Programowanie
przed 4 wykładem

Andrzej Giniewicz

22.03.2024

Dzisiejszy blok tematyczny będzie poświęcony liczbom pierwszym.

1 Liczby pierwsze

Dla pewności przypomnijmy definicję liczb pierwszych oraz złożonych.

Definicja 1. *Liczba naturalna dodatnia n jest liczbą pierwszą, jeśli ma dokładnie dwa różne dzielniki naturalne dodatnie:* 1 *oraz n.*

Definicja 2. *Liczba naturalna dodatnia n jest liczbą złożoną, jeśli ma przynajmniej trzy różne dzielniki naturalne dodatnie.*

Z definicji tej wynika, że 1 nie jest liczbą pierwszą ani złożoną. Można debatować, dlaczego nie podjęto decyzji, żeby 1 była liczbą pierwszą. Jako pojęcie tak stare, że pojawiało się w książkach matematycznych już 300 lat przed naszą erą (w *Elementach* Euklidesa[1]), nie sposób mieć pewności, jakie były powody takiej decyzji. Niemniej jednak jednym z powodów mogło być to, żeby „działało" *podstawowe twierdzenie arytmetyki*.

Twierdzenie 1 (Podstawowe twierdzenie arytmetyki[2])**.** *Liczba naturalna* $n > 1$ *jest liczbą pierwszą albo może być jednoznacznie przedstawiona jako iloczyn liczb pierwszych.*

Gdybyśmy dopuścili 1 jako liczbę pierwszą, stracilibyśmy jednoznaczność w tym twierdzeniu, ponieważ

$$7 = 1 \cdot 7 = 1 \cdot 1 \cdot 7 = \dots$$

więc nie tylko każda liczba mogłaby być zapisana jako iloczyn liczb pierwszych, ale dodatkowo zapis ten nie byłby jednoznaczny, tylko byłoby wiele zapisów tej samej liczby. Aby *podstawowe twierdzenie arytmetyki* miało jakikolwiek sens, 1 nie może być liczbą pierwszą.

Następujące twierdzenie będzie nam przydatne do praktycznego sprawdzania, czy dana liczba jest liczbą pierwszą.

[1] Ten klasyczny tekst, na którym bazuje dużo współczesnej matematyki, można przeczytać w wersji angielskiej na stronie `https://mathcs.clarku.edu/~djoyce/java/elements/toc.html`.

[2] Zachęcam do poszperania za dowodem lub pomysłem na dowód podstawowego twierdzenia arytmetyki w VII Księdze Elementów Euklidesa.

Twierdzenie 2. *Jeśli liczba n jest liczbą złożoną, ma dzielnik k taki, że* $k \leqslant \sqrt{n}$.

Dowód. Ponieważ liczba n jest liczbą złożoną, to możemy ją zapisać jako iloczyn dwóch liczb naturalnych a i b takich, że

$$n = ab.$$

Oczywiście wtedy a i b są dzielnikami liczby n.

Załóżmy nie wprost, że a i b są jednocześnie większe od $\sqrt{n}$. Wtedy

$$n = ab > \sqrt{n}b > \sqrt{n}\sqrt{n} = n,$$

czyli $n > n$, co jest sprzeczne. Oznacza to, że nasze założenie było fałszywe. Wobec tego $a \leqslant \sqrt{n}$ lub $b \leqslant \sqrt{n}$, czyli liczba n ma dzielnik mniejszy lub równy $\sqrt{n}$. □

To pozwala nam napisać funkcję sprawdzającą, czy dana liczba jest pierwsza.

```
def czy_pierwsza(n):
    if n < 2:
        return False
    k = 2
    while k*k <= n:
        if n%k == 0:
            return False
        k += 1
    return True
```

W wersji pesymistycznej sprawdzenie, czy n jest liczbą pierwszą, wymaga dzięki temu $\sqrt{n}$ przejść pętli algorytmu. Bez twierdzenia 2, musielibyśmy wykonać aż n przejść algorytmu, więc choć wydaje się „teoretyczne", w praktyce bardzo przyspiesza działanie naszego programu.

2 Tworzenie listy liczb pierwszych

Załóżmy teraz, że chcemy stworzyć listę liczb pierwszych mniejszych od N. Pierwszy pomysł, na jaki możemy wpaść, to wykorzystanie funkcji `czy_pierwsza` z poprzedniej sekcji.

```
def pierwsze(N):
    return [n for n in range(N) if czy_pierwsza(n)]
```

To samo możemy zapisać bez list składanych.

```
def pierwsze(N):
    wynik = []
    for n in range(N):
        if czy_pierwsza(n):
            wynik.append(n)
    return wynik
```

Zastanówmy się, ile kroków wykonują te algorytmy. Jeśli przypomnimy sobie, że sprawdzenie, czy liczba n jest pierwszą, trwa około $\sqrt{n}$ kroków, to całe generowanie listy potrwa

$$\sum_{n=0}^{N-1} \sqrt{n}.$$

Zauważmy najpierw, że sumę tę można zapisać jako całkę

$$\sum_{n=0}^{N-1} \sqrt{n} = \int_1^N \sqrt{\lfloor x \rfloor}\, dx.$$

Ponieważ $x - 1 \leqslant \lfloor x \rfloor \leqslant x$, wiemy że

$$\int_1^N \sqrt{x-1}\, dx \leqslant \int_1^N \sqrt{\lfloor x \rfloor}\, \mathrm{d}x \leqslant \int_1^N \sqrt{x}\, dx.$$

Możemy zatem założyć, z monotoniczności całki, że

$$\int_1^N \sqrt{\lfloor x \rfloor}\, \mathrm{d}x \approx \int_1^N \sqrt{x - \frac{1}{2}}\, dx,$$

czyli też

$$\sum_{n=0}^{N-1} \sqrt{n} \approx \int_1^N \sqrt{x + \frac{1}{2}}\, dx.$$

Obliczmy

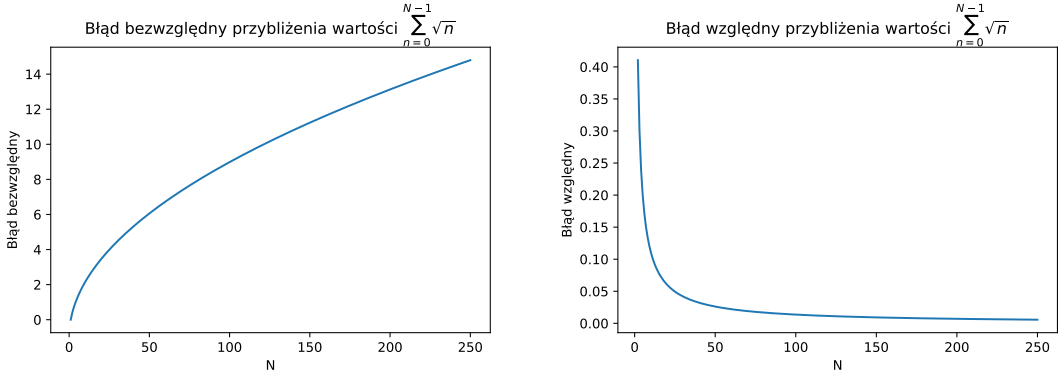
$$\sum_{n=0}^{N-1} \sqrt{n} \approx \int_1^N \sqrt{x + \frac{1}{2}}\, dx = \int_{\frac{3}{2}}^{N+\frac{1}{2}} \sqrt{t}\, dt = \left[\frac{2}{3} t^{\frac{3}{2}}\right]_{\frac{3}{2}}^{N+\frac{1}{2}} = \frac{2}{3}\left(N + \frac{1}{2}\right)^{\frac{3}{2}} - \frac{2}{3}\left(\frac{3}{2}\right)^{\frac{3}{2}} =$$
$$= \frac{2}{3}\left(N + \frac{1}{2}\right)^{\frac{3}{2}} - \sqrt{\frac{3}{2}}.$$

Aby zobaczyć jak dobre (lub niedobre) jest to przybliżenie, wykonamy wykres błędu bezwzględnego, czyli wartości

$$\frac{2}{3}\left(N + \frac{1}{2}\right)^{\frac{3}{2}} - \sqrt{\frac{3}{2}} - \sum_{n=0}^{N-1} \sqrt{n}$$

oraz błędu względnego, czyli

$$\left| \frac{\frac{2}{3}\left(N + \frac{1}{2}\right)^{\frac{3}{2}} - \sqrt{\frac{3}{2}} - \sum_{n=0}^{N-1} \sqrt{n}}{\sum_{n=0}^{N-1} \sqrt{n}} \right|.$$



Na wykresach możemy zaobserwować, że nasze przybliżenie nieznacznie przeszacowuje potrzebną liczbę kroków, dla $N = 250$ jest to około 15 kroków algorytmu więcej niż w rzeczywistości, niemniej jednak jeśli porównamy tę wartość z szacowaną wielkością, zobaczymy, że błąd ten wynosi około 0.5% (zachęcam, by wykonać obliczenia i sprawdzić).

Podsumowując, liczba kroków naszego algorytmu wyznaczania liczb pierwszych mniejszych niż N, wynosi około

$$\frac{2}{3}\left(N + \frac{1}{2}\right)^{\frac{3}{2}} - \sqrt{\frac{3}{2}}.$$

Poszukajmy efektywniejszego podejścia. Będziemy podążać za pomysłem Eratostenesa, który urodził się około 200 lat przed naszą erą. Okazuje się, że opracował on całkiem szybki algorytm, dużo lepszy niż nasz pierwszy pomysł.

2.1 Sito Eratostenesa

Eratostenes wpadł na pomysł, aby wypisać wszystkie liczby mniejsze od N i kolejno od lewej wykreślać liczby, które nie są pierwsze. Przeprowadźmy przykładowe rozumowanie dla $N = 17$

Zaczynamy od wypisania liczb naturalnych mniejszych od 17:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
```

Teraz wykreślamy 0 i 1, ponieważ nie są liczbami pierwszymi.

```
X, X, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
```

Teraz patrzymy na kolejne wartości, pomijając wykreślone. Pierwsza niewykreślona liczba będzie liczbą pierwszą. Wiemy, że dowolna jej wielokrotność nie jest liczbą pierwszą. W naszym przykładzie patrzymy na 2 i wykreślamy jej wielokrotności, czyli 4, 6, 8, 10, 12, 14 i 16. To na pewno nie są liczby pierwsze.

```
X, X, 2, 3, X, 5, X, 7, X, 9, X, 11, X, 13, X, 15, X
```

Teraz patrzymy na kolejną niewykreśloną liczbę. Jest to 3. Wykreślamy jej wielokrotności, czyli 6, 9, 12, 15.

```
X, X, 2, 3, X, 5, X, 7, X, X, X, 11, X, 13, X, X, X
```

Kolejna niewykreślona liczba to 5, ale jest ona większa od $\sqrt{16} = 4$, więc możemy przerwać wykreślanie — niewykreślone zostały nam tylko liczby pierwsze.

Spróbujmy to zaimplementować

```
def pierwsze_sito(N):
if N < 2:
    return []
kandydaci = list(range(N))
kandydaci[0] = None
```

```
    kandydaci[1] = None
    for x in kandydaci:
        if x is None:
            continue
        if x*x >= N:
            break
        for y in range(2*x, N, x):
            kandydaci[y] = None
    return [x for x in kandydaci if x is not None]
```

Implementacja jest bardzo podobna do tego, co opisaliśmy w przykładzie. Na początek tworzymy listę liczb. Wykreślanie implementujemy poprzez wstawienie wartości `None`. Następnie przechodzimy po niewykreślonych elementach i wykreślamy wielokrotności.

Spróbujmy oszacować liczbę wykreśleń wykonanych w algorytmie. Na początku mamy dwa skreślenia, potem dla liczb pierwszych mniejszych równych od $\sqrt{N-1}$, czyli mniejszych od $\sqrt{N}$, wykonujemy w przybliżeniu $\frac{N}{p_i} - 1$ skreleń, gdzie p_i to i-ta wykreślana liczba pierwsza. Jest tak, ponieważ wykreślane liczby to $2p_i$, $3p_i$, …, $\left\lfloor \frac{N}{p_i} \right\rfloor \cdot p_i$.

Oznacza to, że liczba kroków będzie wynosić w przybliżeniu

$$2 + \sum_{p_i \in \mathbb{P}, p_i < \sqrt{N}} \left(\frac{N}{p_i} - 1 \right),$$

gdzie $\mathbb{P}$ to zbiór liczb pierwszych. Jeśli przez $\pi(N)$ oznaczymy liczbę liczb pierwszych mniejszych lub równych od N, to wzór sprowadzi się do

$$2 + N \left(\sum_{p_i \in \mathbb{P}, p_i < \sqrt{N}} \frac{1}{p_i} \right) - \pi(\sqrt{N}).$$

Wartość ta jest w przybliżeniu równa

$$2 + N(\ln(\ln(\sqrt{N})) + M) - \pi(\sqrt{N}),$$

gdzie $M \approx 0.2614972$ jest stałą Martensa[3]. Liczba liczb pierwszych może być przybliżona następującym wzorem

$$\pi(N) \approx \frac{N}{\ln(N) + \gamma - \frac{3}{2}},$$

gdzie $\gamma \approx 0.5772156649$ jest stałą Eulera-Mascheroniego[4]. Podsumowując, liczba wykreśleń w algorytmie wynosi w przybliżeniu

$$2 + N(\ln(\ln(\sqrt{N})) + M) - \frac{\sqrt{N}}{\ln(\sqrt{N}) + \gamma - \frac{3}{2}}.$$

[3] Patrz `https://mathworld.wolfram.com/MertensConstant.html`.

[4] Patrz `https://mathworld.wolfram.com/PrimeCountingFunction.html`, `https://mathworld.wolfram.com/HarmonicNumber.html` oraz `https://mathworld.wolfram.com/Euler-MascheroniConstant.html`.

Możemy wyciągnąć pierwiastek spod logarytmu

$$2 + N\left(\ln\left(\frac{\ln(N)}{2}\right) + M\right) - \frac{\sqrt{N}}{\frac{\ln(N)}{2} + \gamma - \frac{3}{2}}$$

i zamienić logarytm ilorazu na różnicę logarytmów

$$2 + N(\ln(\ln(N)) - \ln(2) + M) - \frac{2\sqrt{N}}{\ln(N) + 2\gamma - 3}$$

Wstawiając przybliżone wartości $\ln(2)$, M oraz γ, otrzymujemy przybliżoną liczbę kroków algorytmu

$$N\ln(\ln(N)) + 2 - 0.4316499677123025N - \frac{2\sqrt{N}}{\ln(N) - 1.8455686701969342}.$$

Można w szacowaniu bezpiecznie przyjąć przybliżenie

$$N\ln(\ln(N)),$$

ponieważ jest to najbardziej istotny składnik sumy powyżej — stała 2 szybko staje się mała w porównaniu z $N\ln(\ln(N))$, natomiast pozostałe składniki są ujemne dla dużych N, więc $N\ln(\ln(N))$ stanowi pewnego rodzaju pesymistyczne ograniczenie z góry. Podsumowując, sito Eratostenesa jest zdecydowanie szybszym rozwiązaniem niż poprzednie, które wykonywało

$$\frac{2}{3}\left(N + \frac{1}{2}\right)^{\frac{3}{2}} - \sqrt{\frac{3}{2}}$$

operacji.

Kolejna optymalizacja wynika z kolejności wykreślania. Spójrzmy raz jeszcze na kod

```
def pierwsze_sito(N):
    if N < 2:
        return []
    kandydaci = list(range(N))
    kandydaci[0] = None
    kandydaci[1] = None
    for x in kandydaci:
        if x is None:
            continue
        if x*x >= N:
            break
        for y in range(2*x, N, x):
            kandydaci[y] = None
    return [x for x in kandydaci if x is not None]
```

Podczas wykreślania wielokrotności liczby x, wszystkie wielokrotności od $2x$ do $(x-1)x$ zostały już wykreślona podczas usuwania wielokrotności liczb od 2 do $x-1$. Oznacza to, że możemy zacząć od liczby x^2.

```
def pierwsze_sito(N):
    if N < 2:
        return []
    kandydaci = list(range(N))
    kandydaci[0] = None
    kandydaci[1] = None
    for x in kandydaci:
        if x is None:
            continue
        if x*x >= N:
            break
        for y in range(x*x, N, x):
            kandydaci[y] = None
    return [x for x in kandydaci if x is not None]
```

Korzystając z oszacowania na sumę liczb pierwszych[5], możemy wyliczyć liczbę kroków poprawionego algorytmu. Pomimo zastosowanej optymalizacji, ponieważ wszystkie stałe znikają w zapisie „duże O", możemy stwierdzić, że sito Eratostenesa jest algorytmem $O(N \log(\log(N)))$ w porównaniu do naiwnego podejścia, które jest $O\left(N^{\frac{3}{2}}\right)$.

3 Liczba liczb pierwszych

W poprzedniej sekcji korzystaliśmy z funkcji $\pi(N)$, która oznacza liczbę liczb pierwszych mniejszych lub równych N. Podaliśmy przybliżenie

$$\pi(N) \approx \frac{N}{\ln(N) + \gamma - \frac{3}{2}}.$$

Sprawdzimy je w Pythonie. Najpierw musimy zaimplementować funkcję $\pi(N)$. Dla wygody, ponieważ zamierzamy ją rysować, napiszemy w Pythonie funkcję `ile_pierwszych(N)`, która zwróci listę wartości $[\pi(0), \pi(1), \ldots, \pi(N-1)]$.

```
def ile_pierwszych(N):
    P = pierwsze_sito(N)
    wyniki = []
    y = 0
    n = len(P)
    for x in range(N):
        if y<n and P[y] <= x:
            y += 1
        wyniki.append(y)
    return wyniki
```

[5] Patrz https://mathworld.wolfram.com/PrimeSums.html.

Prześledź algorytm na stronie `https://tinyurl.com/primecounting-30`, wykonując go krok po kroku dla $N = 30$. Zwróć uwagę, jak natrafienie zmiennej `x` na liczbę pierwszą `P[y]` powoduje zwiększenie licznika liczb pierwszych o 1. Interpretacja tej funkcji wymaga chwilę pracy, nie bój się przeanalizować jej na kartce.

Aby zbadać przybliżenie, skorzystamy z wykresu. Zachęcam, aby wpisać poniższy kod do Pythona i pobawić się parametrami, aby zbadać, co się dzieje z przybliżeniem dla małych wartości N.

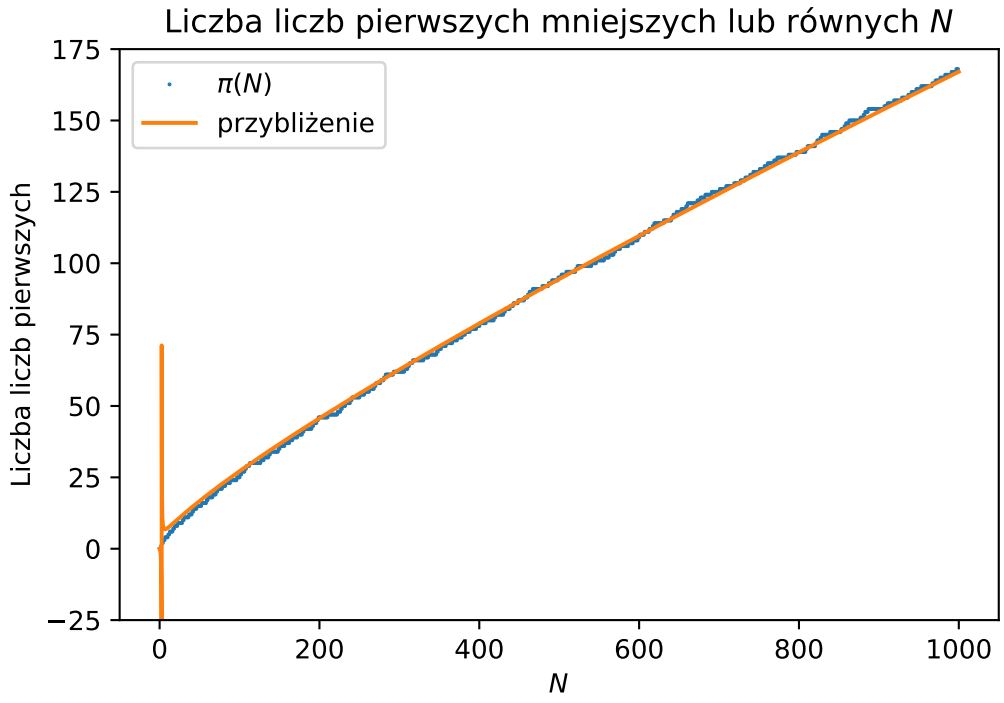
```
import matplotlib.pyplot as plt
import numpy as np

gamma = 0.5772156649
N = np.linspace(0.01, 1000, 10000)
approx = N/(np.log(N)+gamma-1.5)

plt.plot(range(1000), ile_pierwszych(1000), '.', label=r"$\pi(N)$", markersize=1)
plt.plot(N, approx, label=r"przybliżenie")

plt.legend()
plt.ylabel("Liczba liczb pierwszych")
plt.xlabel("$N$")
plt.ylim(-25, 175)
plt.title("Liczba liczb pierwszych mniejszych lub równych $N$")
```

Wynik działania programu możemy zobaczyć na ilustracji poniżej.



4 Tworzenie listy liczb pierwszych zadanej długości

Jeśli naszym zadaniem jest znalezienie pierwszych N liczb pierwszych, sprawa się nieco komplikuje — aby stworzyć sito, musimy znać ograniczenie górne na N-tą liczbę pierwszą.

Okazuje się, że dla $N > 6$, N-ta liczba pierwsza p_N spełnia nierówność[6]

$$p_N < N\ln(N) + N\ln(\ln(N)),$$

przy czym ponieważ $P_6 = 17$, możemy założyć, że

$$p_N < \max\{17, \lceil N\ln(N) + N\ln(\ln(N))\rceil\}.$$

Korzystając z tego, można napisać funkcję, która znajduje pierwszych N liczb pierwszych

```
from math import log, ceil

def pierwsze_pierwsze(N):
    if N > 6:
        NO = ceil(N*log(log(N))+N*log(N))
    else:
        NO = 17
    P = pierwsze_sito(NO)
    return P[:N]
```

Dzięki oszacowaniu górnemu możemy wygenerować sito właściwego rozmiaru, następnie znaleźć liczby pierwsze mniejsze od ograniczenia i mieć gwarancję, że jest tam przynajmniej N liczb pierwszych, które zwracamy.

[6] To i inne przybliżenia dla liczb pierwszych znajdziesz w artykule — J. Barkley Rosser, Lowell Schoenfeld, „Approximate formulas for some functions of prime numbers", Illinois J. Math. Volume 6, Issue 1 (1962), 64-94. (https://projecteuclid.org/euclid.ijm/1255631807).