Алгоритми в теория на числата

Основни понятия

- С $x|n$ означаваме "x е делител на n".
- С $a^b = c$ означаваме $\underbrace{a \times a \times a \cdots a}_{b \text{ пъти}} = c$.
- Функцията $\sqrt[b]{c}$ връща стойността на a в горното уравнение или отговаря на въпроса "Кое число трябва да вдигна на степен b, за да получа c ?"
- Функцията $\log_a c$ връща стойността на b в горното уравнение или отговаря на въпроса "На коя степен трябва да вдигна a, за да получа c ?"
- "Канонично разлагане на число" x ще наричаме представянето му като произведение на прости числа. Пример: $84 = 2^2 \times 3^1 \times 5^0 \times 7^1$. Формално ще означаваме $x = p_1^{\alpha_1} \times p_2^{\alpha_2} \cdots p_k^{\alpha_k}$, където p_i са простите множители, а α_i са степенните показатели.
- Най-малкото общо кратно на a и b ще означаваме $\text{lcm}(a, b)$ от английски - *least common multiple*.
- Най-големият общ делител на a и b ще означаваме $\gcd(a, b)$ от английски - *greatest common divisor*.

Обхождане на делителите на n в $O(\sqrt{n})$ време

1.2.1 Всички делители

За да надминем наивната $O(n)$ идея за намиране на делителите на n, ще трябва да се възползваме от някакво математическо свойство. Нека да сме намерили x, такова че $x|n$. Друг запис на това твърдение е следния: съществува y, такова че $xy = n$. Това обаче автоматично ни намира и втори делител на n (освен когато $x = y$, тоест $x^2 = n$).

За да се възползваме напълно, трябва да забележим, че за $x < x_0$, съответното $y_0 < y$ (увеличавайки първия множител, трябва да намалим втория). Това дава достатъчна мотивация да напишем решение, което ще се окаже, че работи с правилната сложност, но нека все пак я докажем (в последствие ще можем да напишем и по-изчистен код).

Трябва да се убедим, че по-малкия множител ще бъде винаги по-малък или равен на $\sqrt{n}$. Да допуснем противното – съществуват $x, y > \sqrt{n}$ или $x, y < \sqrt{n}$, такива че $xy = n$. Ще разпишем първия случай, а вторият ще оставим за упражнение (решава се аналогично). Нека за удобство $x = x' + \sqrt{n}$ и $y = y' + \sqrt{n}$ за x', y' естествени числа. Тогава обаче трябва да е вярно следното:

$$xy = (x' + \sqrt{n})(y' + \sqrt{n}) = x'y' + (x' + y')\sqrt{n} + n > n$$

което е видимо противоречие. Следователно не е възможно допускането, че $x, y > \sqrt{n}$ – трябва единият множител да изпълнява $\leq \sqrt{n}$.

Възползвайки се от това ограничение финалния код изглежда по следния начин:

```
for(int x = 1; x * x <= n; x ++){
    if(n % x == 0){
        cout << x << " is divisor" << endl;
        if(x != n / x) cout << n / x << " is divisor" << endl;
    }
}
```

Забележете, че не пишем условието директно като *x <= sqrt(n)* – функцията *sqrt(n)* би бавила в случая и ще я заменим с еквивалентното *x * x <= n*. Освен това проверяваме, че

$x \neq y$ - в противен случай може да получим дубликиране на един делител при n - точен квадрат.

- **Задача:** НОИ3 2022 E6.abundant
- **Задача:** ПТИ 2023 E1.trivia

1.2.2 Само прости делители

Като заключение на този алгоритъм ще отбележим, че можем да го ползваме и за да обиколим само простите делители на числото / за да намерим каноничното разлагане на числото - когато срещнем даден делител, делим променливата n на него, докато е възможно. Така никога няма да срещнем съставен делител - той би се състоял от по-малки, прости делители, които вече са "премахнати" от стойността на n, съответно променливата n вече не се дели нито на тях, нито на съставния делител. При такава употреба на алгоритъма трябва също накрая да проверим, че наистина в n не са останали други делители - конкретно, може да остане един "голям" прост делител, който for-цикъла не е достигнал, за да премахне.

Да разгледаме работата на алгоритъма при $n = 24$. Тогава за $x = 2$, ще разделим три пъти и ще останем с $n = 3$. В последствие цикълът ще спре, защото вече условието $x \times x \leq n$ всъщност проверява дали $2 \times 2 \leq 3$. Няма да се случи n накрая да е нещо различно от 1 или просто число, защото в противен случай условието на for-цикъла, все още ще е изпълнено. Също не е опция да заменим условието с нещо като $x \leq n$ - тогава при вход n =голямо просто число (примерно $10^9 + 7$) ще направим твърде много проверки и ще ударим time limit-а.

```
for(int x = 2; x * x <= n; x ++){
    if(n % x == 0){
        cout << x << " is divisor" << endl;
        while(n % x == 0) n /= x;
    }
}
if(n > 1){
    cout << n << " is divisor" << endl;
}
```

Чести ползи на решето на Ератостен

Тук ще генерализираме идеята на решето на Ератостен, отвъд намиране на простите числа в интервал $[1, n]$. С общи думи, чрез решето на Ератостен можем за всяко число в интервала $[1, n]$ да съберем информация, зависеща от делителите му със сложност $O(n \log n)$. Лесно може да го оптимизираме до $O(n \log \log n)$, а в допълнение съществува и модификация, която води до сложност $O(n)$. Последното ще пропуснем в този текст, а за по-добрата сложност ще има допълнение чак накрая.

1.3.1 Сума/брой на делителите

За по-нагледно ще отработим два примера за информация, която може да събираме: "сумата/броя на делителите на число" и "минималния прост делител на число" за всички числа в интервала $[1, n]$.

С наличните ни знания, сумата/броя на делителите звучи като задача, която бихме решили с горния алгоритъм за обикаляне на делителите със сложност $O(\sqrt{n})$. Понеже обаче трябва да изпълним това за всички числа в интервала, ще получим обща сложност $O(n\sqrt{n})$, която може

да се подобри. Неформално, пропуск в сегашното ни решение е, че всяко число отделно проверява $O(\sqrt{n})$ стойности дали са делители или не - съответно има някакви неуспешни проверки, които можем да избегнем.

Ще обърнем начина, по който гледаме задачата - вместо всяко число да си търси делителите, всяко число ще си търси кратните. Така вече няма да има неуспешни проверки, защото няма всъщност какво да се проверява - за дадено число x лесно можем да обходим кратните му: $x, 2x, 3x \cdots$.

```
for(int d = 1; d <= n; d ++){
    for(int k = d; k <= n; k += d){
        cnt_div[k] ++;
        sum_div[k] += d;
    }
}
```

Първият цикъл избира едно число, което ще се явява делител (съответно го означаваме d). Вторият цикъл обикаля числата, които имат d за делител (съответно са кратни и ги означаваме с k). Коректността на алгоритъма е ясна, остава да изясним, защо наистина сложността е по-добра?

Грубо казано, алгоритъмът работи за $O(\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{n})$ време. Ще извадим общ множител n и ще правим анализ само върху останалите дроби.

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

Ще "заключим" горната сума между две суми – една по-малка (ще означаваме s) и една по-голяма (съответно S). За тях по-лесно ще можем да обосновем. Ще получим по-малка (а по-късно и по-голямата) сума, като заменим някои от елементите в сумата със стриктно по-малки. Нека всяка дроб $\frac{1}{x}$ да заменим с $\frac{1}{2^k}$, където k е най-малката възможна степен, такава че $2^k > x$. Тогава малката сума s ще бъде:

$$s = \frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \cdots + \frac{1}{n}$$

Ще групираме числата, така че да получаваме $\frac{1}{2}$ във всяка група.

$$s = \frac{1}{1} + \left(\frac{1}{2}\right) + \left(\frac{1}{4} + \frac{1}{4}\right) + \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right) + \cdots + \frac{1}{n}$$

Така, грубо казано, $s = 1 + \frac{\log n}{2}$. Неформално, логаритъмът се появи тук, защото всяка следваща група, макар пак да има сума $\frac{1}{2}$, е два пъти по-дълга. Ако не е ясно, препоръчвам да си го разпишете.

Аналогични сметки ни показват, че $S = 1 + \log n$. Получаваме, че s и S са $O(\log n)$, а началната сума, която беше "заключена" между тях няма как да "расте" с друга скорост – и тя е $O(\log n)$. С това приключва обяснението на първата полза от решето на Ератостен.

- **Задача:** ЕТИ 2016 D2.numdiv
- **Задача:** НОИ3 2022 D4.izobilni
- **Задача:** НОИ3 2024 D3.prime3

1.3.2 Минимален прост делител

По-горе споменахме, че решето на Ератостен може да се ползва и за намиране на най-малкия/големия прост делител на всяко число в интервал $[1, n]$. Това ще ни е полезно, когато искаме бързо да намираме каноничното разлагане на много различни числа в интервала.

Нека си представим, че имаме запълнен масив $min\_div[]$, който в $min\_div[x]$ държи най-малкия прост делител на x. Тогава чрез многократно изпълнение на $x \setminus = min\_div[x]$ ще обходим всички прости делители на x толкова пъти, колкото се срещат в каноничното разлагане и накрая стойността на x ще е 1. Броят деления / стъпки на този алгоритъм е равен на броя прости делители, а той е $O(\log_2 n)$. Доказателството е следното: Нека $x = p_1^{\alpha_1} \times p_2^{\alpha_2} \cdots p_k^{\alpha_k}$. Ще заменим всички прости множители с най-малкия възможен - 2. Така ще получим:

$$2^{\alpha_1+\alpha_2+\cdots+\alpha_k} = x' \leq x$$

$$\alpha_1 + \alpha_2 + \cdots + \alpha_k = \log_2 x' \leq \log_2 x$$

Ако преходът от първото към второто неравенство не е ясен, припомнете си дефиницията на log *от основните понятия.*

Сумата от алфите е точно броя прости делители в числото и ясно виждаме, че той е $O(\log n)$.

След като вече е ясно защо ни е полезно да имаме масива $min\_div[]$, нека се научим и как да го запълваме. Както в 1.3.1 ще започнем да обикаляме първо по числа d, а после и по неговите кратни, които ще означаваме с k. За всяко кратно ще проверим "Вече намерили ли сме минимален прост делител?". Ако не сме - сега ще сложим - в противен случай просто ще прескочим това число. Забележете, че винаги в $min\_div[x]$ ще имаме делител на x, защото на стъпката, в който сме го поставили сме имали $min\_div[k] := d$, за $k = d \times m$ за някакво естествено число m. За пълнота ще докажем и че делителят винаги ще е просто число. Да допуснем, че $min\_div[x]$ се оказва съставно число y - тогава то може да бъде разделено на един прост множител и някакъв "остатъчен" множител. Този прост множител е бил проверен по-рано в алгоритъма и също е бил делител на x, тоест няма как да сме поставили y в $min\_div[x]$ - там вече е бил поставен по-малък, прост делител на x.

Кодът е следният (ако нещо от горния абзац не е станало ясно, помогнете си като симулирате кода на лист хартия за първите 20 до 30 числа).

```
for(int d = 2; d <= n; d ++){
    for(int k = d; k <= n; k += d){
        if(min_div[k] == 0) min_div[k] = d;
    }
}
```

1.3.3 Решето със сложност $O(n \log \log n)$

До сега решенията ни имаха доказана сложност $O(n \log n)$. Можем да я намалим до $O(n \log \log n)$, ако директно пропускаме изпълнението на вътрешния цикъл, когато d е съставно число. Важно е да отбележим, че тогава **не получаваме информация** за съставните делители на числото - тоест в случай, че търсим сумата от **всички** делителите на числата, това ще доведе до грешка, но примерно когато се интересуваме от кои числа са прости или $min\_div[]$ масива, тази оптимизация е приложима. Доказателство няма да прилагам, понеже е твърде сложно за целевата група на текста.

Поставям имплементация на решето, оптимизирано до $O(n \log \log n)$ сложност, което намира простите числа в $[1, n]$.

```
    for(int d = 2; d <= n; d ++){
        if(is_comp[d] || d * d > n) break;
        for(int k = d * d; k <= n; k += d){
            is_comp[k] = true;
        }
    }
```

Тук трябва да внимаваме също за *integer overflow* при по-големи числа (може да се наложи да ползвате не *int*, *long long*). Освен това в допълнение на горната оптимизация обхождаме само кратните на d, които са по-големи от d^2, защото по-малките кратни вече са били посетени при някое по-малко просто число.

- **Задача:** НОИ3 2013 D4.primecnt
- **Задача:** ЕТИ 2024 B1.lpd

НОК / НОД. Алгоритъм на Евклид

Ще започнем с решаване на задачата "Как бързо да намерим НОД на две числа?", после ще докажем полезна връзка между стойността на $\text{lcm}(a, b)$ и $\gcd(a, b)$.

1.4.1 Алгоритъм на Евклид за намиране на НОД

Наивната идея за намиране на $\gcd(a, b)$ би била да обходим делителите на на по-малкото от тях и да ги проверим всичките. Това ще върви със сложност $O(\sqrt{\min(a, b)})$, която можем да подобрим с математически наблюдения.

Нека $g = \gcd(a, b)$ - все още не знаем точната стойност на g, но не пречи да го ползваме. Тогава, по дефиниция, съществуват $a = a' \times g$ и $b = b' \times g$. Без ограничение на общността, да кажем, че $a > b$ (дали $a \leq b$ или $a > b$ следващите ни аргументи винаги ще са верни). Нека почнем да вадим b от a, докато е възможно - така ще получим две числа $a - k \times b$ и b, чийто НОД все още е g (тук k е максималното възможно, за да е $a - k \times b \geq 0$).

$$b = b'g$$

$$a - k \times b = (a' - k \times b')g$$

Забележете, че тук може да получим $a - k \times b = 0$ *- това не е проблем, защото по дефиниция* $\gcd(g, 0) = g$.

Ако сега приложим горния алгоритъм за обикаляне на делителите ще направим по-малък брой стъпки, но можем да се справим и по-добре. Прилагайки горния аргумент за вадене на по-малкото число от по-голямото, докато едното число не стане 0, можем директно да намерим НОД-а на двете числа. Точно това и наричаме алгоритъм на Евклид - на практика рядко ще ни се налага да го разписваме, защото имаме функциите *__gcd* (дадена ни директно от компилатора) и *std::gcd* (дадена ни от *#include<numeric>*).

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

- **Задача:** ЕТИ 2019 D3.bus
- **Задача:** ПТИ 2019 D1.numbers

1.4.2 Връзка между a, b, $\gcd(a,b)$ и $\text{lcm}(a,b)$

Нека въведем каноничните разлагания на a и b.

$$a = p_1^{\alpha_1} \times p_2^{\alpha_2} \cdots p_k^{\alpha_k}$$

$$b = p_1^{\beta_1} \times p_2^{\beta_2} \cdots p_k^{\beta_k}$$

Понеже се задължаваме да ползваме еднакви прости числа в двата случая, може да се наложи някой степенен показател в някое от числата да е 0. Примерно за $a = 18$ и $b = 10$ получаваме $a = 2^2 \times 3^1 \times 5^0$ и $b = 2^1 \times 3^0 \times 5^1$.

Ще докажем, че:

$$\gcd(a,b) = p_1^{\min(\alpha_1,\beta_1)} \times p_2^{\min(\alpha_2,\beta_2)} \cdots p_k^{\min(\alpha_k,\beta_k)}$$

$$\text{lcm}(a,b) = p_1^{\max(\alpha_1,\beta_1)} \times p_2^{\max(\alpha_2,\beta_2)} \cdots p_k^{\max(\alpha_k,\beta_k)}$$

Първо ще отбележим, че колко пъти ползваме p_i и колко пъти ползваме p_j за $p_i \neq p_j$ в gcd или lcm няма нищо общо - тоест броя петици примерно в каноничното разлагане няма как да повлияе на броя двойки пак в него. Така че броя p_i-та в каноничното разлагане на gcd или lcm зависи само и единствено от броя им в същото разлагане на a и b.

Благодарение на горното свойство може да разгледаме какво се случва за a и b - степени на едно и също просто число. В общия случай, когато имаме няколко различни прости числа, просто ще умножим резултати за тях (все пак те не си "пречат" един на друг). Нека $a = 2^3$ и $b = 2^5$ - тогава най-големият общ делител на двете, ще бъде 2^3, а най-малкото общо кратно - 2^5. Формално, за $a = p^\alpha$ и $b = p^\beta$, $\gcd(a,b) = p^{\min(\alpha,\beta)}$ и $\text{lcm}(a,b) = p^{\max(\alpha,\beta)}$. Вече горните равенства за gcd и lcm стават ясни.

За финал, ще изразим lcm чрез a, b и gcd. Първо да отбележим, някои полезни свойства равенства:

$$a \times b = p_1^{\alpha_1+\beta_1} \times p_2^{\alpha_2+\beta_2} \cdots p_k^{\alpha_k+\beta_k}$$

$$\frac{x_1^y}{x_2^y} = x^{y_1-y_2}$$

$$x^{y_1} \times x^{y_2} = x^{y_1+y_2}$$

$$x + y = \max(x,y) + \min(x,y)$$

Ползвайки тях излиза, че:

$$\text{lcm}(a,b) = p_1^{\max(\alpha_1,\beta_1)} \times p_2^{\max(\alpha_2,\beta_2)} \cdots p_k^{\max(\alpha_k,\beta_k)}$$

$$lcm(a,b) = p_1^{\alpha_1+\beta_1-\min(\alpha_1,\beta_1)} \times p_2^{\alpha_2+\beta_2-\min(\alpha_2,\beta_2)} \cdots p_k^{\alpha_k+\beta_k-\min(\alpha_k,\beta_k)}$$

$$lcm(a,b) = \frac{a \times b}{p_1^{\min(\alpha_1,\beta_1)} \times p_2^{\min(\alpha_2,\beta_2)} \cdots p_k^{\min(\alpha_k,\beta_k)}}$$

$$\text{lcm}(a,b) = \frac{a \times b}{\gcd(a,b)}$$

Сега вместо да търсим алгоритъм за lcm, ще се задоволим с алгоритъма на Евклид и това свойство. (може и да ползваме функцията *std::lcm*, дадена ни от *#include<numeric>*)

- **Задача:** ЕТИ 2021 D2.divide

1.5.1 НОД/НОК на множество от числа

Ще покажем твърдението за три числа, то естествено се разширява за повече.

$g = \gcd(a, b, c) = \gcd(\gcd(a, b), c)$. Търсим най-голям общ делител на a, b и c - да се абстрахираме за момент от "най-голям" - този общ делител на a, b и c, трябва и да е общ делител само на a и b. Всички такива делители могат да се получат от $\gcd(a, b)$ чрез целочислено деление, така че като търсим g първо ще намерим $\gcd(a, b)$ и после ще се занимаваме с какви ограничения трябва да изпълнява g, за да е хем делител на $\gcd(a, b)$, хем на c. Това вече се свежда до решената задача за намиране на НОД на две числа. Сложността на това е $O(n + \log(A))$, където n е броят числа, а A е максималното от тях. Няма да влизаме в детайлите на доказателството, но в общи линии можем да си представим, че извършваме веднъж алгоритъма на Евклид и при нужда добавяме следващатото число в редицата. Този факт е детайлно доказан в анализа на НОИ2 2024 A2.segment.

$\text{lcm}(a, b, c) = \text{lcm}(\text{lcm}(a, b), c)$. Аналогично на горните разсъждения.

Гореспоменатото твърдение $a \times b = \gcd(a, b) \times \text{lcm}(a, b)$ може да се генерализира за повече променливи, но става по-сложно и непрактично. Като упражнение за читателя, може да се опитате да изведете някакво подобно твърдение.

1.5.2 Броят делители на n е приблизително $O(\sqrt[3]{n})$

Тук няма да даваме формално доказателство, защото е твърде сложно, а и самата теорема, от която следва това твърдение е по-обща. Общо казано, ако се налага да ползвате някаква оценка на броят делители на число в сложността на решението Ви, това не е зле.

1.5.3 Връзка между броя на делители на n и каноничното му разлагане

Нека $x = p_1^{\alpha_1} \times p_2^{\alpha_2} \cdots p_k^{\alpha_k}$. Тогава броя делители на x е равен на $(\alpha_1+1) \times (\alpha_2+1) \cdots (\alpha_k+1)$. Доказваме го като разгледаме каноничното разлагане на произволен делител на x - то ще бъде от вида $p_1^{\alpha'_1} \times p_2^{\alpha'_2} \cdots p_k^{\alpha'_k}$, където за всяко $\alpha'_i \leq \alpha$. Тук остава комбинаторната задача "Колко избора на променливите α'_i ще удовлетворяват съответните условия?"

1.5.4 Префиксен НОД

Нека имаме редица от числа $a_1, a_2, \cdots, a_n$. Ще построим нова редица $g_i{}_{i=1}^{n}$ по следните правила:

- $g_1 = a_1$
- $g_i = \gcd(g_{i-1}, a_i)$ за $i > 1$

Тогава са в сила следните две наблюдения:

- Редицата е ненарастваща.
- В редицата се срещат $\log(A)$ различни числа за $A = \max_{i=1}^{n}(a_i)$.

Първото свойство е ясно - няма как да вземем НОД на две числа и да очакваме да получим число, по-голямо от което и да е двете.

За да докажем второто свойство, ще разгледаме каноничното разлагане на числата g_i. Всяко следващо число от редицата g или запазва същите делители като миналото, или губи някакви делители. В 1.3.2 доказахме, че броя делители на n е не повече от $O(\log_2(n))$. Тук губим някакви делители, като сме почнали от най-много $\log_2(n)$ такива, тоест няма как да получим повече различни числа, от броя на делители ни.

- **Задача:** SOA 2020 D3.divs
- **Задача:** НОИ2 2024 A2.segment
- **Задача:** НОИ2 2014 A3.tab
- **Задача:** НОИ3 2022 A1.threediv

Автор: Иван Лупов