Test Pierwszości Millera-Rabina

Damian Wojtyczko

1 Test Pierwszości Millera-Rabina

1.1 Wstęp

Test Millera-Rabina to probabilistyczny test pierwszości, który sprawdza czy dana liczba jest liczbą pierwszą. Algorytm w wersji deterministycznej początkowo został opracowany przez Garry'ego Millera w 1975 roku, jednak jego poprawność opierała się na założeniu prawdziwości Uogólnionej hipotezy Riemanna. Kilka lat później, Michael Oser Rabin przekształcił ten algorytm do postaci probablistycznej i udowodnił jego niezależną od prawdziwości Uogólnionej hipotezy Riemanna poprawność.

1.2 Skrócony opis algorytmu

Test Millera-Rabina na wejściu przyjmuje nieparzystą liczbę naturalną n, której pierwszość będziemy sprawdzać oraz parametr k, od którego uzależniona jest dokładność testu (im większe k, tym mniejsze prawdopodobieństwo błędu). Parametr ten bezpośrednio definiuje liczbę powtórzeń Testu Millera-Rabina. Ponadto, obliczana jest największa możliwa liczba s oraz współczynnik d, takie, że:

$$n - 1 = 2^s \cdot d$$

Następnie w pętli (k razy) wykonywane są następujące działania:

- Wylosuj liczbę $a \in \{2, 3, ..., n-2\}$
- Jeśli $a^d \equiv \pm 1 \ (mod(n))$ zwróć *True* (n prawdopodobnie pierwsze)
- Jeśli $a^d \not\equiv \pm 1 \ (mod(n))$ oraz $\forall r \in \{0, 1, ..., s-1\}$ $a^{2^r d} \not\equiv n-1 \ (mod(n))$, zwróć *False* ($n$ złożone)
- W przeciwnym wypadku zwróć *True* (n prawdopodobnie pierwsze)

Jak zatem widzimy, Test Millera-Rabina w istocie „*testuje*” złożoność wprowadzonej liczby n. Jeżeli uda się mu wykazać, że liczba jest złożona, zwraca wartość *False* - wówczas mamy pewność, że wprowadzona liczba jest liczbą złożoną. Natomiast w przypadku, gdy nie uda się dowieść złożoności liczby n, algorytm zwraca nam wartość *True* - odpowiada ona stwierdzeniu „*liczba n jest prawdopodobnie pierwsza*”. W związku z tym, tak naprawdę jedyną pewną informacją, jaką Test Millera-Rabina może zwrócić jest to, że wprowadzona liczba jest liczbą złożoną.

1.3 Prawdopodobieństwo błędu

Wiedząc już, że wynik Testu, mówiący nam o tym, że liczba jest prawdopodobnie pierwsza jest obarczony pewnym prawdodobieństwem błędu, możemy zastanowić się ile ono tak właściwie wynosi.

Otóż dowiedzono, że prawdopodobieństwo błędu (czyli zwrócenia wyniku *„n prawdopodobnie pierwsze”* dla złożonej liczby n) wynosi nie więcej niż 4^{-k}, gdzie k jest liczbą powtórzeń Testu. Wynika to z faktu, że co najwyżej $\frac{1}{4}$ wszystkich możliwych wartości liczby a może być fałszywym świadkiem pierwszości liczby n. Dlatego też $\forall k \in \mathbb{N}$ wykonując Test Millera-Rabina z $k+1$ powtórzeniami możemy czterokrotnie zmniejszyć prawdopodobieństwo błędu względem Testu powtórzonego k razy.

Mając na uwadze powyższe informacje, wiemy już, że w celu obliczenia liczby iteracji koniecznych do uzyskania wyniku z prawdopodobieństwem błędu mniejszym od 10^{-6} musimy rozwiązać następujące równanie:

$$10^{-6} = 4^{-k}$$

Które równoważnościowo możemy przekształcić do postaci:

$$k = -log_4 10^{-6}$$

Rozwiązując równanie otrzymujemy przybliżoną wartość k:

$$k \approx 9.965784...$$

Przyjmijmy $k = 10$, wówczas prawdopodobieństwo błędu będzie mniejsze od 10^{-6}.

1.4 Implementacja algorytmu w SageMath

Przejdźmy teraz do implementacji Testu Millera-Rabina w SageMath. Wykorzystamy przy tym kilka funkcji wbudowanych natywnie w SageMath, które znacznie skrócą nam czas wykonywania Testu.

Na początek importujemy potrzebną bibliotekę i ustawiamy seed dla generatora liczb losowych (nie jest to konieczne, ale robimy to, aby wyniki były powtarzalne):

[ ]:
```
import random as rd

rd.seed(int(123))  # Ustawiamy seed, aby uzyskać powtarzalność wyników
```

Korzystając z wyznaczonego wcześniej wzoru na liczbę powtórzeń Testu Millera-Rabina (aby uzyskać prawdopodobieństwo błędu mniejsze od 10^{-6}) obliczamy stałą liczbę iteracji dla każdej ze sprawdzanych liczb:

[ ]:
```
# Obliczamy liczbę iteracji Testu Millera-Rabina konieczną do uzyskania␣
↪wymaganej dokładności
k = ceil(-log(10**(-6), 4))
```

Możemy już wylosować liczby, których pierwszość będziemy sprawdzać. Generujemy 100 tysięcy losowych liczb funkcją `randint()`, która w przypadku naszego seeda generuje kilka tysięcy liczb pierwszych. W ekstremalnym przypadku (choć jak najbardziej możliwym) gdybyśmy nie zadeklarowali seeda wcześniej (który akurat generuje m.in. liczby pierwsze) moglibyśmy nie mieć żadnej liczby pierwszej wśród wylosowanych liczb.

[ ]:
```python
# Generujemy 100 tys. losowych liczb w zakresie [10^5, 10^10].
RandomNumbers = [rd.randint(10**5, 10**10) for x in range(100_000)]
```

Pora na zaimplementowanie Testu Millera-Rabina. Na początek deklarujemy funkcję wykonującą sam Test Millera-Rabina (warto zaznaczyć, że korzystamy z wbudowanej w SageMath funkcji potęgowania modulo, wyjaśnienie w dalszej części):

[ ]:
```python
def millerRabinTest(n, d, s):  # Implementujemy funkcję wykonującą Test␣
 ↪Millera-Rabina
    a = rd.randint(2, n - 2)
    x = Integer(a).powermod(d, n)

    if (x == 1 or x == (n - 1)):
        return True

    for i in range(s):
        y = (x * x) % n

        if (y == 1 and x != 1 and x != (n - 1)):
            return False

        x = y

    if (y != 1):
        return False

    return True
```

A następnie deklarujemy funkcję, która sprawdza pierwszość liczby przy wykorzystaniu wcześniej zaimplementowanego Testu Millera-Rabina:

[ ]:
```python
def isPrime(n, k):  # Definiujemy funkcję sprawdzającą pierwszość podanej␣
 ↪liczby z wykorzystaniem Testu Millera-Rabina
    if (n == 2 or n == 3):
        return True
    if (n <= 1 or (n % 2) == 0):
        return False

    s = (n - 1).valuation(2)
    d = (n - 1) / (2 ** s)

    for i in range(k):
        if not millerRabinTest(n, d, s):
            return False

    return True
```

Możemy już sprawdzić, które z wylosowanych liczb są (prawdopodobnie) liczbami pierwszymi. Dla każdej z wylosowanych liczb wywołamy funkcję sprawdzającą pierwszość z obliczoną wcześniej liczbą $k = 10$, która odpowiada za ilość iteracji Testu Millera-Rabina (innymi słowy, w oparciu o ile losowych liczb a wykonamy Test Millera-Rabina):

[ ]:
```
def CheckPrimalityInList(Numbers):
    A = []
    for n in Numbers:
        if isPrime(n, k):
            A.append(n)
    return A


PrimeNumbers = CheckPrimalityInList(RandomNumbers)
```

Zobaczmy ile liczb pierwszych udało się wylosować i jak wyglądają początkowe 3 liczby pierwsze:

[17]:
```
print("Ilość wylosowanych liczb pierwszych: ", len(PrimeNumbers))
print("Liczba nr 1: ", PrimeNumbers[0])
print("Liczba nr 2: ", PrimeNumbers[1])
print("Liczba nr 3: ", PrimeNumbers[2])
```

```
Ilość wylosowanych liczb pierwszych:  4541
Liczba nr 1:  6204179921
Liczba nr 2:  1736156999
Liczba nr 3:  1799853749
```

Na koniec możemy sprawdzić, czy nasz algorytm sprawdzania pierwszości liczby oparty o Test Millera-Rabina poprawnie określił pierwszość wygenerowanych liczb. Jeżeli poniższy skrypt zwróci nam informację, że znaleziono liczbę złożoną w liście, może to oznaczać dwie rzeczy:

1. Zaimplementowany algorytm Millera-Rabina posiada błąd w kodzie
2. Mamy bardzo duże szczęście (ale niestety niepożądane w tym przypadku), ponieważ probabilistyczny Test Millera-Rabina pomimo prawidłowej implementacji błędnie stwierdził pierwszość liczby - w tym programie szansa na to jest mniejsza niż raz na milion!

[18]:
```
for n in PrimeNumbers:
    if not is_prime(n):
        print("Znaleziono liczbę złożoną w liście!")
        break
    if n == PrimeNumbers[-1]:
        print("Sukces! Wszystkie liczby w liście są liczbami pierwszymi.")
```

```
Sukces! Wszystkie liczby w liście są liczbami pierwszymi.
```

1.5 Dodatek

Możemy jeszcze sprawdzić, czy przy użyciu własnej implementacji szybkiego potęgowania modulo, Test Millera-Rabina będzie wykonany szybciej niż przy użyciu funkcji wbudowanej w SageMath.

Implementujemy funkcję potęgowania modulo oraz funkcje do sprawdzania pierwszości liczby przy pomocy Testu Millera-Rabina:

[ ]:
```
def power_b_mod(a, b, mod):  # Definiujemy własną funkcję wykonującą operację␣
 ↪potęgowania modulo
    b = int(b)
    r = 1
    a = a % mod

    if (a == 0):
        return 0

    while (b > 0):
        if ((b & 1) == 1):
            r = (r * a) % mod

        b = b >> 1
        a = (a * a) % mod

    return r


# Implementujemy funkcję wykonującą Test Millera-Rabina z własnym potęgowaniem␣
 ↪modulo
def millerRabinTestOwnExp(n, d, s):
    a = rd.randint(2, n - 2)
    x = power_b_mod(a, d, n)

    if (x == 1 or x == (n - 1)):
        return True

    for i in range(s):
        y = (x * x) % n

        if (y == 1 and x != 1 and x != (n - 1)):
            return False

        x = y

    if (y != 1):
        return False

    return True


def isPrimeOwnExp(n, k):  # Definiujemy funkcję sprawdzającą pierwszość podanej␣
 ↪liczby z wykorzystaniem Testu Millera-Rabina
    if (n == 2 or n == 3):
        return True
    if (n <= 1 or n % 2 == 0):
```

```
        return False

    s = (n-1).valuation(2)
    d = (n-1)/(2**s)

    for i in range(k):
        if not millerRabinTestOwnExp(n, d, s):
            return False

    return True
```

Następnie porównamy czasy sprawdzania pierwszości wygenerowanych liczb, zarówno dla wcześniej zaimplementowanej funkcji, która wykorzystuje wbudowane w SageMath potęgowanie modulo oraz dla nowej funkcji, która korzysta z własnej implementacji potęgowania:

[20]:
```
def OwnExpImplementation(Numbers):
    A = []
    for n in Numbers:
        if isPrimeOwnExp(n, k):
            A.append(n)
    return A

print("Sprawdzanie czasu wykonania funkcji, może to potrwać około 1 minutę. \n")
print("Potęgowanie modulo z Sage Math: ")
%timeit X = CheckPrimalityInList(RandomNumbers)
print("Własna implementacja potęgowania modulo: ")
%timeit Y = OwnExpImplementation(RandomNumbers)
```

```
Sprawdzanie czasu wykonania funkcji, może to potrwać około 1 minutę.

Potęgowanie modulo z Sage Math:
268 ms ± 2.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Własna implementacja potęgowania modulo:
1.99 s ± 76.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Jak widać, pomimo najlepszych starań Test Millera-Rabina korzystający z własnej funkcji potęgowania modulo jest znacznie wolniejszy od tego, który wykorzystuje funkcję wbudowaną w SageMath. Złożoność czasowa Testu Millera-Rabina zależy w dużej mierze od zastosowanego algorytmu potęgowania, dlatego też warto korzystać z lepiej zoptymalizowanych funkcji.

1.6 Źródła

Do opracowania tego artykułu wykorzystano informacje z poniższych źródeł:

https://pl.wikipedia.org/wiki/Test_Millera-Rabina
https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test
https://doc.sagemath.org/html/en/reference/
https://www.cs.cmu.edu/~glmiller/Publications/Papers/Mi76.pdf https://www.sciencedirect.com/science/article
https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/