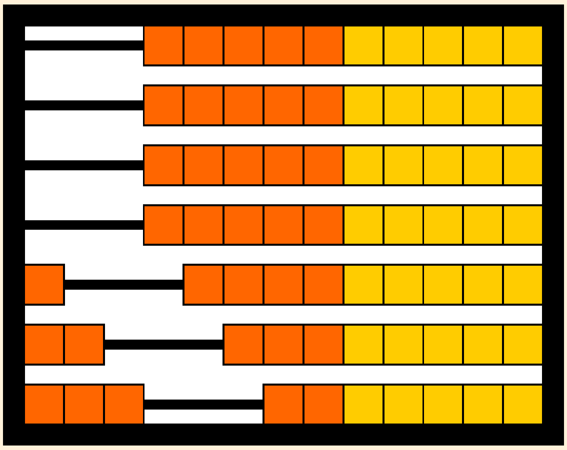
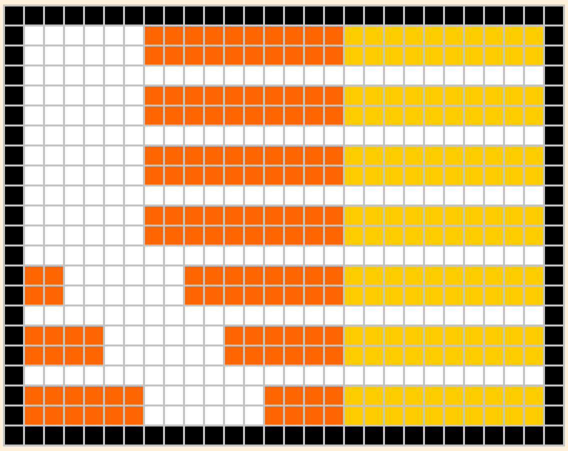
Zadanie Liczydło – LOGIA 24 (2023/24), etap 2

Treść zadania

Liczydło to przyrząd służący do wykonywania prostych działań arytmetycznych, zbudowany z kolorowych ruchomych koralików umieszczonych na prętach. Koraliki są w dwóch kolorach pogrupowane po 5, tak jak na rysunkach przykładowych. Każdy poziom odpowiada określonej potędze liczby 10. Dolny poziom reprezentuje jedności (10^0), drugi od dołu dziesiątki (10^1), kolejny setki (10^2), itd. Liczbę można przedstawić na liczydle przesuwając wybrane koraliki z prawej strony na lewą, maksymalnie 9 na jednym poziomie. Układ na rysunku poniżej reprezentuje liczbę 123 na liczydle złożonym z 7 prętów.

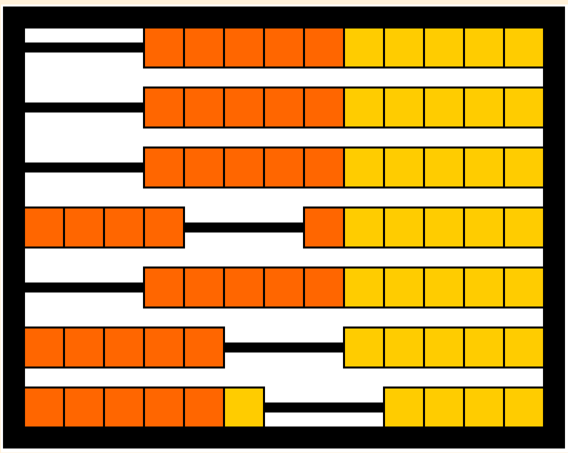




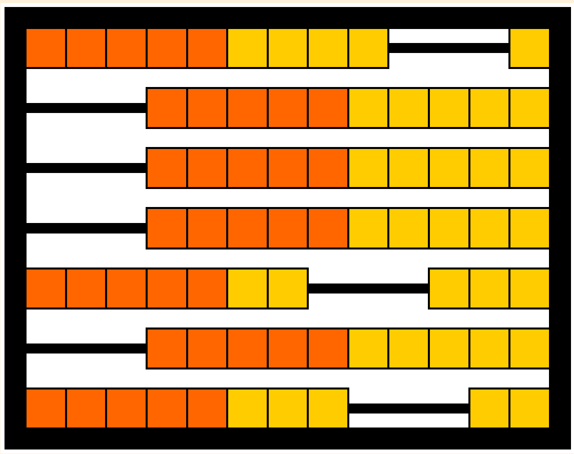
rysunek pomocniczy

Napisz funkcję **ustaw(n)**, po wywołaniu której powstanie na środku ekranu rysunek liczydła złożonego z 7 prętów z ustawionymi koralikami reprezentującymi parametr **n** z zakresu od 0 do 9 999 999. Długość boku koralika wynosi 50, a grubość pręta 10. Pozostałe wymiary odczytaj z rysunku pomocniczego.

Przykłady:



ustaw(4056)



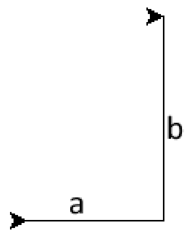
ustaw(9000708)

Omówienie rozwiązania

Podstawowymi elementami liczydła są kolorowe prostokąty, dlatego w rozwiązaniu będzie wykorzystana funkcja pomocnicza **prost** z trzema parametrami – **szerokość** i **wysokość prostokąta** oraz **kolor zamalowania**. Rysunek prostokąta będzie tworzony z dolnego lewego narożnika.

```
def prostokat(x, y, kolor):
    fillcolor(kolor)
    begin_fill()
    for i in range(2):
        fd(x); lt(90); fd(y); lt(90)
    end_fill()
```

Kolejną funkcję pomocniczą **skok** z dwoma parametrami **a** i **b** warto wykorzystać do przemieszczania żółwia z podniesionym pisakiem po ekranie.




```
def skok(a, b):
    pu()
    lt(90); fd(b); rt(90); fd(a)
    pd()
```

Cały rysunek można podzielić na trzy etapy. Rysowanie:

- ramki,
- prętów,
- koralików.

Przyjmijmy za jednostkę długość boku koralika. Ramka składa się z czarnego prostokąta o szerokości 14***bok** i wysokości 11***bok** oraz nałożonego na niego białego prostokąta o szerokości 13***bok** i wysokości 10***bok**. Prostokąty są współśrodkowe.

```
def ramka(bok):
    prostokat(14 * bok, 11 * bok, "black")
    skok(0.5 * bok, 0.5 * bok)
    prostokat(13 * bok, 10 * bok, "white")
```

Pręty są koloru czarnego. Szerokość prętów wynosi 13***bok** a ich wysokość jest równa 1/5 długości boku. Odległości między prętami to 3/2 długości boku.

```
def prety(bok):
    for i in range(7):
        prostokat(13 * bok, bok / 5, "black")
        skok(0, 1.5 * bok)
```

Ostatni etap odnosi się do kluczowej części zadania – rysowania koralików. Zrobimy to dla pojedynczej cyfry. Należy zauważyć, że zawsze mamy do narysowania 10 koralików. Pierwsze pięć ma kolor pomarańczowy (reprezentacja w kodzie szesnastkowym `'#ff6600'`), kolejne pięć kolor żółty (`'#ffcc00'`). Po lewej stronie liczydła rysujemy tyle koralików, jaka jest wartość parametru **cyfra**. Następnie robimy odstęp szerokości trzech koralików i rysujemy pozostałe koraliki. Możemy przy tym posłużyć się regułą:

```
jeżeli masz do narysowania zero koralików po lewej stronie liczydła to
  przesuń się o trzy długości boku koralika w prawo
narysuj koralik i przesuń się o długość boku koralika w prawo
zmniejsz liczbę koralików do narysowania o jeden
```

```
def koraliki(cyfra, bok):
    kolor = "#ff6600"
    po_lewej = cyfra
    for j in range(10):
        if j > 4:
            kolor = "#ffcc00"
        if po_lewej == 0:
            skok(3 * bok, 0)
        prostokat(bok, bok, kolor)
        skok(bok, 0)
        po_lewej -= 1
```

Cały układ składa się z siedmiu serii koralików. Idąc od dołu liczydła, serie koralików odpowiadają poszczególnym cyfrom liczby od prawej do lewej. Najniższa seria odpowiada ostatniej cyfrze, druga przedostatniej itd. Ostatnią cyfrę danej liczby wyznaczamy obliczając jej resztę z dzielenia przez 10. Po narysowaniu układu dla danej cyfry liczbę dzielimy całkowicie przez 10. Odcinamy w ten sposób rozpatrzoną cyfrę. Jeśli liczba ma mniej cyfr niż prętów, to na górnych prętach są reprezentacje cyfry 0.

Funkcję **ustaw()** należy uzupełnić o skoki tak, aby rysunek był na środku ekranu oraz poszczególne elementy znajdowały się we właściwych miejscach.

```
def ustaw(n):
    bok = 50
    #ramka
    skok(-7 * bok, -5.5 * bok)
    ramka(bok)
    #pręty
    skok(0, 2 * bok / 5)
    prety(bok)
    #korale
    skok(0, -7 * 1.5 * bok - 2 * bok / 5)
    for i in range(7):
        cyfra = n % 10
        n = n // 10
        koraliki(cyfra, bok)
        skok(-13 * bok, 1.5 * bok)
```

Rozwiązanie w języku Python

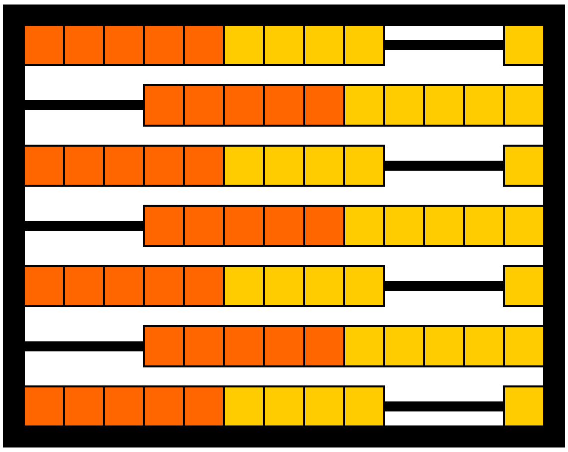
```
from turtle import *
def prostokat(x, y, kolor):
    fillcolor(kolor)
    begin_fill()
    for i in range(2):
        fd(x); lt(90); fd(y); lt(90)
    end_fill()

def skok(a, b):
    pu()
    lt(90); fd(b); rt(90); fd(a)
    pd()

def ramka(bok):
    prostokat(14 * bok, 11 * bok, "black")
    skok(0.5 * bok, 0.5 * bok)
    prostokat(13 * bok, 10 * bok, "white")

def prety(bok):
    for i in range(7):
        prostokat(13 * bok, bok / 5, "black")
        skok(0, 1.5 * bok)

def koraliki(cyfra, bok):
    kolor = "#ff6600"
    po_lewej = cyfra
    for j in range(10):
        if j > 4:
            kolor = "#ffcc00"
        if po_lewej == 0:
            skok(3 * bok, 0)
        prostokat(bok, bok, kolor)
        skok(bok, 0)
        po_lewej -= 1

def ustaw(n):
    bok = 50
    #ramka
    skok(-7 * bok, -5.5 * bok)
    ramka(bok)
    #pręty
    skok(0, 2 * bok / 5)
    prety(bok)
    #korale
    skok(0, -7 * 1.5 * bok - 2 * bok / 5)
    for i in range(7):
        cyfra = n % 10
        n = n // 10
        koraliki(cyfra, bok)
        skok(-13 * bok, 1.5 * bok)
```

Testy

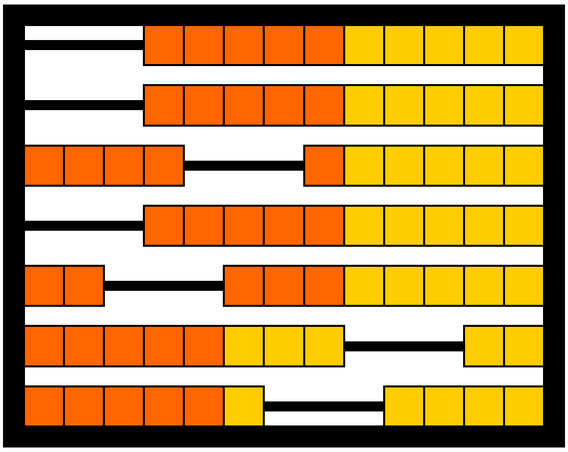
Wywołujemy funkcję **ustaw()** dla różnych parametrów. Liczby wytypowane do testów powinny zawierać wszystkie cyfry od 0 do 9 oraz składać się od 1 do 7 cyfr. Należy uwzględnić liczbę 0 oraz liczbę kończącą się cyfrą 0 (np. *7953100*).

W języku Python, aby przyspieszyć tworzenie rysunku przez żółwia, stosujemy wywołanie złożone z funkcji `tracer()` – rysownie w pamięci, właściwego wywołania funkcji **ustaw()** i na końcu uaktualniamy ekran za pomocą funkcji `update()`.

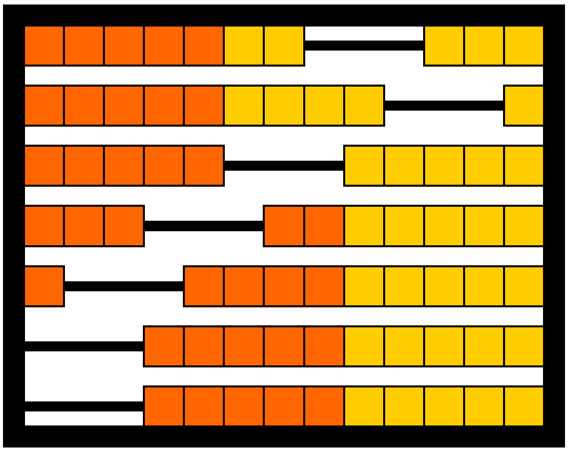
```
tracer(0); ustaw(); update()
```



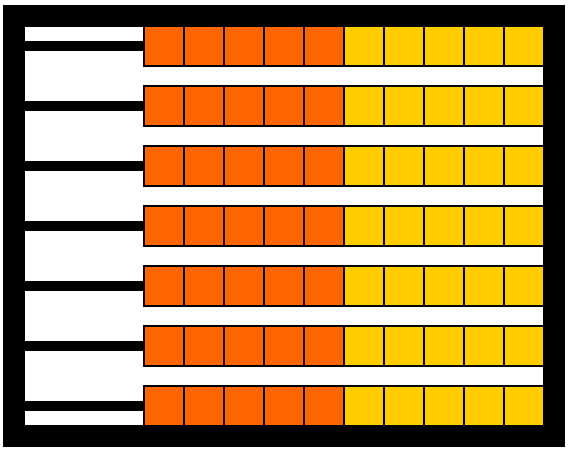
ustaw(9090909)



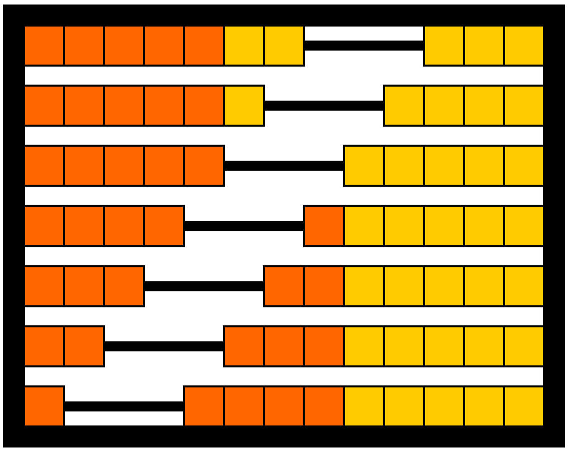
ustaw(40286)



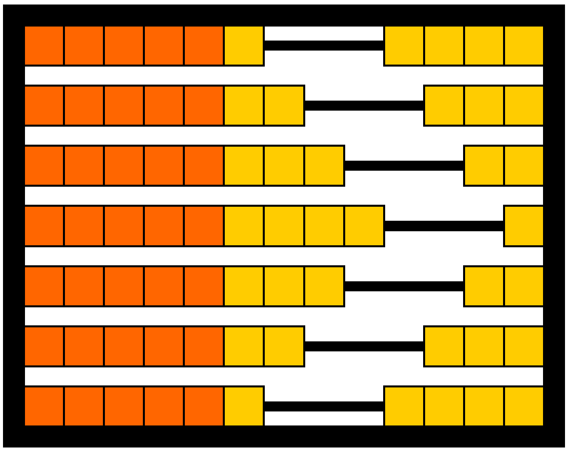
ustaw(7953100)



ustaw(0)



ustaw(7654321)



ustaw(6789876)

Zadanie Kolorowy zegar – LOGIA 24 (2023/24), etap 2

Treść zadania

Jola korzysta z cyfrowego zegara, na którym czas (godzina i minuty) wyświetlany jest na kolorowych planszach. Każda z 24 plansz określających godziny ma jeden z czterech kolorów:

- czerwony (c) – godziny 0, 4, 8, 12, 16, 20;
- zielony (z) – godziny 1, 5, 9, itd.;
- niebieski (n) – godziny 2, 6, 10, itd.;
- fioletowy (f) – godziny 3, 7, 11, itd.

Plansze z minutami mają 5 cyklicznie powtarzających się kolorów: 0 – z, 1 – n, 2 – f, 3 – c, 4 – p (pomarańczowy).

Napisz program, który wczyta dwie liczby określające czas (godzinę i minuty), a następnie policzy i wypisze minimalną liczbę minut, jakie muszą upłynąć do chwili, gdy obie plansze na zegarze będą miały ten sam kolor.

Wejście

Dwie liczby całkowite nieujemne *g* i *m* oddzielone spacją, $0 \le g \le 23$, $0 \le m \le 59$.

Wyjście

Liczba całkowita nieujemna.

Przykłady:

Wejście	**6 33**	**7 36**	**22 58**
Wyjście	**3**	**1**	**4**
	6 – niebieska, 33 – czerwona, za 3 minuty będą dwie plansze niebieskie	7 – fioletowa, 36 – niebieska, za 1 minutę obie będą fioletowe	22 – niebieska, 58 – czerwona, za 4 minuty obie będą fioletowe

Omówienie rozwiązania

Rozwiązanie zadania najlepiej konstruować wykorzystując arytmetykę modularną. Istotna własność – okresowość operacji reszta z dzielenia przy stałym dzielniku znacznie uprości rozwiązanie.

Weźmy sekwencje pierwszych liter pojawiających się kolorów oddzielnie dla godzin – „cznf” i minut – „znfcp”. Do sprawdzenia jaki mamy kolor karteczki dla danej godziny obliczamy resztę z dzielenia `g` przez 4 i sprawdzamy numer litery w napisie „cznf”. Podobnie w przypadku minut, lecz dzielenie z resztą jest przez 5, a literę sprawdzamy w napisie „znfcp”. Numerowanie zaczynamy od zera. W zmiennej `odp` będziemy zliczać czas do momentu pojawienia się dwóch jednokolorowych karteczek.

Zwiększanie wartości czasu `g` i `m` o jedną minutę może być przeprowadzone na kilka sposobów. Można przykładowo po dodaniu minuty i obliczeniu reszty z dzielenia przez 60 sprawdzić, czy otrzymaliśmy wartość 0. W tym przypadku zwiększamy wskazania godziny o 1. Nie przejmujemy się zmienną `g` z wartością 24 zamiast 0, bo interesuje nas w zasadzie jej wartość dzielenia przez 4, która w obu przypadkach jest ta sama.

Pseudokod rozwiązania:

```
wczytaj g, m
godz ← "cznf"
min ← "znfcp"
odp ← 0
dopóki godz[g mod 4] <> min[m mod 5] wykonuj
    odp = odp + 1
    m ← (m + 1) mod 60
    jeżeli m = 0 to g ← g + 1
wypisz odp
```

Rozwiązanie w języku Python

```python
g, m = input().split()
g, m = int(g), int(m)

godz = "cznf"
min = "znfcp"
odp = 0

while godz[g % 4] != min[m % 5]:
    odp = odp + 1
    m = (m + 1) % 60
    if m == 0: g += 1

print(odp)
```

Testy

Najpierw należy przetestować zadanie na przykładach z treści zadania. Następnie dla różnych testów, uwzględniając zmianę godziny oraz przejście przez północ. Możliwe wyniki to wartości od 0 do 5, więc należy tak dobrać testy, aby uwzględnić wszystkie możliwe wyniki.

Test	Wynik
20 0	3
14 0	1
7 5	2
17 16	4
18 22	4
23 58	5
0 59	1
9 15	0
21 56	5
11 52	0

Zadanie Naj… – LOGIA 24 (2023/24), etap 2

Treść zadania

Adam analizuje napisy złożone z małych liter alfabetu polskiego (32 litery):

a	ą	b	c	ć	d	e	ę	f	g	h	i	j	k	l	ł	m	n	ń	o	ó	p	r	s	ś	t	u	w	y	z	ź	ż

Z każdego napisu wybiera literę najwcześniejszą w alfabecie i najpóźniejszą.

Pomóż Adamowi i napisz program, który wczyta napis, znajdzie i wypisze najwcześniejszą oraz najpóźniejszą literę występujące w napisie, a także odległość między nimi w alfabecie.

Wejście

Niepusty napis złożony z małych liter alfabetu polskiego o długości nie większej niż 1000 znaków.

Wyjście

Dwie litery i liczba całkowita nieujemna oddzielone spacjami.

Przykłady:

Wejście	**abrakadabra**	**żółw**	**misiek**
Wyjście	**a r 22**	**ł ż 16**	**e s 17**

Omówienie rozwiązania

Zadanie polega na analizie napisu składającego się z małych liter alfabetu polskiego. Z napisu należy wybrać literę najwcześniejszą i najpóźniejszą w alfabecie, a następnie obliczyć odległość w alfabecie między nimi. Program powinien wczytać napis i wypisać trzy dane – dwie litery, najwcześniejszą oraz najpóźniejszą występującą w napisie oraz liczbę – odległość między nimi w alfabecie.

Ponieważ alfabet zawiera również polskie znaki diakrytyczne, nie można skorzystać z kodów ASCII do przyporządkowania literom alfabetu kolejnych liczb, należy wybrać inne rozwiązanie. Gdy utworzymy napis, który zawiera litery w kolejności alfabetycznej, to pozycja litery w napisie będzie pozycją litery w alfabecie. W ten sposób możemy wczytany napis zamienić na listę liczb. Z listy wybieramy liczbę najmniejszą i największą, a następnie znajdujemy odpowiadające im litery. Ponadto liczymy odległość jako różnicę liczby największej i najmniejszej.

Rozwiązanie w języku Python

Rozwiązanie korzysta z mechanizmu list składanych (ang. *list comprehensions*), który umożliwia szybkie i efektywne tworzenie nowych list poprzez przekształcenie elementów danej sekwencji. W tym przypadku, konstrukcja listy składanej `[alfabet.index(x) for x in napis]` jest używana do iteracji po znakach napisu. Dla każdego znaku `x` w zmiennej `napis`, wykonywane jest wyrażenie `alfabet.index(x)`, które szuka pozycji tego znaku w alfabecie. Działanie to przekształca każdy znak napisu na odpowiadający mu indeks w zmiennej `alfabet`, co pozwala na reprezentację napisu jako listy liczb. Ta lista indeksów jest następnie wykorzystywana do dalszych operacji, takich jak znalezienie minimalnej i maksymalnej wartości, co odpowiada odpowiednio najwcześniejszej i najpóźniejszej literze w alfabecie spośród liter występujących w napisie. Do znalezienia wartości minimalnej i maksymalnej wykorzystujemy funkcje `min()` i `max()`.

```python
def znajdz(napis):
    alfabet = "aąbcćdeęfghijklłmnńoóprsśtuwyzźż"

    lista = [alfabet.index(x) for x in napis]
    p = min(lista)
    k = max(lista)

    return alfabet[p], alfabet[k], k - p

napis = input()
wynik = znajdz(napis)
print(wynik[0], wynik[1], wynik[2])
```

Testy

Najpierw testujemy zadanie na przykładach z treści zadania. Potem warto sprawdzić proste, krótkie słowa, które nie zawierają polskich znaków diakrytycznych. W testach nie powinno zabraknąć sprawdzenia, jak algorytm radzi sobie z dłuższym napisem i czy poprawnie identyfikuje skrajne litery zarówno z alfabetu łacińskiego jak i polskiego. Ważnym testem jest również napis zawierający wszystkie litery alfabetu.

Test	Wynik
kij	i k 2
glif	f l 6
logia	a o 19
algorytmika	a y 28
www	w w 0
"mn" * 100	m n 1
wół	ł w 12
nietoperz	e z 23
czwórki	c z 26
ćąlwaęócktosjhińbpugśfżźrzełmynd	a ż 31

Zadanie Liczby pierwsze Germain – LOGIA 24 (2023/24), etap 2

Treść zadania

Janek interesuje się kryptografią. Dowiedział się, że liczby pierwsze są wykorzystywane w kryptografii, a szczególne znaczenie mają liczby pierwsze Germain, które zostały zdefiniowane przez francuską matematyczkę Sophie Germain. Liczba pierwsza *p* jest liczbą Germain, jeśli liczba 2 * *p* + 1 też jest pierwsza.

Na przykład 11 jest liczbą pierwszą Germain, ponieważ jest pierwsza i liczba 2 * 11 + 1 = 23 też jest pierwsza, a 13 nie jest liczbą pierwszą Germain, bo 2 * 13 + 1 = 27 nie jest liczbą pierwszą. Janek zastanawia się, jak dużo jest liczb pierwszych Germain.

Pomóż mu i napisz program, który wczyta dwie liczby naturalne *a* i *b* oraz policzy, ile jest liczb pierwszych Germain w przedziale [*a*, *b*]. Postaraj się tak napisać program, żeby Janek nie czekał długo na wynik.

Wejście

Dwie liczby naturalne *a* i *b* oddzielone spacją, 1 < *a*, *a* < *b*, *b* < 5000000.

Wyjście

Liczba naturalna określająca, ile jest liczb pierwszych Germain w przedziale [*a*, *b*].

Przykłady:

Wejście	**2 10**	**6 13**	**10 1000**
Wyjście	**3**	**1**	**34**
	Liczby 2, 3 i 5 są liczbami pierwszymi Germain, a 7 nie jest.	Liczba 11 jest liczbą pierwszą Germain, a 7 i 13 nie są.	

Omówienie rozwiązania

Zostaną omówione trzy rozwiązania zadania o różnej złożoności obliczeniowej, ponieważ w treści zadania znajduje się sformułowanie „Postaraj się tak napisać program, aby Janek nie czekał długo na wynik”. Dwa pierwsze będą wykorzystywały algorytm sprawdzania, czy dana liczba jest liczbą pierwszą, trzecia wykorzysta metodę sita Eratostenesa do wygenerowania zbioru liczb pierwszych.

Algorytm 1

Żeby sprawdzić, czy dana liczba naturalna *n* jest liczbą pierwszą, będziemy poszukiwać jakiegoś dzielnika większego od 1 i mniejszego od *n*. Spróbujemy wykazać, że liczba jest złożona. Jeśli się to nie uda, liczba *n* jest liczbą pierwszą. Sprawdzanie wszystkich dzielników od 2 do *n*-1 nie ma większego sensu, łatwo zauważyć, że ewentualny dzielnik nie może być większy od połowy *n*. Można także sprawdzić oddzielnie parzystość liczby (2 jest jedyną liczbą pierwszą parzystą) i pętlę poszukującą dzielnika wykonywać z krokiem 2 (sprawdzać tylko potencjalne dzielniki nieparzyste).

```
funkcja pierwsza(n)
    jeżeli n=2 to zwróć prawda i zakończ
    jeżeli n jest parzyste to zwróć fałsz i zakończ
    dla d od 3 do n/2 z krokiem 2 wykonuj
        jeżeli d jest dzielnikiem n to zwróć fałsz i zakończ
    zwróć prawda i zakończ
```

Żeby policzyć ile jest liczb pierwszych Germain w przedziale [*a*, *b*] wystarczy rozpatrzyć wszystkie liczby *x* w tym przedziale i policzyć te, dla których zachodzi warunek `pierwsza(x)` oraz `pierwsza(2*x+1)`.

```
ile ← 0
dla x od a do b wykonuj
    jeżeli pierwsza(x) oraz pierwsza(2*x+1) to ile ← ile + 1
```

Można też sprawdzić poza pętlą, czy liczba 2 (która jest liczbą pierwszą Germain) należy do przedziału [*a*, *b*], a następnie wykonywać pętlę z krokiem 2 tylko dla liczb nieparzystych. Nie zmienia to jednak złożoności obliczeniowej algorytmu.

Algorytm 2

Można zdecydowanie efektywniej sprawdzać, czy liczba jest liczbą pierwszą. Nie trzeba poszukiwać dzielnika w zakresie do połowy liczby, wystarczy do pierwiastka. Jeżeli liczba n jest złożona, to znaczy, że można ją przedstawić w postaci iloczynu $n = d1 * d2$, gdzie $d1$ i $d2$ są różne od 1 i n. Jedna z wartości $d1$, $d2$ musi być mniejsza równa od pierwiastka z n, druga większa równa. Gdyby obydwie były większe, to iloczyn byłby większy od n. Wystarczy, że spróbujemy znaleźć ten mniejszy dzielnik.

```
funkcja pierwsza(n)
    jeżeli n=2 to zwróć prawda i zakończ
    jeżeli n jest parzyste to zwróć fałsz i zakończ
    d ← 3
    dopóki d * d ≤ n wykonuj
        jeżeli d jest dzielnikiem n to zwróć fałsz i zakończ
        w przeciwnym przypadku d ← d + 2
    zwróć prawda i zakończ
```

Warunek $d \leq \sqrt{n}$ został przekształcony do postaci $d * d \leq n$, żeby nie prowadzać obliczeń na liczbach rzeczywistych.

Warto porównać, ile prób dzielenia wykonają dwie wersje funkcji `pierwsza`. Na przykład dla liczby pierwszej rzędu 10000 pierwsza z nich wykona ok. 2500 prób dzielenia, druga tylko ok. 50. Warto wiedzieć, że liczbę prób dzielenia można jeszcze o 1/3 ograniczyć (wykonując dwa sprawdzenia dzielenia w pętli, którą wykonujemy z krokiem 6), zapis tego algorytmu pomijamy.

Sposób zliczania liczb pierwszych Germain jest taki sam jak w poprzednim algorytmie, zmieniła się tylko funkcja `pierwsza`.

Algorytm 3

Efektywniejszym rozwiązaniem jest zastosowanie algorytmu sita Eratostenesa. Algorytm polega na wykreślaniu liczb złożonych (czyli ustawianiu dla nich wartości `fałsz`), będących wielokrotnościami kolejnych liczb pierwszych. Unikamy w ten sposób wielokrotnego badania podzielności.

```
funkcja sito(n)
    pierwsze ← lista wartości logicznych indeksowana od 0 do n
    # wartości początkowe listy to prawda dla indeksów nieparzystych,
    # fałsz dla parzystych, dla 1 fałsz, dla 2 prawda
    d ← 3
    dopóki d * d ≤ n wykonuj
        jeżeli pierwsze[d] to
            dla i od d*d do n z krokiem d wykonuj pierwsze[i] ← fałsz
        d ← d + 2
    zwróć pierwsze i zakończ
```

Należy pamiętać, że dla przedziału [*a*, *b*] górny zakres listy powinien wynosić 2**b*+1, a więc funkcję `sito` należy wywołać z takim parametrem. W algorytmie liczącym liczby pierwsze Germain zamiast wywołania funkcji `pierwsza` należy odwołać się do wartości pamiętanych na liście.

```
pierwsze ← sito(2*b+1)
ile ← 0
dla x od a do b wykonuj
    jeżeli pierwsze[x] oraz pierwsze[2*x+1] to ile ← ile + 1
```

Rozwiązanie w języku Python

Algorytm 1

```
def pierwsza(n):
    if n == 2: return True
    if n % 2 == 0: return False
    for d in range(3, n // 2 + 1, 2):
        if n % d == 0: return False
    return True

a, b = input().split()
a = int(a)
b = int(b)
ile = 0
for i in range(a, b + 1):
    if pierwsza(i) and pierwsza(2 * i + 1): ile += 1
print(ile)
```

Algorytm 2

Program różni się jedynie definicją funkcji `pierwsza`.

```
def pierwsza(n):
    if n == 2: return True
    if n % 2 == 0: return False
    d = 3
    while d * d <= n:
        if n % d == 0: return False
        else: d += 2
    return True
```

Algorytm 3

Definicja funkcji `sito` korzysta z mechanizmu list składanych do utworzenia listy `pierwsze` oraz nadania jej wartości początkowych. W tym przypadku, konstrukcja listy składanej `[i%2==1 for i in range(n+1)]` określa wartości `True` dla liczb nieparzystych, a wartości `False` dla liczb parzystych. Potem następuje korekta dla liczb 1 i 2.

```
def sito(n):
    pierwsze = [i % 2 == 1 for i in range(n + 1)]
    pierwsze[1] = False
    pierwsze[2] = True
    d=3
    while d * d <= n:
        if pierwsze[d]:
            for i in range(d * d, n + 1, d): pierwsze[i] = False
        d += 2
    return pierwsze

a, b = input().split()
a = int(a)
b = int(b)
```

```
pierwsze = sito(2 * b + 1)
ile = 0
for i in range(a, b + 1):
    if pierwsze[i] and pierwsze[2 * i + 1]: ile += 1
print(ile)
```

Testy

Najpierw należy przetestować zadanie na przykładach z treści zadania. Kolejne testy należy dobrać tak, aby sprawdzały zarówno poprawność i złożoność algorytmu, w szczególności sprawdzić działanie programu dla przedziału, gdy liczby pierwsze Germain są końcami przedziału. Podczas konkursu zadanie było testowane na następujących grupach testów, kolejne grupy zawierały coraz większe przedziały danych.

Grupa testów	Test	Wynik
I	20 100 9 75	6 5
II	2 89 7 97	10 7
III	9973 10007 2 50000	0 670
IV	997 499979 100 1000000	4287 7736
V	97 4000037 1000 4999999	25298 30620

Rozwiązania o złożoności obliczeniowej zgodnej z algorytmem 1 (liniowe sprawdzanie pierwszości liczby) uzyskiwały do 60 % możliwych punktów (3 pierwsze grupy testów). Rozwiązania o złożoności obliczeniowej zgodnej z algorytmem 2 (sprawdzanie pierwszości liczby do pierwiastka) uzyskiwały do 80 % możliwych punktów (4 pierwsze grupy testów).