Kubin

O modulo

Modulo przez liczby Mersenne'a

Przy hashowaniu możemy spotkać się z potrzebą szybkiego modulowania przez pewną liczbę. Pozostaje dobór odpowiedniej (najlepiej powinna mieścić się w ~31 bitach oraz być pierwsza). Okazuje się, że liczenie modulo przez liczby Mersenne'a (liczby postaci $2^k - 1$ dla $k \in \mathbb{N}$) można łatwo zapisać w postaci prostych operacji (modulo jest jedną z najwolniejszych operacji arytmetycznych).

Opiszmy funkcję μ:

$$\mu(x) = \left\lfloor \frac{x}{2^k} \right\rfloor + (x \bmod 2^k)$$

Okazuje się, że $\mu(x) \equiv x \pmod{2^k - 1}$.

Dowód

$$\begin{aligned}
\mu(x) &= \left\lfloor \tfrac{x}{2^k} \right\rfloor + (x \bmod 2^k) \\
&= \tfrac{x-(x \bmod 2^k)}{2^k} + (x \bmod 2^k) \\
&= \tfrac{x+(2^k-1)(x \bmod 2^k)}{2^k} \\
2^k\mu(x) &= x + (2^k-1)(x \bmod 2^k) \\
(2^k-1)\mu(x) + \mu(x) &\equiv x + (2^k-1)(x \bmod 2^k) \pmod{2^k - 1} \\
\mu(x) &\equiv x \pmod{2^k - 1}
\end{aligned}$$

$q.e.d.$

Dodatkowo warto zauważyć, że $\mu(x) \le x$, ale $\mu(2^k - 1) = 2^k - 1$. Możemy też wywnioskować, że dowolny x będzie $\le 2^k - 1$ po około $\log_{2^k} x = \frac{\log_2 x}{k}$ iteracjach μ.

Na podstawie tego możemy napisać prosty kod liczący liczby modulo $2^k - 1$, w przykładzie dla $2^{31} - 1$. Ponieważ wykonywaliśmy operacje z potęgami 2, można wykorzystać operacje bitowe.

```
1.   constexpr uint32_t K = 31; constexpr uint64_t P = (1 <<
2.   uint64_t M(uint64_t x)
3.   {
4.       x = (x >> K) + (x & P);
5.       x = (x >> K) + (x & P);
6.       return x == P ? 0 : x;
7.   }
```

- `5.` – dla liczb wymagających 2 iteracji, czyli od około 2^{61}.
- `6.` – ponieważ $\mu(2^k - 1) = 2^k - 1$.

Okazuje się, że $2^{31} - 1$ jest tzw. pierwszą Mersenne'a (czyli liczbą pierwszą, która jest jednocześnie liczbą Mersenne'a), więc będzie odpowiednim modulo do hashowania.

W praktyce ten kod okaże się nawet 4-5 razy szybszy (od zwykłego `x % P`) w rzeczywistym systemie i do 2 razy szybszy w systemie zliczającym instrukcje (np. olimpijskim). W przypadku wolniejszych rozwiązań na hashach może nam to zapewnić dodatkowe punkty.

Warto zauważyć, że przyspieszenie jest znaczne na systemach 32-bitowych, ponieważ modulo jest tam implementowane przez kompilator – procesor nie obsługuje domyślnie liczb 64-bitowych – więc będzie znacznie wolniejsze od zwykłych operacji na liczbach 32-bitowych. Możemy to pokonać wykorzystując właśnie operacje bitowe.

Liczenie $ab \bmod m$ dla $a, b, m \in \mathbb{N}_0 \wedge a, b, m < 2^{63}$

Problem mnożenia dużych liczb z pewnym modulo pojawia się przy zaawansowanych algorytmach teorii liczb (test Millera-Rabina, metoda ρ (rho) Pollarda). To rozwiązanie można też wykorzystać do większych modulo przy hashowaniu - tym samym unikając komplikacji związanych z implementacją podwójnych hashy.

Oczywiście rozważania zaczynamy od spostrzeżenia, że $ab \equiv (a \bmod m)(b \bmod m)$.

Notatka 1: Zakładam, że poniższe funkcje będą przyjmowały tylko liczby `unsigned` (modulo na nich może być szybsze) oraz że będą mniejsze niż m. Oczywiście łatwo można sprowadzić dowolną liczbę do tego przypadku modulując przez m. Uwaga: dla liczb ujemnych operator `%` w C++ może zwrócić liczbę ujemną, w takiej sytuacji należy dodać m. Kod: `x < 0 ? x%m + m : x%m`

Notatka 2: Wszystkie kongruencje będą modulo m.

Notatka 3: Okazuje się, że najszybszym sposobem na modulowanie liczb 64-bitowych jest `a >= m ? a % m : a`. Warto o tym pamiętać, jeżeli poniższym funkcjom przekazujemy liczby z zakresu innego niż $[0, m)$. Razem z notatką 1 można napisać:

```
uint64_t umod(int64_t x, uint64_t m)
{
    if(x < 0)
        return x%int64_t(m) + m;
    else
        return uint64_t(x) >= m ? uint64_t(x) % m : x;
}
```

Manipuluję tutaj typami, ponieważ modulo na `unsigned` może być szybsze, a cast prawie nic nie kosztuje w tym wypadku. Można sobie po prostu uprościć i korzystać tylko z `int64_t`.

Notatka 4: Przy arytmetyce modularnej warto jak najczęściej zmienną, przez którą modulujemy, oznaczać jako `constexpr`. Dzięki temu kompilator będzie mógł czynić swoją magię optymalizacyjną (np. czasami umie policzyć odwrotność modulo i zamiast liczyć modulo będzie mnożył, co będzie szybsze). Oczywiście możemy to zrobić tylko jeżeli liczba jest z góry znana.

Przede wszystkim sposób najprostszy (`(a*b)%m`) odpada, bo zajdzie overflow i zamiast $ab \bmod m$ policzymy $(ab \bmod 2^{64}) \bmod m$.

Typ 128-bitowy

Najprostszą znaną mi metodą jest udostępniana w 64-bitowym gcc/g++ implementacja liczb 128-bitowych – `__int128` – i prawdopodobnie wykorzystanie tego typu okaże się najlepszym wyjściem. W 32-bitowym systemie musimy sobie radzić inaczej.

Metoda rosyjskich chłopów

Najpopularniejszym sposobem jest *metoda rosyjskich chłopów*, działająca w $O(\log b)$. Opiera się ona na prostym lemacie: $ab \equiv 2a \left\lfloor \frac{b}{2} \right\rfloor + (b \bmod 2)a$. Implementacja przypomina szybkie potęgowanie.

```
 1. uint64_t russian_peasant_method(uint64_t a, uint64_t b,
 2. {
 3.     uint64_t r = 0;
 4.     while(b)
 5.     {
 6.         if(b % 2) r += a, r %= m;
 7.         b /= 2;
 8.         a *= 2; a %= m;
 9.     }
10.     return r;
11. }
```

Ograniczamy m od góry liczbą 2^{63}, ponieważ w każdej iteracji pętli mnożymy przez 2 liczbę a (`8.`), która jest mniejsza od m. Jeżeli $a \geq 2^{63}$, zajdzie overflow.

Aproksymacja

▸ Stary opis

Okazuje się jednak, że istnieje sposób lepszy, działający w $O(1)$ i znacznie szybszy. Aby polepszyć swoją pozycję i zmieścić się w limitach będziemy musieli trochę oszukać system.

Co tak naprawdę chcemy policzyć licząc modulo pewnej liczby? Należy przypomnieć sobie definicję kongruencji. Licząc $x \bmod m$ szukamy $q, r \in \mathbb{N}_0$ spełniających:

$$x = qm + r$$

(q jak *quotient*, r jak *remainder*)

Przy czym szukamy jak najmniejszego nieujemnego r (wszystkie możliwe różnią się o wielokrotności m). Wtedy q wynosi $\left\lfloor \frac{x}{m} \right\rfloor$. Możemy więc przekształcić równanie i liczyć $r = x - qm = x - \left\lfloor \frac{x}{m} \right\rfloor m$.

Chcemy skorzystać z tej definicji. Oczywiście, nie ma szans aby precyzyjnie policzyć $\frac{ab}{m}$, bo z założenia nie mamy typu który mógłby reprezentować $ab < 2^{126}$. Zamiast tego skorzystamy z liczb zmiennoprzecinkowych i aproksymujemy ile wynosi ten iloraz. Otrzymane w ten sposób r' będzie się różniło o pewną (małą) wielokrotność m (a przynajmniej tak zakładamy), więc aby odzyskać wynik policzymy $r = r' \bmod m$.

A oto implementacja:

```
1.  uint64_t approximation_mm(uint64_t a, uint64_t b, uint64
2.  {
3.      uint64_t q = (long double)(a) * b / m;
4.      int64_t r = (int64_t)(a * b - q * m) % (int64_t)m;
5.      return r < 0 ? r + m : r;
6.  }
```

- `3.`
  - Korzystamy z `long double`, ponieważ ten typ ma 64 bity mantysy, która jest potrzebna aby zachować jak najwięcej precyzji. `double` ma za małą precyzję (53 bity mantysy) i nie jest dużo szybszy w tym przypadku.
  - Wystarczy castować jedynie `a`, cast pozostałych typów jest automatyczny (*type promotion*).
  - Są co najmniej dwie sensowne kolejności wykonania tej operacji – `a * b / m` oraz `a / m * b`. Okazuje się że dla dużych zakresów pierwsza z tych kolejności ma większą precyzję (testy wykonane w Pythonie, który korzysta z typu analogicznego do `double` (z IEEE 754)).
  - Umyślnie unikamy wywołania `floor`, które jest wolne. Zamiast tego pozwalamy aby wartość uległa *truncation*, czyli 'obcięciu' części po przecinku. Teoretycznie jest to przypadek *Undefined Behaviour* (UB), więc warto sprawdzić czy wszystko działa poprawnie na danym systemie (głównie będzie to zależało od

kompilatora i procesora).

- `4.` – Wynik liczymy przy założeniu że $-2^{63} \leq r' < 2^{63}$, tzn. mieści się w zakresie `int64_t`. O ile to zachodzi na pewno dostaniemy poprawnego kandydata spełniającego równanie. Sprowadzamy r' do r licząc modulo kandydata. Tutaj pojawia się główne ograniczenie tej metody – nie może dojść do overflow.
- `5.` – tak jak wcześniej: dla liczb ujemnych `%` może zwrócić liczbę ujemną.

Wraz ze wzrostem `m` rośnie szansa na overflow. Jednak można założyć że metoda będzie działać nawet dla $m < 2^{63}$. Warto jednak zawsze to przetestować (można porównywać z działaniem metody ruskich chłopów). Sposób na pewno nie zadziała gdy wynik nie mieści się w `int64_t`. Nie unikniemy tego ograniczenia nie czyniąc założeń na temat błędu zaokrąglenia – można zawsze dodawać w nawiasie `C*m` dla pewnej stałej `C`, zakładając że $q' \geq C$ (oznaczmy zaokrąglenie na q jako q') – lub tracąc na wydajności – zamieniając modulo na pętlę dodającą `m` w odpowiedni sposób).

Bibliografia

- @ https://ariya.io/2007/02/modulus-with-mersenne-prime
- @ https://codeforces.com/blog/entry/17281?#comment-221520