w języku Python

```python
def ilerazy(n, lista):
    ilen = 0
    suma = 0
    for i in range(len(lista)):
        if i % 2 == 0:
            if suma < n and suma + lista[i] >= n:
                ilen += 1
            suma += lista[i]
        else:
            if suma > n and suma - lista[i] <= n:
                ilen += 1
            suma -= lista[i]
    return ilen
```

Elementy listy w Pythonie są indeksowane (numerowane) od 0, w związku z tym parzysty indeks oznacza wkładanie cukierków do torebki, a nieparzysty wyjmowanie.

Rozwiązanie w języku Logo

```
oto ilerazy :n :lista
  niech "ilen 0
  niech "suma 0
  powtórz długość :lista
    [
      jeżeli reszta npw 2 = 1
        [
          jeśli i :suma < :n (:suma + element npw :lista) >= :n [ zwiększ "ilen ]
          (zwiększ "suma element npw :lista)
        ]
        [
          jeśli i :suma > :n (:suma - element npw :lista) <= :n [ zwiększ "ilen ]
          (zmniejsz "suma element npw :lista)
        ]
    ]
  wy :ilen
już
```

Elementy listy w Logo są numerowane od jeden, w związku z tym nieparzysta wartość numeru powtórzenia pętli oznacza wkładanie cukierków do torebki, a parzysta wyjmowanie.

Testy

Testowanie rozwiązania warto rozpocząć od przykładów, których wyniki można prosto sprawdzić ręcznie np. podobnych, jak w treści zadania. Potem można testować działanie programu dla większych danych, choć i w tym przypadku warto tak dobrać dane, aby można było ocenić poprawność wyniku. W testach powinny być ujęte wszystkie możliwe przypadki, czyli: dodanie lub wyjęcie cukierków powoduje przejście przez wartość graniczną **n**, dodanie lub wyjęcie cukierków nie powoduje przejścia przez wartość graniczną.

Python	Logo	
ilerazy(1, [2, 1])	ilerazy 1 [2 1]	2
ilerazy(2, [3, 2, 1, 1, 2, 3])	ilerazy 2 [3 2 1 1 2 3]	5
ilerazy(100, [90, 80])	ilerazy 100 [90 80]	0
ilerazy(1, [2, 1, 2, 2, 3, 3])	ilerazy 1 [2 1 2 2 3 3]	4
ilerazy(2, [2, 1, 2, 1, 2, 1, 1, 2])	ilerazy 2 [2 1 2 1 2 1 1 2]	4
ilerazy(6, [3, 1, 3, 1, 3, 2, 3, 1])	ilerazy 6 [3 1 3 1 3 2 3 1]	3
ilerazy(51, [100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1])	ilerazy 51 [100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1]	99

Zadanie Dukaty – miniLOGIA 16 (2017/18), etap 2

Treść zadania

Bajtek planuje systematycznie oszczędzać. Będzie codziennie rano dokładać jednego dukata do skarbonki, a co piąty dzień dołoży również kolejne dwa dukaty. W momencie, gdy w skarbonce uzbiera się co najmniej 50 dukatów Bajtek będzie z niej pobierał 3/4 dukatów na zakup ulubionych książek. Dukat jest najmniejszą jednostką, dlatego Bajtek pobierze ze skarbonki maksymalnie dużo dukatów, ale nie więcej niż zaplanował. Zdefiniuj jednoparametrową funkcję **dukaty**, która pomoże Bajtkowi kontrolować zawartość skarbonki. Wynikiem funkcji jest zawartość skarbonki w dniu podanym jako parametr, po dokonanych operacjach.

Przykłady:

Python:	Logo:
wynikiem **dukaty(3)** jest **3**,	wynikiem **dukaty 3** jest **3**,
wynikiem **dukaty(6)** jest **8**,	wynikiem **dukaty 6** jest **8**,
wynikiem **dukaty(36)** jest **13**.	wynikiem **dukaty 36** jest **13**.

Omówienie rozwiązania

Zadanie dukaty należy do grupy zadań symulacyjnych. Analizując treść zadania, należy zwrócić uwagę na kluczowe elementy związane z oszczędzaniem. Zawartość skarbonki zmienia się w następujący sposób:

- codziennie przybywa jeden dukat,
- co piąty dzień kolejne dwa dukaty,
- jeśli w skarbonce jest co najmniej 50 dukatów pobieramy nie więcej niż ¾ dukatów, ale maksymalnie dużo, pamiętając o tym, że dukaty są niepodzielne.

Symulacja, rozpisana na kolejne dni i wykonywane operacje, będzie wyglądać tak:

Dzień	Dodane dukaty	Stan po dodaniu dukatów	Pobrane dukaty	Stan po pobraniu dukatów
0				0
1	1	1	0	1
2	1	2	0	2
3	**1**	**3**	**0**	**3**
4	1	4	0	4
5	1 + 2 = 3	7	0	7
6	**1**	**8**	**0**	**8**

…				
10	1 + 2 = 3	14	0	14
…				
35	1 + 2 = 3	49	0	49
36	**1**	**50**	**(3 * 50) div 4 = 37**	**50 – 37 = 13**

Każdego kolejnego dnia zmienia się zawartość skarbonki. Zapis algorytmu może być następujący:

1. stan ← 0
2. dla każdego kolejnego dnia, od 1 do ostatniego dnia
   - stan ← stan + 1
   - jeśli kolejny mod 5 = 0
     - stan ← stan + 2
   - jeśli stan >= 50
     - stan ← stan – (3 * stan) div 4
3. wynik stan

uwaga: *mod* – reszta z dzielenia całkowitego, *div* – dzielenie całkowite

Rozwiązanie w języku Python

```
1.  def dukaty(dzien):
2.      stan = 0
3.      for kolejny in range(1, dzien + 1):
4.          stan = stan + 1
5.          if kolejny % 5 == 0:
6.              stan = stan + 2
7.          if stan >= 50:
8.              stan = stan - (3 * stan) // 4
9.      return stan
```

Rozwiązanie w języku Logo

```
1.  oto dukaty :dzien
2.   niech "stan  0
3.   powtórz :dzien [
4.                   zwiększ "stan
5.                   jeśli reszta npw 5 = 0 [ (zwiększ "stan 2) ]
6.                   jeśli :stan >= 50 [ niech "stan :stan - ilorazc (3 * :stan) 4 ]
7.                  ]
8.   wy :stan
9.  już
```

W obu językach programowania rozwiązania są identyczne z dokładnością co do składni. Warto zwrócić uwagę na brak operatorów operacji modulo i ilorazu całkowitego w języku Logo. Operatory zastąpione są funkcjami, odpowiednio **reszta** dla operacji modulo oraz **ilorazc** dla ilorazu całkowitego.

Warto zauważyć, że stan skarbonki uprawniający do wypłaty to: 50 (49 + 1), 51 (48 + 3) lub 52 (49 + 3). Różnica stanu skarbonki i kwoty do wypłaty jest stała i wynosi 13:

50 - 3 * 50 div 4 = 13

51 - 3 * 51 div 4 = 13

52 - 3 * 52 div 4 = 13

Zatem stan skarbonki po wypłacie można zapisać tak:

```
stan = 13 #Python

niech "stan 13 ;Logo
```

Nigdzie w treści zadania nie napisano, że dukaty do skarbonki trafiają po jednej sztuce i stan skarbonki należy analizować po dodaniu pojedynczego dukata tak, aby wypłaty dokonać w momencie, gdy jej zawartość wyniesie dokładnie 50 (a nie co najmniej 50 – 50, 51 lub 52). Dlatego symulacje przeprowadzone w inny sposób nie dawały poprawnego wyniku.

Testy

Testowanie rozwiązania warto rozpocząć od wartości, dla których jest łatwo ręcznie policzyć wynik.

Logo	Python	
dukaty 1	dukaty(1)	1
dukaty 20	dukaty(20)	28
dukaty 5	dukaty(5)	7
dukaty 40	dukaty(40)	19
dukaty 37	dukaty(37)	14
dukaty 89	dukaty(89)	49
dukaty 1009	dukaty(1009)	29
dukaty 1234	dukaty(1234)	45
dukaty 4	dukaty(4)	4
dukaty 3000	dukaty(3000)	18

Testy zostały tak dobrane, żeby sprawdzić wszystkie możliwe przypadki zachodzące podczas symulacji:

- pierwszy i czwarty dzień,
- pierwszy dzień, w którym dołożono trzy dukaty,
- wielokrotność piątego dnia, bez dotychczasowego opróżniania skarbonki,
- dzień przed opróżnieniem skarbonki,
- następny dzień po opróżnieniu skarbonki,
- odległe dni w symulacji, gdzie w międzyczasie wielokrotnie opróżniano skarbonkę; w tych dniach dokładano do skarbonki 1 lub 3 dukaty.

Zadanie Neony – LOGIA 18 (2017/18), etap 2

Treść zadania

W Turtlandii przygotowują neon do zawieszenia na dwóch słupach. Słupy stoją w rzędzie, odległość pomiędzy dwoma sąsiednimi wynosi 2. Dział marketingu uzależnia wybór słupów od oceny zdefiniowanej jako suma ich wysokości i odległości między nimi.

Napisz jednoparametrową funkcję **neon**, której wynikiem jest najwyższa możliwa ocena. Parametr jest listą wysokości kolejnych słupów (ma co najmniej 2 elementy, co najwyżej 500). Wysokość każdego słupa wynosi od 1 do 10000.

Przykłady:

Język	**Wywołanie**	**Wynik**	Uzasadnienie
Python	`neon([10, 4, 5, 7, 1, 4, 1])`	24	najwyższa ocena będzie uzyskana przy wyborze pierwszego i przedostatniego słupa, 24 = 10 + 4 + 5 · 2
Logo	`neon [10 4 5 7 1 4 1]`		
Python	`neon([1, 10, 1])`	13	13 = 1 + 10 + 1 · 2, też 13 = 10 + 1 + 1 · 2
Logo	`neon [1 10 1]`		

Omówienie rozwiązania

Pierwszym nasuwającym się rozwiązaniem jest rozpatrzenie każdej pary słupów i wybranie tej, która uzyska najwyższą ocenę. Dokładniej, przeglądamy daną listę i dla każdego elementu, który jest dalej na liście niż rozpatrywany słup, liczymy ocenę. Najlepszą ocenę zapamiętujemy. Takie rozwiązanie wymaga dla listy długości n, n * (n-1 + n-2 +… +1) porównań. Jest to rozwiązanie o złożoności kwadratowej. Można jednak zaproponować szybsze rozwiązanie.

Drugi algorytm zakłada przeglądanie listy element po elemencie, z tym że pamiętamy dwie wartości – **`naj`** – najlepszą dotychczasową ocenę i **`pom`** – najlepszą ocenę, która uwzględnia odległość do bieżącego słupa. W danym kroku porównujemy te wartości i jeśli druga z nich jest większa, to uaktualniamy wartość maksymalną. Ponadto ustalamy nową wartość zmiennej **pom** jako maksymalną z dwóch liczb: dotychczasowej wartości i oceny neonu, który rozpoczyna się w rozpatrywanym słupie.

Przeanalizujmy to na przykładzie:

Dane: lista 2, 3, 4, 10, 2, 1

Obrót pętli	**0**	**1**	**2**	**3**	**4**	**5**
naj	0	7	10	18	18	18
pom	2	4	6	10	12	14

Najpierw **naj** ma wartość 0 (mniejszą od najmniejszego możliwego wyniku), a **pom** wartość 2 – pierwszego elementu listy. Rozpatrujemy drugi element listy. Dodajemy do wartości zmiennej **pom** odległość między słupami, czyli 2. Zmiennej **naj** przypisujemy 4 + 3 = 7, zmienna **pom** pozostaje bez zmiany (4 jest większe od 3, wysokości aktualnego słupa). Rozpatrujemy trzeci element listy: zmiennej **naj** przypisujemy 10, bo max(7, 6 + 4) = 10, zmienna **pom** ma wartość 6 (max(6,4)=6). W następnym kroku zmienna **naj** otrzymuje wartość 18 = max(10,8+10), a **pom** 10 = max(10,8). W przedostatnim kroku **naj** = max(18, 12 +2), **pom** = max (12, 2), w ostatnim kroku **naj** = max(18, 14+1).

Wynikiem funkcji jest wartość **naj** po analizie ostatniego elementu listy, w omawianym przykładzie liczba 18.

Rozwiązanie w języku Python

W omawianym rozwiązaniu rozpatrujemy listę począwszy od drugiego elementu, w języku Python zapisujemy to następująco: `lista[1:]`, czyli nazwa zmiennej oraz w nawiasie kwadratowym dwa indeksy oddzielone dwukropkiem: pierwszy wskazujący od jakiej wartości zaczynamy – 1 (listy numerujemy od 0), drugi wskazujący do jakiej wartości. Przy czym, jeśli elementy listy mają być brane do końca, to wartość indeksu można pominąć.

```
1.  def neon(lista):
2.      naj = 0
3.      pom = lista[0]
4.      for x in lista[1:]:
5.          pom = pom + 2
6.          naj = max(naj, pom + x)
7.          pom = max(pom, x)
8.      return naj
```

Warto też zwrócić uwagę, że w rozwiązaniu skorzystano z wbudowanej funkcji `max`, której wynikiem jest maksymalna wartość liczb podanych jako parametr.

Rozwiązanie w języku Logo

Ponieważ w pętli przeglądamy listę od drugiego elementu, odwołujemy się do elementu o indeksie npw+1 – numer powtórzenia +1.

```
1.  oto neon :lista
2.    niech "naj 0
3.    niech "pom pierw :lista
4.    powtórz (długość :lista) - 1
5.    [
6.      niech "x element npw+1 :lista
7.      (zwiększ "pom 2)
8.
9.      jeśli :pom + :x > :naj
10.       [niech "naj :pom + :x]
11.
12.     jeśli :x > :pom
13.       [niech "pom :x]
14.   ]
15.
16.   wy :naj
17. już
```

Testy

Testowanie rozwiązania zaczynamy od przypadków, których wynik można wyznaczyć w pamięci. Pierwsze dwa testy obejmują bardzo proste przykłady, kolejne dwa trochę trudniejsze. Przykład trzeci i czwarty bierze te same wartości jako podstawę optymalnego rozwiązania, ale w innym położeniu na liście. W piątym przykładzie jest testowana poprawność dla przypadków, które niewiele się różnią od siebie, by sprawdzić zdefiniowane warunki. Kolejne przykłady są dla losowych danych z wybranych przedziałów, a ostatni test dla wartości maksymalnych.

Dla języka Python

Wywołanie	Wynik
`neon([10,20])`	32
`neon([10,2,3,5,20])`	38
`neon([2,200,4,3,2,94,1])`	302
`neon([2,2,5,200,4,3,2,94,1])`	302
`neon([1,2,3,200,4,3,2,1,1,1,1,1,1,1,1,1,2,3,90,4,3,2])`	320
`neon([9001,9000,8999,8998,8997,8996,8995,8994,8993,8992,8991,8990,8989, 8988,8987,8986,8985,8984,8983,8982,8981,8980,8979,8978,8977,8976,8975, 8974,8973,8972,8971,8970,8969,8968,8967,8966,8965,8964,8963,8962,8961, 8960,8959,8958,8957,8956,8955,8954,8953,8952,8951,8950,8949,8948,8947, 8946,8945,8944,8943,8942,8941,8940,8939,8938,8937,8936,8935,8934,8933, 8932,8931,8930,8929,8928,8927,8926,8925,8924,8923,8922,8921,8920,8919, 8918,8917,8916,8915,8914,8913,8912,8911,8910,8909,8908,8907,8906,8905, 8904,8903,8902])`	18101
`neon([28,65,19,5,55,81,46,18,24,8,55,80,63,30,62,70,13,85,56,21])`	190
`neon([500,288,622,701,425,810,355,274,495,422,875,407,152,882,310,334, 482,541,276,805,858,870,120,706,716,639,565,212,916,716,241,622,382,315, 543,522,405,498,460,186,588,416,525,852,368,180,647,439,709,144])`	1828
`neon([9896,1183,6511,375,4949,9803,1836,3,6465,5925])`	19709
`neon([10000 for i in range(500)])`	20998

W ostatnim teście generowana jest lista długości 500, gdzie wartość każdego elementu wynosi 10000.

Dla języka Logo

Wywołanie	Wynik
`neon [10 20]`	32
`neon [10 2 3 5 20]`	38
`neon [2 200 4 3 2 94 1]`	302
`neon [2 2 5 200 4 3 2 94 1]`	302
`neon [1 2 3 200 4 3 2 1 1 1 1 1 1 1 1 2 3 90 4 3 2]`	320
`neon [9001 9000 8999 8998 8997 8996 8995 8994 8993 8992 8991 8990 8989 8988 8987 8986 8985 8984 8983 8982 8981 8980 8979 8978 8977 8976 8975 8974 8973 8972 8971 8970 8969 8968 8967 8966 8965 8964 8963 8962 8961 8960 8959 8958 8957 8956 8955 8954 8953 8952 8951 8950 8949 8948 8947 8946 8945 8944 8943 8942 8941 8940 8939 8938 8937 8936 8935 8934 8933 8932 8931 8930 8929 8928 8927 8926 8925 8924 8923 8922 8921 8920 8919 8918 8917 8916 8915 8914 8913 8912 8911 8910 8909 8908 8907 8906 8905 8904 8903 8902]`	18101
`neon [28 65 19 5 55 81 46 18 24 8 55 80 63 30 62 70 13 85 56 21]`	190
`neon [500 288 622 701 425 810 355 274 495 422 875 407 152 882 310 334 482 541 276 805 858 870 120 706 716 639 565 212 916 716 241 622 382 315 543 522 405 498 460 186 588 416 525 852 368 180 647 439 709 144]`	1828
`neon [9896 1183 6511 375 4949 9803 1836 3 6465 5925]`	19709
`neon (generuj 500 [10000] 10000)`	20998

W ostatnim teście generowana jest lista długości 500, gdzie wartość każdego elementu wynosi 10000.

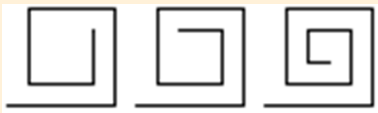
Zadanie Zawijasy – LOGIA 18 (2017/18), etap 2

Treść zadania

Tablica Polibiusza jest kwadratową tabelą zawierającą litery alfabetu łacińskiego. Kolumny numerujemy od 0 do 4, a wiersze od 1 do 5. Kodujemy słowo, zastępując każdą literę sumą numerów wiersza i kolumny, w których stoi. Na przykład litera **s** zostanie zastąpiona liczbą 6.

	0	1	2	3	4
1	a	b	c	d	e
2	f	g	h	i/j	k
3	l	m	n	o	p
4	q	r	s	t	u
5	v	w	x	y	z

Napisz jednoparametrową procedurę/funkcję **koduj**, której parametrem jest słowo składające się z co najmniej 2 i co najwyżej 18 małych liter alfabetu łacińskiego. Po jej wywołaniu na środku ekranu powstaje rysunek zakodowanego słowa. Każdą literę słowa zastępujemy liczbą, według zasady podanej powyżej, a następnie rysujemy zawijas stopnia takiego, jak liczba odpowiadająca zakodowanej literze. Pierwszy odcinek zawijasa jest poziomą kreską. Każdy kolejny odcinek zawijasa jest o 4 krótszy od poprzedniego. Odległości między kolejnymi zawijasami są równe 1/5 długości najdłuższego odcinka. Szerokość rysunku wynosi 780.



Rysunek pomocniczy – zawijasy stopnia 6, 7 i 9

efekt wywołania: w Pythonie – `koduj("fghjastuz")`, w Logo – koduj `"fghjastuz`

Omówienie rozwiązania

Warto zastanowić się chwilę, jak sprytnie zamienić każdą z liter alfabetu łacińskiego na liczbę będącą sumą numerów jej wiersza i kolumny. Warto skorzystać z operacji obliczania reszty z dzielenia oraz ilorazu całkowitego.

Po zamianie liter na liczby, możemy przystąpić do rysowania zawijasów.

Z treści zadania wynika, że szerokość całego rysunku jest stała, niezależna od liczby kodowanych liter. Dlatego musimy obliczyć szerokość pojedynczego zawijasa. Należy wziąć pod uwagę fakt, że zawijasów jest tyle co liter, a odstępów pomiędzy nimi mniej o 1.

Zgodnie z treścią zadania rysunek ma być wyśrodkowany, dlatego ważna jest wysokość rysunku. Wszystkie litery, oprócz **a**, mają wysokość równą szerokości zawijasa pomniejszonej o 4, bo tyle wynosi drugi odcinek zawijasa. W przypadku litery **a**, zawijas składa się z tylko jednego odcinka poziomego, wtedy jego wysokość wynosi 0 i trzeba to uwzględnić przy wyśrodkowaniu.

Rozwiązanie w języku Python

```python
from turtle import *

def liczba(litera):
    alfabet = "abcdefghjklmnopqrstuvwxyz"
    if (litera == "i"):
        return 5
    return alfabet.index(litera) % 5 + alfabet.index(litera) // 5 + 1

def zawijas(a, ile):
    for i in range(ile):
        fd(a); lt(90)
        a -= 4
    for i in range(ile):
        a += 4
        rt(90); bk(a)

def czysamoa(wyraz):
    for i in wyraz:
        if(i != "a"):
            return False
    return True

def koduj(wyraz):
    # szerokość zawijasa
    a = 780 / (1.2 * len(wyraz)- 0.2)
    pu(); bk(390); pd()
    if not czysamoa(wyraz):
        pu(); lt(90); bk((a - 4)/2); rt(90); pd()
    for i in range(len(wyraz)):
        zawijas(a, liczba(wyraz[i]))
        pu(); fd(1.2 * a); pd()
```

Rozwiązanie w języku Logo

```
oto liczba :litera
 niech "alfabet "abcdefghjklmnopqrstuvwxyz
 jeśli (:litera = "i) [wy 5]
 niech "nr (numel :litera :alfabet) - 1
 wy (reszta :nr 5) + (ilorazc :nr 5) + 1
już

oto zawijas :a :ile
 powtórz :ile [np :a lw 90 (zmniejsz "a 4)]
 powtórz :ile [(zwiększ "a 4) pw 90 ws :a]
już

oto czysamoa :wyraz
 powtórz długość :wyraz [jeśli (element npw :wyraz) <> "a [wy "fałsz]]
 wy "prawda
już

oto koduj :wyraz
 ; szerokość zawijasa
 niech "a 780 / (1.2 * (długość :wyraz) - 0.2)
 pw 90
 pod ws 390 opu
 jeśli nie czysamoa :wyraz[pod lw 90 ws (:a - 4) / 2 pw 90 opu]
 powtórz długość :wyraz [zawijas :a liczba element npw :wyraz
                         pod np 1.2 * :a opu]
już
```

Wszystkie zawijasy mają pierwszy odcinek poziomy, dlatego warto w Logo na samym początku procedury głównej żółwia obrócić w prawo o 90^0.

Podczas rozwiązywania zadań, powinno się testować procedury/funkcje pomocnicze. Jak zrobić to w jednej linijce dla wszystkich możliwych wartości parametru? Na przykład, aby sprawdzić, czy funkcja liczba daje prawidłowe wyniki dla wszystkich liter alfabetu łacińskiego w Pythonie można napisać taką pętlę:

```
for i in range(26):
    print(liczba(chr(97 + i)))
```

To samo w Logo będzie to wyglądało następująco:

```
powtórz 26 [pokaż liczba znak 96 + npw]
```

Kolejny test, który warto wykonać, to sprawdzenie, czy szerokość rysunku jest prawidłowa.

Testy

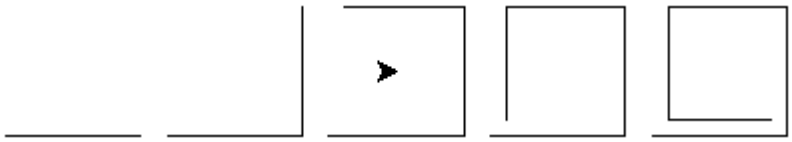
Testowanie rozwiązania powinno obejmować wszystkie litery i przypadki szczególne.

Python

```
koduj("abcde")
```

Logo

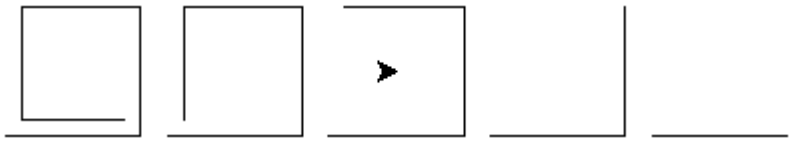
```
koduj "abcde
```



Uwaga! Na zrzucie żółw stoi na środku ekranu. Sprawdzamy w ten sposób wyśrodkowanie rysunku.
Dane z pierwszego wiersza tablicy Polibiusza.

```
koduj("vqlfa")
```

```
koduj "vqlfa
```



Dane z pierwszej kolumny tablicy Polibiusza

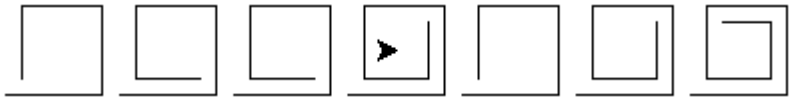
```
koduj("ngatz")
```

```
koduj "ngatz
```

Dane z przekątnej tablicy Polibiusza

```
koduj("hijkmop")
```

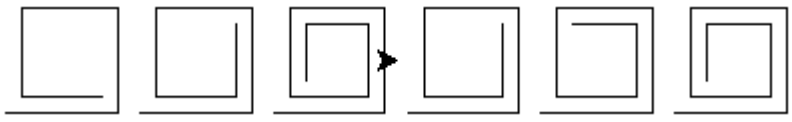
```
koduj "hijkmop
```



*Zwracamy uwagę, czy kody liter **i** oraz **j** są identyczne.*
Pozostałe dane z tablicy Polibiusza

```
koduj("rsuwxy")
```

```
koduj "rsuwxy
```



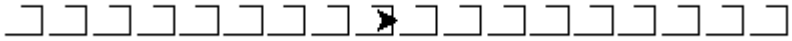
Pozostałe dane z tablicy Polibiusza

```
koduj("aaaaaaa")
```

```
koduj "aaaaaaa
```

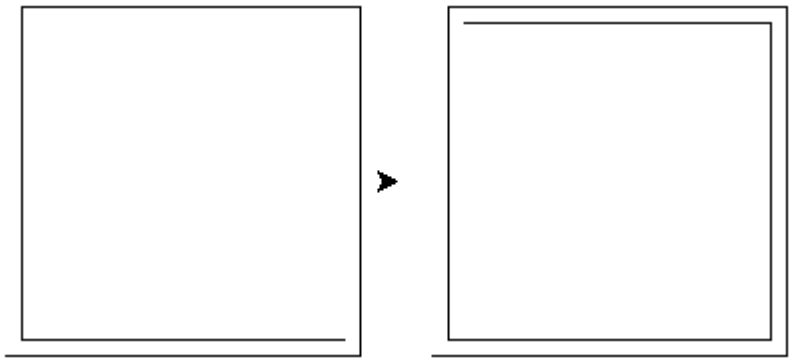
*Szczególny przypadek – wszystkie litery **a**.*

```
koduj("cccccccccccccccccc")        koduj "cccccccccccccccccc
```



Szczególny przypadek – wszystkie jednakowe litery, maksymalna długość danych.

```
koduj("rp")                              koduj "rp
```



Szczególny przypadek – tylko 2 litery.