Algorytmy w teorii liczb

Teoria liczb jest działem matemtyki dotyczącym własności liczb naturalnych. Rozważa się zagadnienia związane z liczbami pierwszymi, podzielnością liczb, rozkładem liczb na czynniki pierwsze, kongruencje itd. Formułuje się także wiele ciekawych problemów algorytmicznych, z których wiele ma ważne zastosowania, np. w kryptografii (dziedzina zajmująca się szyfrowaniem informacji). Oto przykład dobrze wszystkim znanego problemu algorytmicznego:

Dane: liczby a i b

Problem: znaleźć liczbę $c = a \cdot b$.

Znamy oczywiście „szkolny" algorytm rozwiązywania tego problemu. Powstaje pytanie jaka jest jego złożoność czasowa. I tu ważna informacja: rozważając problemy teorioliczbowe jako rozmiar danych przyjmujemy **długość zapisu liczby** w pewnym układzie pozycyjnym (np. dziesiątkowym lub dwójkowym). Długość zapisu liczby x w układzie o podstawie b będziemy oznaczać przez $lh_b(x)$.

Fakt 1. Długość zapisu liczby x w układzie o podstawie b wynosi $lh_b(x) = 1 + \lfloor \log_b x \rfloor$.

Uzasadnienie. To, że n jest długością zapisu liczby x w układzie o podstawie b oznacza, że

$$x = c_{n-1}b^{n-1} + c_{n-2}b^{n-2} + \ldots + c_1 b + c_0$$

gdzie c_i – to kolejne cyfry, tzn. $0 < c_{n-1} < b$ oraz $0 \leq c_i < b$ dla $i = 0, 1, \ldots c_{n-2}$. Wtedy

$$b^n > x \geq b^{n-1}.$$

Zlogarytmujmy obustronnie, otrzymujemy $n > \log_b x \geq n - 1$. To oznacza dokładnie tyle, że $n - 1 = \lfloor \log_b x \rfloor$. A więc $lh_b(x) = n = 1 + \lfloor \log_b x \rfloor$. □

Jeśli interesuje nas złożoność asymptotyczna algorytmów, to nie ma znaczenia w jakim układzie pozycyjnym liczymy, bo przecież $lh_a(x) = 1 + \lfloor \log_a x \rfloor = 1 + \lfloor \frac{1}{\log_b a} \cdot \log_b x \rfloor = O(\log_b x)$. W dalszej kolejności będziemy więc zakładać, że liczby są dane binarnie, tzn. $b = 2$ oraz oznaczymy $lh(x) = lh_2(x)$.

Fakt 2. Niech $lh(x) = n$, $lh(y) = m$. Wtedy złożoności czasowe algorytmów „szkolnych" wynoszą odpowiednio:

- dla dodawania: $O(n + m)$,
- dla odejmowania: $O(n + m)$,
- dla mnożenia: $O(nm)$,
- dla dzielenia (obliczanie wyniku z dzielenia i reszty): $O(nm)$.

Dodawanie i odejmowanie działa w czasie liniowym a więc optymalnie. Okazuje się, że można szybciej. Niech $lh(a) = lh(b) = 2n$. Przedstawimy liczby a i b w następującej postaci:

$$a = 2^n A_1 + A_2, \quad b = 2^n B_1 + B_2,$$

gdzie $lh(A_1) = lh(A_2) = lh(B_1) = lh(B_2) = n$. Wtedy:

$$\begin{aligned}
&(2^{2n} + 2^n)A_1B_1 + 2^n(A_1 - A_2)(B_2 - B_1) + (2^n + 1)A_2B_2 = \\
&2^{2n}A_1B_1 + 2^nA_1B_1 + 2^n(A_1B_2 - A_1B_1 - A_2B_2 + A_2B_1) + 2^nA_2B_2 + A_2B_2 = \\
&2^{2n}A_1B_1 + 2^nA_1B_2 + 2^nA_2B_1 + A_2B_2 = \\
&ab.
\end{aligned}$$

W pierwszej linijce mamy sposób obliczania iloczynu ab. Zauważmy, że mnożenie przez potęgę dwójki 2^k wykonujemy w czasie $O(k)$ – liczby mamy zapisane binarnie a więc takie mnożenie to po prostu dodanie k zer na końcu. Oprócz dopisywania zer mamy 6 operacji dodawania i odejmowania, które w sumie zajmują czas $O(n)$ no i trzy mnożenia liczb długości n a więc dwa razy krótszych. Te mnożenia wykonujemy rekurencyjnie. Jaką złożoność ma nasz algorytm? Niech $T(n)$ będzie liczbą wykonywanych operacji. Wtedy:

$$T(2n) \leq 3T(n) + Cn.$$

dla pewnej stałej C. Rozwiążemy równanie rekurencyjne:

$$\begin{aligned}
&T(1) = 1, \\
&T(2n) = 3T(n) + Cn.
\end{aligned}$$

Przy obliczaniu złożoności wystarczy założyć, że n jest potęgą dwójki. Wtedy:

$$T(2^k) = 3T(2^{k-1}) + C2^{k-1}.$$

I dalej:

$$\begin{aligned}
T(2^k) &= 3T(2^{k-1}) + C2^{k-1} = \\
&= 3(T(2^{k-2}) + C2^{k-2}) + C2^{k-1} = \\
&= 3^2(T(2^{k-3}) + C2^{k-3}) + 3C2^{k-2} + C2^{k-1} = \\
&\dots \\
&= 3^{k-1}T(2^0) + 3^{k-1}C2^0 + 3^{k-2}C2^1 + \ldots + 3C2^{k-2} + C2^{k-1} \leq \\
&\leq 3^k c = \\
&= (2^{\log_2 3})^k = \\
&= (2^k)^{\log_2 3}.
\end{aligned}$$

dla pewnej stałej c. Stąd $T(n) = n^{\log_2 3}$. Tymczasem

$$\log_2 3 \approx 1,585$$

czyli mamy algorytm w czasie $O(n^{1,585})$ czyli dużo lepiej niż n^2.

NWD i Algorytm Euklidesa

Przypomnijmy kilka prostych pojęć. Zbiór liczb całkowitych oznaczamy symbolem $\mathbf{Z}$. Liczba całkowita $d \geq 0$ jest dzielnikiem liczby cakowitej a, gdy dla pewnego $k \in \mathbf{Z}$ ma miejsce $a = k \cdot b$. Piszemy wtedy $d \mid a$ i mówimy, że d *dzieli* a.

Przykład: $2 \mid 10, \quad 5 \mid 5, \quad 2 \not| \, 3.$

Największy wspólny dzielnik liczb a i b to największa liczba, która jest jednocześnie dzielnikiem a i dzielnikiem b, inaczej:

$$\mathrm{NWD}(a, b) = \max\{d \in \mathbf{Z} : d \mid a \text{ oraz } d \mid b\}.$$

Przykład: $\mathrm{NWD}(15, 6) = 3, \quad NWD(18, 12) = 6, \quad NWD(7, 15) = 1, \quad NWD(a, 0) = 6$ dla $a > 0$.

Zajmiemy się następującym problemem algorytmicznym:

Dane: liczby a i b

Problem: znaleźć $\mathrm{NWD}(a, b)$.

Twierdzenie (algorytm „szkolny"). *Jeśli mamy rozkład liczb a i b na czynniki pierwsze:*

$$a = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_n^{\alpha_n},$$
$$b = p_1^{\beta_1} \cdot p_2^{\beta_2} \cdot \ldots \cdot p_n^{\beta_n}.$$

to wtedy

$$\mathrm{NWD}(a, b) = p_1^{\min(\alpha_1,\beta_1)} \cdot p_2^{\min(\alpha_2,\beta_2)} \cdot \ldots \cdot p_n^{\min(\alpha_n,\beta_n)}.$$

Oczywiście w powyższym twierdzeniu dopuszczamy możliwość $\alpha_i = 0$, $\beta_i = 0$ – dzięki temu ten zapis ma sens nawet gdy pewna liczba pierwsza p_i dzieli a ale nie dzieli b. Zauważmy, że powyższy wzór odzwierciedla dokładnie sposób, w jaki obliczają NWD dzieci w szkole. Zauważmy jednak, że taki algorytm ma jedynie znaczenie dydaktyczne pozwalające oswoić się z faktem, że wszystkie liczby „składają się" z liczb pierwszych. Gdybyśmy chcieli taki algorytm zapisać jako program komputerowy powstaje pytanie jak znajdować rozkłady liczb na czynniki pierwsze. Okazuje się jednak że znajdowanie takiego rozkładu to bardzo trudny problem: żeby rozłożyć liczbę długości n najlepsze znane algorytmy potrzebują czasu wykładniczego w rodzaju $O(e^{n^{1/3} \log^{2/3} n})$. A jaka jest złożoność Algorytmu szkolnego? Załóżmy, że rozkładamy n-cyfrową liczbę a. Bierzemy kolejne liczby pierwsze (wystarczy od 2 do $\lfloor\sqrt{a}\rfloor$) i dla każdej z nich sprawdzamy, czy dzielą a. Tymczasem prawdziwe jest twierdzenie:

Twierdzenie. *Jest* $\Theta(k/\log k)$ *liczb pierwszych mniejszych od* k.

Jeśli a jest n-cyfrowa to $10^n > a \geq 10^{n-1}$, czyli $10^{n/2} > \sqrt{a} \geq 10^{(n-1)/2}$ a zatem liczb pierwszych mniejszych od $\lfloor\sqrt{a}\rfloor$ jest $\Theta(10^{(n-1)/2}/n)$. A więc algorytm szkolny też jest wykładniczy. Pokażemy lepszy algorytm. Potrzebny będzie następujący fakt:

Fakt 3. $\mathrm{NWD}(a + cb, b) = \mathrm{NWD}(a, b)$.

Dowód. Pokażemy, że wspólne dzielniki liczb $a + cb, b$ są takie same jak wspólne dzielniki liczb a i b. Wtedy największe wspólne dzielniki też będą takie same.

Jeśli $d \mid a$ i $d \mid b$ to $d \mid (a + cb)$ czyli jeśli d jest wspólnym dzielnikiem a i b to jest też wspólnym dzielnikiem $a + cb$ i b.

Na odwrót, jeśli $d \mid (a + cb)$ oraz $d \mid b$, czyli $(a + cb) = k \cdot d$ oraz $b = l \cdot d$ to $a = kd - cb = kd - cld - d(k - lc)$ a więc $d \mid a$. Stąd d jest wspólnym dzielnikiem a i b. □

Co wynika z tego faktu? Jeśli chcemy obliczyć $\mathrm{NWD}(x, y)$ dla $x > y$ to wykonujemy dzielenie z resztą x przez y. Wtedy $x = ky + r$ gdzie $0 \le r < y$ (r jest resztą). Wiemy, że $\mathrm{NWD}(x, y) = \mathrm{NWD}(ky + r, y) = \mathrm{NWD}(r, y)$. Powtarzamy tę operację aż jedna z liczb będzie równa zero. Oto pseudokod algorytmu:

Algorithm 1 Algorytm Euklidesa

function Euklides (a, b) : Integer
if $b = 0$ **then**
    $Result \leftarrow a$
else
    $Result \leftarrow$ Euklides $(b,\ a\ \mathbf{mod}\ b)$

Przykład. Obliczymy $\mathrm{NWD}(574, 252)$:

$$
\begin{aligned}
574 &= 2 \cdot 252 + 70, \\
252 &= 3 \cdot 70 + 42, \\
70 &= 1 \cdot 42 + 28, \\
42 &= 1 \cdot 28 + 14, \\
28 &= 2 \cdot 14 + 0,
\end{aligned}
$$

A więc $\mathrm{NWD}(574, 252) = 14$.

Wbrew pozorom algorytm Euklidesa wymyślił Eudoksos w IV wieku p.n.e. Wymyślił też kilka innych pożytecznych rzeczy jak np. liczby rzeczywiste albo tzw. metodę wyczerpywania, podobną do współczesnej całki Riemmana. Wkład Euklidesa jest natomiast taki, że opisał on nasz algorytm w swoich *Elementach*.

Zastanówmy się nad złożonością algorytmu Euklidesa. Jaki jest pesymistyczny przypadek? Najgorzej jest gdy zawsze wynik dzielenia a **div** $b \le 1$, bo wtedy liczby najwolniej „spadają”. Tak jest dla liczb Fibonacciego. Dla przypomnienia n-ta liczba Fibonacciego dana jest wzorem rekurencyjnym: $F_1 = 1$, $F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$ dla $n > 2$. Kolejne liczby Fibonacciego to: $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$. Policzmy np NWD(21, 13):

$$\begin{aligned}
F_8 &= 21 = 1 \cdot 13 + 8, \\
F_7 &= 13 = 1 \cdot 8 + 5, \\
F_6 &= 8 = 1 \cdot 5 + 3, \\
F_5 &= 5 = 1 \cdot 3 + 2, \\
F_4 &= 3 = 1 \cdot 2 + 1, \\
F_3 &= 2 = 2 \cdot 1 + 0.
\end{aligned}$$

Widzimy, że kolejne reszty z dzielenia to kolejne liczby Fibonacciego. Ma więc miejsce następujący fakt:

Fakt 4. Dla dowolnego $n > 0$ jest $\mathrm{NWD}(F_{n+2}, F_{n+1}) = 1$ i algorytm Euklidesa wykonuje dokładnie n dzieleń.

Twierdzenie (Lamégo). *Jeśli $a > b \geq 0$ oraz $b < F_{k+1}$ to algorytm Euklidesa wykonuje mniej niż k dzieleń.*

Uzasadnienie. Po pierwszym dzieleniu dostajemy dwie liczby, z których większa jest mniejsza niż F_{k+1}. Nawet jeśli odtąd wynikiem z dzielenia zawsze będzie 1, to tych dzieleń będzie co najwyżej $k - 1$ (tyle by było gdybyśmy mieli dwie liczby Fibonacciego). □

Wniosek 1. *Algorytm Euklidesa uruchomiony dla liczb $a > b$, z których większa ma n cyfr działa w czasie $O(n^3)$.*

Uzasadnienie. Załóżmy, że $F_k \leq b < F_{k+1}$. Wtedy wykona się co najwyżej k dzieleń. Tymczasem wiadomo, że $F_k > \phi^{k-2}$ dla $k \geq 3$, gdzie $\phi = \frac{1+\sqrt{5}}{2}$. A więc $\phi^{k-2} < b$. Weźmy logarytm $k - 2 < \log_\phi b = O(lh(b)) = O(n)$ a więc wykona się $O(n)$ dzieleń, każde w czasie $O(n^2)$. Koniec uzasadnienia. □

A więc algorytm Euklidesa działa w czasie wielomianowym! Nasze szacowanie jest jednak niedokładne. Można pokazać (i to elementarnymi metodami), że

Twierdzenie. *Algorytm euklidesa obliczający NWD n-cyfrowej liczby a i m-cyfrowej liczby b ma złożoność $O(nm)$.*