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 $d \le \sqrt{n}$ został przekształcony do postaci *d * d ≤ 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

```
 1  def pierwsza(n):
 2      if n == 2: return True
 3      if n % 2 == 0: return False
 4      for d in range(3, n // 2 + 1, 2):
 5          if n % d == 0: return False
 6      return True
 7
 8  a, b = input().split()
 9  a = int(a)
10  b = int(b)
11  ile = 0
12  for i in range(a, b + 1):
13      if pierwsza(i) and pierwsza(2 * i + 1): ile += 1
14  print(ile)
```

Algorytm 2

Program różni się jedynie definicją funkcji `pierwsza`.

```
 1  def pierwsza(n):
 2      if n == 2: return True
 3      if n % 2 == 0: return False
 4      d = 3
 5      while d * d <= n:
 6          if n % d == 0: return False
 7          else: d += 2
 8      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.

```
 1  def sito(n):
 2      pierwsze = [i % 2 == 1 for i in range(n + 1)]
 3      pierwsze[1] = False
 4      pierwsze[2] = True
 5      d=3
 6      while d * d <= n:
 7          if pierwsze[d]:
 8              for i in range(d * d, n + 1, d): pierwsze[i] = False
 9          d += 2
10      return pierwsze
11
12  a, b = input().split()
13  a = int(a)
14  b = int(b)
```

```
15 pierwsze = sito(2 * b + 1)
16 ile = 0
17 for i in range(a, b + 1):
18     if pierwsze[i] and pierwsze[2 * i + 1]: ile += 1
19 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).