Algorytm Euklidesa

#NWD #NWW
#liczby_względnie_pierwsze
#argumenty_funkcji
#złożoność_obliczeniowa #logarytm

Zajmiemy się teraz znajdowaniem *największego wspólnego dzielnika* dla dwóch liczb naturalnych a i b (ang. **greatest common divisor**, wymawiaj: *grejtest komon diwajzor*). Jest to największa liczba naturalna, przez którą dzielą się bez reszty obydwie liczby. Na przykład największym wspólnym dzielnikiem liczb 75 i 100 jest liczba 25, zaś liczb 27 i 35 – liczba 1. W szkole powszechnej dzielnik ten oznacza się skrótem $NWD(a, b)$, a my będziemy używać raczej angielskiego skrótu $\mathbf{gcd}(a, b)$.

Jeżeli jedna z liczb (a, b) jest równa zero, wtedy największym wspólnym dzielnikiem jest po prostu ta druga liczba. Gorzej, jeśli mamy $a = b = 0$, wtedy dzielnik nie jest określony (może w jego roli wystąpić dowolna dodatnia liczba naturalna). Tak więc w programie musimy zadbać, aby taka sytuacja się nie zdarzyła.*

Pierwsza myśl to sprawdzenie kolejnych liczb od 1 do mniejszej z liczb a i b (dlaczego mniejszej?) i wybranie największej, przez którą dzielą się obydwie liczby. To mogłoby wyglądać tak:

```
int gcd(int a, int b)
{
  int p{ 1 }, m;
  m = min(a, b);
  for(int k{ 2 }; k <= m; k++)
    if(a % k == 0 && b % k == 0)
      p = k;
  return p;
}
```

Zauważmy, że w nagłówku funkcji musieliśmy napisać `int` przed każdym argumentem oddzielnie, podczas gdy w deklaracji zmiennych lokalnych p oraz m nie było to konieczne.

Jeśli żadna z wartości k nie będzie pasować, wtedy pozostanie $p = 1$ i taka wartość zostanie zwrócona przez funkcję `gcd()`. *Notabene*, liczby, których największym wspólnym dzielnikiem jest liczba 1, zwane są *liczbami względnie pierwszymi*. Nie muszą one wcale być liczbami pierwszymi *per se*, przykładem niech będzie wspomniana para 27 i 35.

*Możemy też do kodu funkcji `gcd()` dodać sprawdzenie, czy zachodzi ten przypadek i jakoś sobie z tym poradzić.

Jeśli potencjalne dzielniki (k) będziemy sprawdzać w kolejności rosnącej (jak w powyższym przykładzie), wtedy mamy pewność, że funkcja `gcd()` zwróci rzeczywiście największy możliwy dzielnik (dlaczego?).

Nasza funkcja nie wygląda najgorzej, ale widać na oko, że dla dużych argumentów będzie działać naprawdę powoli. Wystarczy wziąć a i b w okolicach miliarda – i klops! Trzeba zatem postarać się bardziej. I tu przyda się pomysł pochodzący od wspomnianego już autora *Elementów*, czyli Euklidesa.

Zaczniemy od najprostszej jego wersji – z wykorzystaniem odejmowania. Zgodzicie się chyba, że dla dowolnych liczb naturalnych (a, b) zachodzi jeden z poniższych przypadków:

1. $a = b$,
2. $a > b$,
3. $a < b$.

Po prostu innych możliwości nie ma. Jeśli zachodzi pierwszy przypadek, to sprawa jest prosta: największy wspólny dzielnik jest którąkolwiek z tych liczb.

Jeśli zachodzi przypadek drugi, wtedy większą z liczb (czyli a) zastępujemy przez różnicę $a - b$. Dlaczego wolno nam tak zrobić? Otóż jeśli liczby a oraz b mają jakiś wspólny dzielnik (w tym ten największy), jest on również dzielnikiem ich różnicy, prawda? Dla przykładu: różnica liczb parzystych (czyli podzielnych przez 2) jest także parzysta, różnica liczb podzielnych przez 7 jest podzielna przez 7, i tak dalej.

Jeśli zachodzi trzeci przypadek, wtedy liczbę b zastępujemy przez $b - a$. Tak więc, w ten sposób (poza pierwszym przypadkiem, który kończy algorytm) jedna z liczb (a, b) ulega zmniejszeniu.[†] I znów musi zachodzić jeden z powyższych przypadków, zatem albo kończymy obliczanie (jeśli liczby sie zrównały), albo zmniejszamy którąś z liczb. Te kroki powtarzamy tak długo, aż uzyskamy $a = b$.

Prześledźmy zmiany wartości a oraz b na przykładzie liczb 120 oraz 72:

a	b	wykonywana operacja
120	72	a zastępujemy przez $120 - 72 = 48$
48	72	b zastępujemy przez $72 - 48 = 24$
48	24	a zastępujemy przez $48 - 24 = 24$
24	24	koniec algorytmu, wynik = 24

Funkcja realizująca ten algorytm może wyglądać tak:

```
int gcd(int a, int b)
{
  if(a == 0) return b;
  if(b == 0) return a;
  while(a != b)
```

[†]Oczywiście należy uwzględnić przypadek, gdy jedna z liczb jest równa 0, ale zrobimy to już w ostatecznym kodzie.

```
    if(a > b)
      a -= b;

    else
      b -= a;
  return a;
}
```

Operator `-=` chyba nie speszył Czytelnika?

To wygląda dobrze, ale zobaczmy, co się dzieje, gdy liczby (a, b) są względnie pierwsze (i bliskie siebie), na przykład $a = 35$ oraz $b = 36$:

a	b	wykonywana operacja
35	36	b zastępujemy przez $36 - 35 = 1$
35	1	a zastępujemy przez $35 - 1 = 34$
34	1	a zastępujemy przez $34 - 1 = 33$
33	1	a zastępujemy przez $33 - 1 = 32$ itd.
...	...	...
2	1	a zastępujemy przez $2 - 1 = 1$
1	1	koniec algorytmu, wynik = 1

Wredny przykład, *n'est-ce pas*? Wynik wyszedł dobry, ale musieliśmy się przespacerować po wszystkich liczbach naturalnych od początkowej wartości b w dół aż do 1. A co jeśli byłby to miliard liczb? Tak być nie może! Łatwo temu zaradzić, sprawdzając, czy któraś z liczb nie osiągnęła wartości 1 i wtedy od razu zwrócić 1 jako rezultat funkcji:

```
int gcd(int a, int b)
{
  if(a == 0) return b;
  if(b == 0) return a;
  while(a != b)
  {
    if(a == 1 || b == 1) return 1;
    if(a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

Ciut lepiej, ale można jeszcze sprytniej. Wystarczy zauważyć, że jeśli liczby a oraz b mają wspólny dzielnik, to nie tylko ich różnica, ale także reszta z dzielenia jednej liczby przez drugą (a mod b oraz b mod a) także się przez niego dzieli. A to umożliwi nam stworzenie prawdziwie ekspresowej wersji algorytmu. Oto jego opis:

1. Jeśli $b = 0$, wtedy zakończ algorytm – wynikiem jest liczba a.

2. Oblicz $r = a \bmod b$.

3. Zastąp a przez b, zaś za b podstaw r i przejdź do kroku 1.

Prześledźmy jak to działa dla $a = 120$ i $b = 72$:

a	b	r	wykonywana operacja
120	72	48	a zastępujemy przez 72, zaś b przez 48
72	48	24	a zastępujemy przez 48, zaś b przez 24
48	24	0	a zastępujemy przez 24, zaś b przez 0
24	0	–	koniec algorytmu, wynik = 24

Czytelnikowi pozostawiamy sprawdzenie przebiegu algorytmu dla liczb zamienionych miejscami, czyli $a = 72$ oraz $b = 120$.

Na pozór wiele się nie zmieniło, więc prześledźmy jeszcze poprzedni przykład (ten wredny):

a	b	r	wykonywana operacja
35	36	35	a zastępujemy przez 36, zaś b przez 35
36	35	1	a zastępujemy przez 35, zaś b przez 1
35	1	0	a zastępujemy przez 1, zaś b przez 0
35	1	–	koniec algorytmu, wynik = 1

No, tym razem szybko poszło. Można pokazać, że *złożoność obliczeniowa* takiej wersji algorytmu Euklidesa jest *logarytmiczna*, a dokładniej że znalezienie największego wspólnego dzielnika tą metodą wymaga około $\log_2(a + b)$ operacji.[‡] Co to takiego ten logarytm? Mówiąc krótko: jest to wykładnik potęgowy – oznaczmy go przez w – taki, że podstawa logarytmu (tutaj: 2) podniesiona do takiej właśnie potęgi jest równa liczbie w nawiasach: $2^w = a+b$. Czyli jeśli $a+b$ byłoby równe tysiąc, wtedy w byłoby równe około 10 (ponieważ $2^{10} = 1024 \approx 1000$). Dla $a + b$ rzędu miliarda mielibyśmy $w \approx 30$, zatem zysk na czasie jest bardzo duży, bo w poprzednich przypadkach trzeba było się liczyć ze złożonością liniową, czyli z grubsza proporcjonalną do wartości liczb a i b.

Zaimplementujemy ten algorytm w wersji rekurencyjnej:

```
int gcd(int a, int b)
{
  if(b == 0) return a;
  return gcd(b, a % b);
}
```

That's it!

Może trudno w to uwierzyć, ale istnieje jeszcze szybsza wersja tego algorytmu, zwana *binarnym algorytmem Euklidesa* albo *algorytmem Steina*. Jeśli Kocurro nie zapomni sobie, to wrócimy do tematu przy okazji operacji bitowych.

[‡]Być może to wyrażenie trzeba pomnożyć przez pewną stałą liczbową.

No tak, Kocurro prawie zapomniał, że trzeba by jeszcze napisać dwa słowa o *najmniejszej wspólnej wielokrotności* (*NWW*), a po angielsku **lcm**, czyli **least common multiple** (wymawiaj: *list komon multipl*, z długim *i* w pierwszym słowie).

Wykorzystamy następującą tożsamość:

$$\mathbf{lcm}(a,b) = \frac{a \cdot b}{\mathbf{gcd}(a,b)}$$

Tak więc funkcję **lcm()** możemy napisać tak:

```
int lcm(int a, int b)
{
  return a * b / gcd(a, b);
}
```

Przewidujemy jednak pewne problemy: dla dużych argumentów możemy „wyjechać" poza zakres typu `int`, więc może lepiej od razu zdać się na typ `long long`. Wystarczy „zrzutować" jeden z czynników w liczniku tego wzoru i już będziemy bezpieczni:

```
long long lcm(int a, int b)
{
  return (long long)a * b / gcd(a, b);
}
```

To działa, ponieważ działania wykonywane są od lewej do prawej,[§] zatem mnożenie (a potem dzielenie) będzie wykonane już zgodnie z regułami typu `long long`.

[§]Tak się dzieje, gdy operatory występujące w działaniu mają ten sam priorytet, a to zachodzi dla operatorów `*` oraz `/`.