Ezoteryczne Kartki
Kwiatki teorii liczb

Jakub Bachurski

wersja 1.1.0

Wstęp

Tym razem porozważamy sobie różne algorytmy teorioliczbowe, w których w ciekawy sposób wykorzystamy techniki z różnych części krainy algorytmiki.

1 Binarne GCD

Zaczniemy od prostego i przyjemnego algorytmu do liczenia największego wspólnego dzielnika. „Co, Euklides, panie autorze? Serio?" – tak pewnie byście odpowiedzieli, jako ludzie, których oszukiwano przez całe życie. Bo przecież nie istnieje żaden inny sensowny algorytm do liczenia NWD...?
Jednak nie jest to prawda: okazuje się, że istnieje algorytm liczenia NWD, który nie potrzebuje modulo, aby osiągnąć logarytmiczną złożoność czasową. Opiera się na prostym pomyśle: rozważamy 2 jako wspólny dzielnik liczb a i b i wyciągamy z tego wnioski, następnie aplikujemy tę samą metodę rekurencyjnie, a jeżeli nie wiemy co zrobić, to korzystamy z tożsamości $\gcd(a, b) = \gcd(b - a, a)$.

Dokładniej, rozpatrujemy przypadki względem reszty z dzielenia przez 2 tych liczb (przy czym wpierw sprawdzamy przypadek brzegowy $\gcd(0, b) = b$:

1. $a \equiv b \equiv 0$: Z pewnością $\gcd(a, b) = 2\gcd(a/2, b/2)$.
2. $a \not\equiv b$: Tym razem, 2 na pewno nie jest wspólnym dzielnikiem tych liczb: zatem $\gcd(a, b) = \gcd(a/2, b)$ lub $\gcd(a, b) = \gcd(a, b/2)$.
3. $a \equiv b \equiv 1$: Tutaj trudno cokolwiek wywnioskować, więc korzystamy z $\gcd(a, b) = gcd(b - a, a)$ (przy tym pamiętamy, żeby utrzymywać $a \leqslant b$).

Jaką złożoność ma to podejście? Zauważmy, że przypadki 1 i 2 muszą zredukować liczby dwukrotnie, więc wydarzą się co najwyżej $O(\log a + \log b)$ razy. Co więcej, za każdym przypadkiem typu 3 natychmiast idzie przypadek 2, bo $b - a \equiv 0$, zatem on również musi się wykonać co najwyżej logarytmicznie wiele razy.

To podejście można znacznie zoptymalizować za pomocą operacji bitowych (dokładniej `ctz`), aby wielokrotnie wykonać przebieg przypadku 2. Poniżej także prosta rekurencyjna implementacja.

```
unsigned rgcd2(unsigned a, unsigned b)
{
    if(a > b) swap(a, b);
    if(a == 0)
        return b;
    else if(a % 2 == 0 and b % 2 == 0)
        return 2 * rgcd2(a / 2, b / 2);
    else if(a % 2 == 0 and b % 2 == 1)
        return rgcd2(a / 2, b);
    else if(a % 2 == 1 and b % 2 == 0)
        return rgcd2(a, b / 2);
    else
        return rgcd2(b - a, a);
}

unsigned gcd2(unsigned a, unsigned b)
{
    if(a > b)
        swap(a, b);
    if(not a)
        return b;
    auto s = __builtin_ctz(a | b);
    a >>= s;
    do {
        b >>= __builtin_ctz(b);
        if(a > b)
            swap(a, b);
        b -= a;
    } while(a);
    return b << s;
}
```

Po co komu taki algorytm? Dzięki niemu wystarczy nam $O(\log ab)$ operacji liczenia reszty modulo 2, dzielenia przez 2 i odejmowania, zamiast trudnej operacji modulo. Może to się przydać w skrajnych przypadkach, gdy potrzebujemy NWD z utrudnionym dostępem do operacji (np. pisząc własną arytmetykę). Jednak w praktyce nie warto pisać tego algorytmu w klasycznym przypadku – według moich testów przyspieszenie jest rzędu 15%, przy czym kluczowe jest wykorzystanie implementacji z operacjami bitowymi.

2 Trik z pierwiastkiem sześciennym

Fakt. Czynników pierwszych liczby n większych od $\sqrt[k]{n}$ jest co najwyżej $k-1$. W przeciwnym wypadku iloczyn czynników pierwszych byłby większy od n.

Ze szczególnego przypadku faktu korzystamy licząc faktoryzację $O(\sqrt{n})$, ponieważ po rozważeniu wszystkich czynników $p_1, ..., p_e \leqslant \sqrt{n}$ liczba $m = \frac{n}{p_1 \dots p_e}$

albo jest równa 1, albo jest pierwsza (bo ma co najwyżej 1 czynnik pierwszy). Pomysł ten możemy rozszerzyć, sprawdzając czynniki do pierwiastka sześciennego. Otrzymamy wtedy parę możliwych postaci pozostałej liczby m do rozważenia:

0. $m = 1$.
1. $m = p$ jest pierwsze.
2. a) $m = p^2$ jest kwadratem liczby pierwszej, lub b) $m = pq$ jest iloczynem dwóch liczb pierwszych.

W niektórych zadaniach rozważenie tych przypadków może nas uchronić od pisania trudniejszych algorytmów. Popatrzmy dla przykładu na zadanie *Kwadraty liczb naturalnych* z finału XIII OIG:

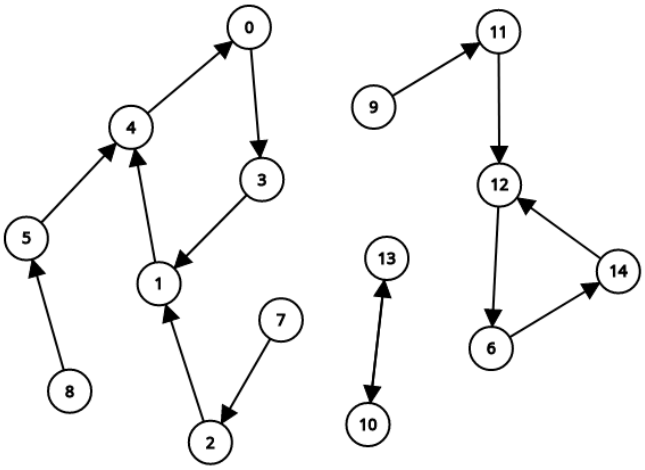
Treść Mamy dane $n \leqslant 3 \cdot 10^5$ liczb całkowitych a_i ($1 \leqslant a_i \leqslant 10^9$). Policz liczbę takich par (a_i, a_j), że $a_i \cdot a_j$ jest kwadratem liczby naturalnej.

Rozwiązanie Przyda nam się spostrzeżenie, że jeżeli liczba jest kwadratem liczby naturalnej, to wszystkie czynniki pierwsze w jej faktoryzacji mają parzysty wykładnik. Zależy nam zatem na poznaniu czynników liczby, które mają nieparzysty wykładnik – jest to „część nieparzysta" (bo „część parzysta" nic nie zmienia). Zauważmy, że iloczyn dwóch liczb jest kwadratem liczby naturalnej, jeżeli mają takie same części nieparzyste (bo po przemnożeniu dodamy do siebie wykładniki, zatem część nieparzysta zniknie).

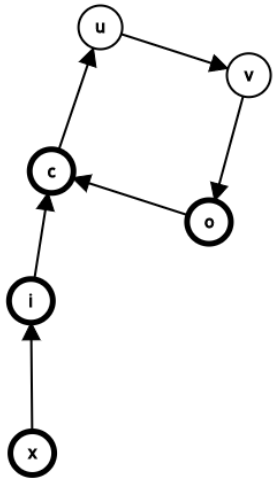
Pozostaje znaleźć część nieparzystą każdej z liczb. Skorzystajmy z triku z pierwiastkiem sześciennym. Pozostaje rozważyć postaci p, p^2, pq. Zarówno pierwsza i trzecia trafią do części nieparzystej, ale druga ma parzysty wykładnik, więc trzeba ją jakoś wykryć. Możemy to zrobić na przykład preprocesując wszystkie kwadraty do 10^9. Zliczanie par o takiej samej części nieparzystej to formalność.

3 Grafy funkcyjne

Zrobimy chwilową odskocznię do tematu bardziej związanego z algorytmami randomizowanymi niźli teorią liczb, ale jest on kluczowy w bardzo użytecznym algorytmie faktoryzacji. Wprowadźmy pojęcie *grafu funkcyjnego*: jest to taki graf skierowany, że z każdego wierzchołka wychodzi dokładnie jedna krawędź. Nazwa pochodzi od faktu, że możemy go opisać za pomocą funkcji f, takiej że dla każdego wierzchołka x istnieje krawędź $x \to f(x)$. Przypomnijmy, że posiada on bardzo przydatną własność: wychodząc ścieżką z dowolnego wierzchołka, prędzej czy później trafimy na cykl. Co ważne, w losowym grafie funkcyjnym (f jest losową permutacją wierzchołków) taki cykl napotkamy po $O(\sqrt{n})$ krawędziach, co wynika z paradoksu dnia urodzin. Na obrazku graf funkcyjny wygląda mniej więcej tak:



Spróbujmy wykorzystać własność krótkiego czasu oczekiwania na cykl. Skąd tak na prawdę wiemy, że napotkaliśmy cykl, z perspektywy pewnego początkowego wierzchołka x? Jest to równoważne z faktem, że weszliśmy do pewnego wierzchołka c dwukrotnie, zatem istnieją wierzchołki i, o, że $c = f(i) = f(o)$. Nawiasem mówiąc, powstały kształt ścieżki nazywamy *rho* od greckiej litery ρ.



Możemy zatem skorzystać z tego, że w złożoności pierwiastkowej od liczby wierzchołków możemy znaleźć wierzchołki i, o, że $f(i) = f(o)$, dla wybranej przez nas funkcji f (o ile jest ona dosyć losowa) oraz wierzchołka początkowego x. Oczywiście zbiór wierzchołków grafu wybieramy sami.

3.1 Poszukiwanie kolizji hashy

Szukanie powtarzających się wartości funkcji dla dwóch argumentów przywodzi nam na myśl hashe. Skonstruujmy graf funkcyjny na zbiorze wartości hashy $V = \{0, 1, ..., n-1\}$ funkcji hashującej $h : A \to V$ (gdzie A to hashowane obiekty).

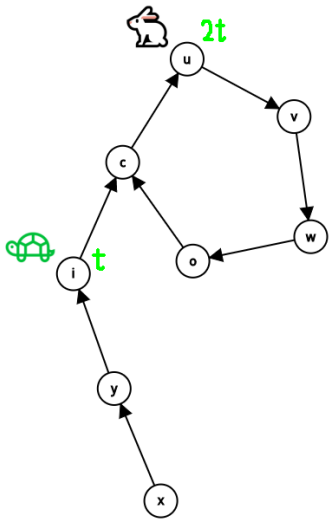
Chcemy teraz znaleźć obiekty i, o, że $h(i) = h(o)$. Nie możemy jeszcze jednak tego zrobić opisanym wyżej sposobem, bo poszukujemy kolizji argumentów ze zbioru A, a nie V. Potrzebujemy zatem funkcji $g : V \to A$, która przemapuje nam wierzchołki V na wymagane obiekty A w deterministyczny, lecz w miarę losowy, sposób. Aby z hasha wygenerować pewien losowy obiekt (co ważne, nie musi on mieć takiego hasha) możemy wykorzystać generator, którego ziarnem jest ten hash.

W ten sposób możemy wreszcie skonstruować funkcję $f(x) = h(g(x)) : V \to V$, która cechuje się w miarę porządną losowością. Pozostaje znaleźć cykl w $O(\sqrt{n})$. Otrzymane hashe x, y (dla których mamy $f(x) = f(y)$) wrzucamy do naszej funkcji g, aby otrzymać obiekty $i = g(x), o = g(y)$, które dają kolizję.

Ale tak właściwie, to po co tyle zachodu, jeżeli w gruncie rzeczy i tak potrzebujemy zużycia pamięci $O(\sqrt{n})$, tak jak w najprostszym podejściu (generujemy słowa i wrzucamy do zbioru, dopóki przypadkiem nie nadarzy się kolizja)? Odpowiedź jest prosta: nie potrzebujemy aż tyle pamięci.

3.2 Szukanie cykli: algorytm Floyda

Do wyszukiwania cyklu w grafie funkcyjnym opracowano specjalne algorytmy[1]. Rozważymy algorytm Floyda (żółwia i zająca), działający w $O(\lambda+\mu)$, gdzie λ to długość cyklu, na który natrafimy, a μ to długość „ogonka" (ścieżki do pierwszego wierzchołka leżącego na cyklu od wierzchołka początkowego). Nietrudno się domyślić, że w losowym grafie jest to złożoność $O(\sqrt{n})$. Co ważne, ten algorytm ma **stałe** zużycie pamięci.



Pomysł jest prosty: będziemy chodzić dwoma wskaźnikami: żółwiem oraz zającem, przy czym zając w każdej iteracji skacze o dwie krawędzie, a żółw o jedną. Okaże się, że z pomocą tego łatwo odtworzyć μ i λ, a co za tym idzie, również potrzebne nam wierzchołki i oraz o (są to wierzchołki po odpowiednio $\mu - 1$ i $\mu + \lambda - 1$ skokach).

[1] `https://en.wikipedia.org/wiki/Cycle_detection`

Zapiszmy warunek na spotkanie się żółwia i zająca w którymś wierzchołku. Oczywistym jest, że nie spotkają się na ogonku, więc na pewno żółw musi najpierw dojść do początku cyklu c (zajmie mu to μ skoków). Łatwo zauważyć, że wtedy zając będzie się znajdował na pozycji o μ skoków od c – ponieważ mamy do czynienia z cyklem, możemy to zastąpić $\mu \bmod \lambda$. Dla ułatwienia, indeksujmy wierzchołki względem odległości (w skokach) od wierzchołka c. Korzystając z arytmetyki modularnej, zapiszmy warunek na spotkanie w s:

$$s \equiv 2s + \mu \pmod{\lambda}$$

To znaczy, po s krokach żółw ma być w tym samym miejscu co zając, który zaczął o μ dalej. Rozwiązanie to $s \equiv -\mu \pmod{\lambda}$, zatem wykrywając moment, gdy wskaźniki spotkają się, trafimy na wierzchołek odległy o $k\lambda - \mu$ (dla pewnego $k > 0$) od początkowego wierzchołka x. Możemy teraz policzyć λ, zataczając pętlę żółwiem i wracając do zająca, ponieważ zajmie nam to dokładnie λ kroków.

Kluczowa obserwacja, aby dokończyć algorytm: idąc zarówno z s jak i z x pojedynczymi skokami, spotkamy się w c po μ krokach. Wystarczy zatem stworzyć odpowiedni licznik, i otrzymujemy drugą z pożądanych wartości.

W ten sposób stworzyliśmy bardzo ogólną wersję tego algorytmu. Pomysł jest elastyczny i możemy osiągnąć to samo na różne sposoby, czasem wykorzystując mniej wywołań f.

3.3 Faktoryzacja z grafami: algorytm rho Pollarda

Wróćmy teraz do świata teorii liczb i spróbujmy wykorzystać grafy funkcyjne w rozwiązaniu elementarnego problemu faktoryzacji liczb naturalnych, czyli ich rozkładu na czynniki pierwsze.

Powiedzmy, że chcemy faktoryzować $n = pq$, przy czym $1 < p \leqslant q$ są nieznanymi liczbami całkowitymi, i skonstruujmy graf funkcyjny z wierzchołkami $V = \{0, 1, ..., n-1\}$. Do tego dobierzmy funkcję $f : V \to V$. W tym algorytmie zwykle wykorzystuje się $f(x) = x^2 + c \mod n$ dla $c \notin \{0, -2\}$.

Najważniejszy pomysł: **Rozważmy** cykl w innym grafie funkcyjnym z wierzchołkami $V' = \{0, 1, ..., p-1\}$ oraz funkcją $g(x) = f(x) \bmod p$ (początkowy graf zmodulowany przez p). Algorytm Floyda w tym grafie z pewnością zatrzyma się (gdy żółw i zając spotkają się) w pewnym wierzchołku t po około $O(\sqrt{p})$ iteracjach. Ponieważ nie wiemy, jak wygląda ten graf, możemy uruchomić algorytm Floyda w oryginalnym grafie i zastanowić się, na jakiej pozycji znajdują się żółw i zając w tym grafie, jeżeli właśnie spotkały się w grafie modulo p. Oznacza to przecież, że $t = x \bmod p = y \bmod p$, zatem $x \equiv y \pmod{p}$, tudzież $p \mid x - y$.

Zastanówmy się, co daje nam to, że $p \mid x - y$. Co jeszcze wiemy o p? Jest ono z definicji dzielnikiem n, zatem $p \mid n$ i p jest wspólnym dzielnikiem n i $x - y$. Co za tym idzie, największy wspólny dzielnik n i $x - y$ jest większy od 1 i tym samym otrzymujemy poszukiwaną wartość (jeden z dzielników n) – możliwe, że ta wartość nie będzie dokładnie równa p. Zwróćmy uwagę, że jeżeli $x = y$, to znaczy, że niedługo zaczniemy się zapętlać w grafie funkcji f i najwyższy czas skonstruować nowy graf.

W takim razie, możemy po prostu sprawdzać wartość $\gcd(|x-y|, n)$, bo jeżeli cykl w tym nieistniejącym grafie jest, to ta wartość jest większa od 1. W ten sposób możemy otrzymać poszukiwaną wartość: pewien dzielnik n. Po co nam te wymyślone grafy? Dzięki ich istnieniu wiemy, że do cyklu dojdzie całkiem szybko: już po $O(\sqrt{p}) = \boldsymbol{O(\sqrt[4]{n})}$ krokach algorytmu Floyda.

Wszystkie te rozważania doprowadzają nas do wybitnie prostego pseudokodu algorytmu, nazywanego algorytmem rho Pollarda:

```
def pollard_rho(n, c=2):
    def f(x):
        return (x**2 + c) % n
    x = y = 1 # wskaźniki
    d = 1
    while d == 1:
        x = f(x)
        y = f(f(y))
        d = gcd(|x - y|, n)
    if d != n: # cykl w f, mamy czynnik
        return d
    else: # próbujemy znowu
        return pollard_rho(n, c+1)
```

3.3.1 Kwestie techniczne implementacji rho Pollarda

- Istnieje odmiana rho Pollarda, która wykorzystuje inny algorytm wyszukiwania cyklu: metodę Brenta. Jest ona trochę szybsza, ze względu na mniej wywołań f.
- Aby liczyć standardowe f dla 64-bitowych n pojawia się potrzeba mnożenia modulo liczb 64-bitowych. Należy pamiętać o metodzie rosyjskich chłopów bądź innym triku.
- Ten algorytm najlepiej wywołać dopiero po pozbyciu się małych czynników (np. tak do $\sqrt[4]{n}$, albo najlepiej do stałej z preprocesowaną listą liczb pierwszych).
- Faktyczna złożoność tego algorytmu jest rzędu $O(\sqrt[4]{n}\log n)$.

4 Sito liniowe

Z czego składa się lista czynników pierwszych liczby? Z któregoś z czynników i reszty. Możemy na przykład powiedzieć, że rozkład liczby n składa się z najmniejszego czynnika p i pozostałych czynników liczby $\frac{n}{p}$, które możemy otrzymać rekurencyjnie (również szukając najmniejszego czynnika pierwszego). Tak się składa, że najmniejsze czynniki pierwsze możemy szybko policzyć hurtowo

dla liczb od 1 do n za pomocą sita Erastotenesa w $O(n \log \log n)$. W ten sposób rozkład liczby równej co najwyżej n możemy przeprowadzić w optymalnym czasie $O(\log n)$, sprawdzając kolejne minimalne czynniki.

Spróbujmy opracować algorytm sita, który odwiedzi każdą liczbę dokładnie raz. Zrobimy to rozpatrując właśnie minimalny czynnik pierwszy, ponieważ każda liczba może być jednoznacznie zapisana jako $n = p \cdot \frac{n}{p}$, gdzie p to minimalny czynnik pierwszy n. Zatem odwiedzając komórkę m chcemy odwiedzić wszystkie liczby postaci qm, przy czy q jest minimalnym czynnikiem pierwszym qm. W ten sposób rozważymy każdą z liczb co najwyżej raz. Opisany algorytm nazywany jest sitem liniowym, bo działa w złożoności $O(n)$.

- Dla każdej liczby x od 1 do n:
  - Jeżeli nie wskazaliśmy najmniejszego czynnika pierwszego x, to x jest pierwsze.
  - Dla każdej liczby pierwszej p mniejszej bądź równej najmniejszemu czynnikowi pierwszemu x:
    - Ustawmy najmniejszy czynnik pierwszy px na p.

5 Algorytm Millera-Rabina

Algorytm Millera-Rabina umie stwierdzić, czy dana liczba n jest pierwsza w czasie $O(w \log n)$, gdzie w to liczba zastosowanych świadków. Jest to algorytm randomizowany, i prawdopodobieństwo podania poprawnej odpowiedzi rośnie wraz w. Wszystko, co warto o nim wiedzieć, można znaleźć na Wikipedii:

`https://pl.wikipedia.org/wiki/Test_Millera-Rabina`

Warto zwrócić uwagę, że istnieją zbiory świadków, dla których udowodniono, że działają dla wszystkich liczb z danego zakresu. Takie listy można znaleźć w Internecie, na przykład tutaj:

`http://miller-rabin.appspot.com/`

6 Logarytm dyskretny

Zadanie „Logarytm”

Innym ciekawym problemem teorioliczbowym, gdzie możemy zastosować klasyczną technikę algorytmiczną, jest problem logarytmu dyskretnego. Polega on na znalezieniu liczby x, takiej że (a, b, n to liczby całkowite, a i n są względnie pierwsze[2]):

$$a^x \equiv b \pmod{n}$$

[2]Metodę można tak zmodyfikować, aby działała bez założenia, że są względnie pierwsze, ale jest to nietrywialne.

Możemy tutaj zastosować pierwiastki wraz z tak zwaną techniką *baby-step giant-step* (można się też tutaj dopatrywać *meet in the middle*). Pomysł: Zapiszemy sobie potęgi a z wykładnikami w malutkich odstępach, a potem będziemy robić duże skoki i sprawdzać, czy przypadkiem wynik nie jest postaci takiego dużego skoku i któregoś z malutkich.

A formalnie:

$$a^x \equiv a^{ik-j} \equiv \left(a^k\right)^i a^{-j} \equiv b \iff \left(a^k\right)^i \equiv ba^j$$

Nietrudno zauważyć, że każdy x można zapisać w postaci $ik-j$. Korzystamy tutaj z pewnego parametru k, którego wartość ustalimy.

A teraz zapisujemy wszystkie możliwe wartości ba^j i sprawdzamy, czy któraś z $\left(a^k\right)^i$ występuje wśród nich. Możemy to zrobić w złożoności około $O\left(k + \frac{n}{k}\right)$ (z dokładnością do złożoności szukania powtórzenia), zatem optymalny parametr to $k = \sqrt{n}$, co daje złożoność $O(\sqrt{n})$.

Ciekawostka: ten problem można też rozwiązać grafami funkcyjnymi. Takie podejście opisane jest tutaj: `https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm_for_logarithms`.