Programowanie
po 4 wykładzie

Andrzej Giniewicz

22.03.2024

Kontynuujemy zajmowanie się podzielnością oraz tematami pokrewnymi. Dziś poznamy prawdopodobnie najstarsze otwarte pytanie w matematyce, hipotezę mającą ponad 2000 lat, na którą do dziś nie znamy odpowiedzi[1].

1 Wyznaczanie wszystkich dzielników liczby

Dzielnik właściwy to dzielnik, który nie jest równy samej liczbie. Pierwszy pomysł na implementację to

```
def lista_dzielników_prosta(N, właściwe = True):
    dzielniki = []
    for n in range(1, N if właściwe else N+1):
        if N%n == 0:
            dzielniki.append(n)
    return dzielniki
```

Jest to prawdopodobnie najprostszy sposób na wypisanie wszystkich dzielników. Niestety, jest on dość czasochłonny. Pomocny w napisaniu szybszej wersji będzie rozkład na czynniki pierwsze. Na zajęciach napisaliśmy naiwną wersję tego algorytmu, teraz udoskonalimy ją, wykorzystując sito Eratostenesa. Zacznijmy od sita. Poprawimy je, dodając `if` na początku, ponieważ dla $N < 2$ nie ma liczb pierwszych mniejszych równych N, czyli możemy od razu zwrócić pustą listę. Nasza poprzednia wersja algorytmu działała tylko, gdy w wyniku mogła się znaleźć choć jedna liczba pierwsza, czyli było czego szukać.

[1] Takich problemów jest wiele — niektóre tak ważne, że za ich rozwiązanie istnieje nagroda w wysokości 1 mln dolarów. Są to problemy milenijne (`https://www.claymath.org/millennium-problems`), z których dopiero jeden — Hipoteza Poincarégo — został całkowicie rozwiązany przez rosyjskiego matematyka Grigorija Jakowlewicza Perelmana (Григорий Яковлевич Перельман), który co ciekawe odmówił przyjęcia nagrody, podobnie jak odmówił przyjęcia medalu Fieldsa, nagrody matematycznej o randze podobnej do nagrody Nobla. Zachęcam do poczytania zarazem o otwartych problemach matematyki, jak i o osobach, które się z nimi mierzą, niekiedy wręcz obsesyjnie poświęcając się im w całości. Niestety problem, który dziś omówimy, nie jest problemem milenijnym, ale osoba, która go rozwiąże, też zapisze się na kartach historii (przynajmniej historii matematyki).

```
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]
```

Teraz możemy wykorzystać listę liczb pierwszych wygenerowanych przez

```
def czynniki_pierwsze(n):
    wyniki = {}
    for p in pierwsze_sito(n+1):
        ile_razy = 0
        while n%p == 0:
            ile_razy += 1
            n //= p
        if ile_razy:
            wyniki[p] = ile_razy
    return wyniki
```

Teraz możemy przejść do implementacji listy dzielników. Skorzystamy z obserwacji dla następującego przykładu. Jeśli liczba jest postaci

$$2^3 3^2 5^7 13^1,$$

to aby uzyskać jej wszystkie dzielniki, możemy najpierw policzyć wszystkie dzielniki liczby

$$3^2 5^7 13^1$$

a następnie przemnożyć je kolejno przez

$$2^0, 2^1, 2^2, 2^3,$$

czyli wszystkie możliwe wielokrotności dwójki wśród dzielników. Algorytm taki można na przykład zaimplementować rekurencyjnie, jeśli rozkład na czynniki zapiszemy jako listę par postaci „liczba pierwsza", „wykładnik" — czyli dla naszego przykładu

$$[(2, 3), (3, 2), (5, 7), (13, 1)].$$

Napiszemy funkcję pomocniczą.

```
def wymnażaj(pary):
    if not pary:
        return [1]
    (n, k) = pary[0]
    bez_pierwszego = wymnażaj(pary[1:])
    wynik = []
    mnoznik = n
    for _ in range(k):
        wynik += [x*mnoznik for x in bez_pierwszego]
        mnoznik *= n
    return bez_pierwszego + wynik
```

Przeanalizujmy tę funkcję. Jeśli lista par jest pusta, zwraca listę `[1]`. Pusta lista par reprezentuje jedynkę, zatem lista dzielników to właśnie `[1]`. Oznacza to, że ten przypadek mamy poprawnie działający. Dla niepustych par wyciągamy pierwszy element (n, k) a na pozostałych wywołujemy rekurencyjnie funkcję, uzyskując wszystkie dzielniki liczby bez pierwszego czynnika n^k. Następnie budujemy listę dzielników, mnożąc wartości z wywołania rekurencyjnego przez kolejne potęgi liczby n.

Używanie tej funkcji nie jest wygodne, ponieważ musimy zamienić liczbę na listę par, zatem stworzymy dodatkową funkcję, które wykorzysta wcześniej zdefiniowane funkcje pomocnicze.

```
def lista_dzielników(N, właściwe = True):
    czynniki = list(czynniki_pierwsze(N).items())
    dzielniki = sorted(wymnażaj(czynniki))
    if właściwe:
        return dzielniki[:-1]
    else:
        return dzielniki
```

Funkcja rozkłada N na czynniki pierwsze, reprezentowane przez słownik. Wobec tego metodą `items` możemy wyciągnąć listę par, uzyskując czynniki. Czynniki wymnażamy funkcją pomocniczą, po czym je sortujemy dla uporządkowania. Dzięki temu dzielnik N jest na samym końcu. Jeśli użytkownik ustawił opcję `właściwe` na `True`, wyrzucamy ostatni dzielnik, czyli N. W momencie, gdy dysponujemy już funkcjami pomocniczymi, rozbiliśmy większy problem na kilka mniejszych podproblemów, dzięki czemu funkcja wyznaczająca listę dzielników powinna być łatwiejsza do złożenia z gotowych już bloków.

Jeśli zainteresowana osoba wykona pomiar czasu wykonania algorytmu, przekona się, że nasze zabiegi na niewiele się zdały. A dzieje się tak dlatego, że przy każdym wywołaniu funkcji, wykonujemy od nowa generowanie sita. Tymczasem, sito moglibyśmy zapamiętać na przyszłe wywołania i wykorzystać je ponownie. Użycie `lru_cache` jest jedną z możliwości, ale w tym przypadku strategia ta nie jest idealna. Pozwala ona jedynie na zapamiętanie dokładnie tego samego wyniku. Jeśli liczyliśmy sito dla $N = 100$ a poprosimy o sito dla

$n = 99$, będzie musiało zostać wyznaczone od nowa. Tymczasem liczby pierwsze mniejsze lub równe N mogą posłużyć do prostego wygenerowania liczb pierwszych mniejszych lub równych dowolnemu $n \leqslant N$. Wtedy, jeśli wywołujemy funkcję z takim samym parametrem, zwracamy zapamiętamy wynik, natomiast jeśli z mniejszym, ograniczamy wynik, filtrując go, jeśli natomiast z większym, nadbudowujemy listę, aby stała się większa. Aby zaimplementować własną strategię przechowywania wyników pośrednich, możemy posłużyć się zmiennymi globalnymi. Słówko kluczowe `global` pozwala na modyfikację zmiennej zdefiniowanej poza funkcją, dzięki czemu możemy w niej zapamiętać wyniki pomiędzy wywołaniami[2]. Początkowa inicjalizacja zmiennych globalnych będzie przechowywała wyniki dla $N = 2$.

```
_N_DLA_SITA = 2
_WYNIK_SITA = [2]
_SITO = [None, None, 2]

def sito_eratostenesa_cache(N):
    global _N_DLA_SITA
    global _WYNIK_SITA
    global _SITO
    if N == _N_DLA_SITA:
        return _WYNIK_SITA.copy()
    elif N < _N_DLA_SITA:
        return [x for x in _WYNIK_SITA if x<=N]
    _SITO += list(range(_N_DLA_SITA+1, N+1))
    for p in _SITO:
        if p is None:
            continue
        if p*p > N:
            break
        for x in range(max(p, (_N_DLA_SITA//p+1))*p, N+1, p):
            _SITO[x] = None
    _WYNIK_SITA = [x for x in _SITO if x is not None]
    _N_DLA_SITA = N
    return _WYNIK_SITA.copy()
```

Zwróćmy uwagę na kilka szczegółów. Funkcja nie zwraca wprost wartości `_WYNIK_SITA`, tylko jego kopię, aby użytkownik mógł modyfikować listę i nie martwić się o uszkodzenie zmiennej globalnej i kolejnych wywołań funkcji generującej listę liczb pierwszych. Dodatkowo etap generowania sita jest odrobinę inny niż etap tworzenia nowego sita — tutaj sito rośnie, zatem doczepiamy do zmiennej `_SITO` tylko nowe liczby, a wykreślając, upewniamy się, żeby niepotrzebnie nie wykreślać tego, co już wcześniej wykreśliliśmy. Zachęcam, aby prześledzić

[2]Nie jest to najbardziej elegancki sposób, ale jest najprostszy. Lepszy sposób poznamy, gdy zaczniemy przerabiać programowanie obiektowe.

ten fragment kodu dla przykładowych uruchomień. Przy zastosowaniu zapamiętywania wyników pośrednich, funkcja szukająca dzielników liczb jest znacznie wydajniejsza. Warto tę procedurę zastosować, jeśli wyliczamy wiele różnych dzielników różnych liczb. Jeśli liczymy tylko raz dzielniki jednej liczby i nie wracamy do tego, rozwiązanie „wprost" z początku podrozdziału jest wystarczające i może nawet w niektórych przypadkach małego N działać nieco szybciej.

Algorytm wykorzystujący liczby pierwsze i zapamiętywanie wyników pośrednich znalazł dzielniki wszystkich stu liczb w $2.12s$, podczas gdy algorytm naiwny zużył aż $6.26s$, czyli rozwiązanie, w którym się nieco nagłowiliśmy, jest w tym przypadku prawie trzy razy szybsze.

2 Ciąg podwielokrotności

Ciąg podwielokrotności liczby n to ciąg s_i, $i = 0, 1, 2, \ldots$ taki, że $s_0 = n$, natomiast s_{i+1} jest sumą dzielników właściwych liczby s_i. Ciąg podwielokrotności danej liczby jest okresowy o okresie $k > 0$, jeśli $s_i = s_{i \bmod k}$. Jeśli ciąg jest okresowy, możemy znaleźć okres zasadniczy ciągu, czyli taką liczbę k, że ciąg jest okresowy o okresie k i nie istnieje mniejsza liczba $m < k$ taka, że ciąg jest okresowy o okresie m. Innymi słowy, jeśli ciąg jest okresowy, możemy znaleźć najmniejszy okres ciągu.

Jeśli ciąg podwielokrotności zaczynający się od liczby n jest okresowy o okresie zasadniczym k, mówimy, że liczba n jest liczbą towarzyską rzędu k. Liczby towarzyskie rzędu 2 nazywamy liczbami zaprzyjaźnionymi, natomiast liczby towarzyskie rzędu 1 nazywamy liczbami doskonałymi. Dzięki funkcji do znajdowania dzielników, możemy napisać sprawdzenie, czy liczba jest doskonała.

```
def czy_doskonała(N):
    return N == sum(lista_dzielników(N))
```

Przykładowo, liczba 6 jest doskonała, ponieważ jej dzielniki właściwe (1, 2, 3) sumują się do 6, czyli ciąg podwielokrotności zaczynający się od liczby 6 jest stały (ma okres 1). Inną liczbą doskonałą jest 28 ($1 + 2 + 4 + 7 + 14 = 28$). Definicja liczb doskonałych jest znana od starożytności[3]. Liczbami zaprzyjaźnionymi zajmowali się też Pitagorejczycy, którzy znali na przykład parę liczb (220, 284), które stanowią naprzemiennie ciąg podwielokrotności zaczynający się od 220. Dostrzegali oni w tych liczbach definicję przyjaźni, ponieważ „suma części jednej z nich daje drugą", co próbowali przekładać na otaczający ich świat w tym filozofię i relacje międzyludzkie. Liczby doskonałe oraz zaprzyjaźnione były i są wykorzystywane przez mistyków i numerologów oraz najróżniejsze religie (stworzenie świata wg. Biblii trwało 6 dni, 220 kóz i 220 baranów było gestem przyjaźni, itp.). Fascynacja liczbami doskonałymi bierze się również z natury, ponieważ przykładowo 28 dni trwa miesiąc księżycowy.

[3] Znajduje się między innymi w księdze VII *Elementów* Euklidesa.

Tu przechodzimy do najprawdopodobniej najstarszego otwartego problemu matematyki — datowanego na około 100 rok naszej ery, gdy powstało dzieło *Introductio Arithmetica* Nikomachosa z Gerazy[4] — wciąż **nie wiadomo, czy istnieje jakakolwiek nieparzysta liczba doskonała**.

Okazuje się, że liczby doskonałe nie mają jedynie zastosowania w mistycyzmie. Już Euklides udowodnił[5] jeśli p i $2^p - 1$ są liczbami pierwszymi, to $2^{p-1}(2^p - 1)$ jest parzystą liczbą doskonałą. Dziś na liczby pierwsze postaci $2^p - 1$, gdzie p jest liczbą pierwszą, mówimy „liczby Mersenne'a". Odwrotne twierdzenie, mówiące, że jeśli $2^{p-1}(2^p - 1)$ jest parzystą liczbą doskonałą, to $2^p - 1$ jest liczbą pierwszą Mersenne'a, udowodnił Euler w XVIII wieku naszej ery. Dziś twierdzenie mówiące o obustronnym wynikaniu nazywa się twierdzeniem Euklidesa-Eulera. Liczby pierwsze Mersenne'a mają szczególne znaczenie dla kryptografii ze względu na swoją szczególną postać[6], wobec czego ten, kto zna większą liczbę pierwszą Mersenne'a a tym samym, ten kto zna większą liczbę doskonałą, ma lepszą technikę szyfrowania wiadomości, które znacznie trudniej złamać. Dziś znamy zaledwie 51 liczb doskonałych[7]. Więcej o nich można przeczytać w bazie znanych ciągów `https://oeis.org/A000396`[8].

[4] Notka biograficzna: `https://mathshistory.st-andrews.ac.uk/Biographies/Nicomachus/`.

[5] Patrz Księga IX *Elementów* Euklidesa.

[6] Liczba postaci $2^p - 1$ w zapisie dwójkowym jest ciągiem zer, po którym następuje ciąg jedynek — na przykład $00000011_2 = 3, 00000111_2 = 5, 00011111_2 = 31$ oraz $01111111_2 = 127$ to liczby pierwsze Mersenne'a mieszczące się w 8 bitach pamięci.

[7] Znane liczby Mersenne'a a tym samym znane liczby doskonałe znajdziemy na stronie `https://www.mersenne.org/various/history.php`.

[8] Bazę znanych ciągów publikuje matematyk Neil James Alexander Sloane. Niel Sloane jest częstym gościem kanału Numberphile na YouTube (`https://www.youtube.com/numberphile`). Listę jego ciekawych opowieści o interesujących ciągach i liczbach można znaleźć na stronie `http://neilsloane.com/doc/videos_1.html`.