

Math [5]

Итак, мы закончили разговоры о C++, теперь можем начать заниматься интересными олимпиадными темами. Эта лекция будет посвящена математике, а если говорить точнее — теории чисел.

Определение простоты числа

Начнём мы с простого, будем говорить о натуральных и простых числах.

Пусть нам дано число n . Наша задача — определить, является ли оно простым или нет. (Note: простые числа - это те, что имеют ровно один делитель, отличный от единицы).

Напишем функцию, которая будет возвращать истину, когда число является простым и ложь, когда не является. Первая мысль - перебрать все возможные числа от 2 до n , не включая его, и убедиться, что среди них нет ни одного делителя.

```
1 bool is_prime(int n) {  
2     for (int i = 2; i < n; i++) {  
3         if (n % i == 0) {  
4             return false;  
5         }  
6     }  
7     return true;  
8 }
```

Сложность. Как можно заметить, в случае, если будет передано простое число, сложность этого алгоритма — $O(n)$. Достаточно медленно, ведь если нам дадут число порядка 10^{10} , то менее чем за секунду мы уже не сможем проверить, является ли оно простым или нет.

Как это дело можно оптимизировать?

Заметим, что у любого числа всегда есть делитель, не превосходящий корня из этого числа. Тогда перебор можно заканчивать после достижения корня.

Однако, если мы будем писать:

```
1 for (int i = 2; i <= sqrt(n); i++)
```

То из-за особенностей реализации вещественных чисел, $\text{sqrt}(n)$ может оказаться меньше, чем математический корень, и $i = \sqrt{n}$ не будет проверено. Поэтому, помня совет "стараться оставаться в целых числах превратим условие в $i * i \leq n$, просто возведя в квадрат левую и правую часть.

```
1 bool is_prime(int n) {
2     for (int i = 2; i * i <= n; i++) {
3         if (n % i == 0) {
4             return false;
5         }
6     }
7     return true;
8 }
```

Сложность. Сложность такого алгоритма уже $O(\sqrt{n})$, что значительно лучше. Сможем проверить даже $n = 10^{16}$. Правда придётся использовать тип *long long*.

Факторизация числа

В школе это наверное не рассказывают, но факторизация — это разложение числа в произведение простых делителей $n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \dots \cdot p_k^{\alpha_k}$. По основной теореме арифметики такое разложение единственно. Научимся делать это, немного преобразовав предыдущий код.

Будем передавать в функцию ссылки на два вектора, в которых будет храниться ответ — вектор самих простых чисел и степени, в которых эти числа входят в разложение.

Если очередное i не является делителем n , то ничего не делаем, переходим к следующей итерации. Если же делит, то добавляем его в вектор простых делителей, а в вектор степеней добавляем 0. Затем считаем, в какой степени простой делитель входит в разложение, деля n на i , пока делится и инкрементируя степень $\text{alpha}[i]$.

```

1 void factorization(int n, vector<int>& prime, vector<int>&
  ↪ alpha) {
2     for (int i = 2; i * i <= n; i++) {
3         if (n % i != 0)
4             continue;
5         prime.push_back(i);
6         alpha.push_back(0);
7         while (n % i == 0) {
8             alpha.back() += 1;
9             n /= i;
10        }
11    }
12    if (n > 1) {
13        prime.push_back(n);
14        alpha.push_back(1);
15    }
16 }

```

Сложность. В худшем случае, то есть когда факторизируемое число простое, сложность этого алгоритма будет также $O(\sqrt{n})$.

Решето Эратосфена

Проверять число на простоту, конечно, полезно, весело. Но вдруг нам придётся проверять сразу много чисел? Например, нужно отвечать, сколько встречается простых чисел в диапазоне $[1, n]$.

Для такой штуки давным давно Эратосфен придумал решето.

Идея заключается в том, что мы идём слева-направо по натуральным числам от 1 до n , просматривая очередное число, определяем не вычеркнуто ли оно. Если число не вычеркнуто, то будем говорить, что оно простое и вычёркивать все кратные ему до n . Хорошая анимация этого процесса (ссылка кликабельна): <https://zh.wikipedia.org/...>

В связи с тем, что единица не является простым числом, заранее вычеркнем его. Теперь посмотрим на пример, $n = 15$:

Первое незачёркнутое число — 2. Вычёркиваем все кратные ему, затем переходим к следующему незачёркнутому — 3. Также вычёркиваем кратные. Затем 4 мы пропускаем, 5 не вычеркнута, вычёркиваем кратные. И так далее. Иллюстрация:

```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

```

Напишем функцию решета Эратосфена. Она не будет возвращать ничего, будет принимать два аргумента. Первый — до какого числа мы хотим найти простые числа, второй — ссылка на вектор, в котором будет храниться ответ, само получившееся решето. Единица на i позиции будет значить, что число i — простое, ноль — составное.

```

1 void sieve(int n, vector<int>& result) {
2     result.assign(n + 1, 0);
3     result[0] = 1;
4     result[1] = 1;
5     for (int i = 2; i <= n; i++) {
6         if (result[i] != 0) // true => число составное
7             continue;
8         for (int j = i * 2; j <= n; j += i) {
9             result[j] = 1;
10        }
11    }
12 }

```

Что в этом коде можно заметить? Если мы рассмотрим какое-нибудь простое число $r > \sqrt{n}$, то оно уже не вычеркнет ни одно другое число, так как во всех $2 \cdot r, 3 \cdot r, \dots, (r-1) \cdot r$ присутствуют числа меньше r , которые были обработаны решетом раньше. А число $r \cdot r > n$. Поэтому имеет смысл завершать цикл после достижения $i = \sqrt{n}$. Из этого же можно заметить, что внутренний цикл можно начинать с $i \cdot i$.

Сложность. Оценим грубо сложность приведённого алгоритма, предполагая, что внутренний цикл выполняется даже для зачёркнутых.

$$\sum_{k=1}^n \frac{n}{i} = n \cdot \sum_{k=1}^n \frac{1}{i}$$

Из матана вы потом узнаете, что

$$\sum_{k=1}^n \frac{1}{i} \approx \log n$$

. Поэтому грубо сложность получается $O(n \log n)$.

На самом деле сложность решета Эратосфена равна $O(n \log \log n)$. Однако доказывать это можно только обладая достаточными знаниями в матане и теории чисел. Кому интересно, не строгое доказательство есть на e-maxx.

Можно прикинуть, насколько маленький множитель даёт $\log \log n$. Если $n = 10^6$, то $\log \log n \approx \log \log 2^{20} \approx \log 20 \approx 4$. Хороший множитель!

Хотелось бы ещё упомянуть о существовании алгоритма решета Эратосфена за $O(n)$. Его можно будет рассказать когда-нибудь потом.

Применение решета

Решето Эратосфена — очень полезная штука. С помощью него можно ускорять некоторые операции, связанные с простыми числами. Например, можно построить из него вектор только простых чисел и проверять числа на простоту, пробегаясь по нему.

Но сейчас рассмотрим кое-что другое. Давайте немного изменим наш алгоритм решета так, чтобы с его помощью можно было находить разложение числа на множители.

```
1 void sieve(int n, vector<int>& result) {
2     result.assign(n + 1, 0);
3     result[0] = 1;
4     result[1] = 1;
5     for (int i = 2; i * i <= n; i++) {
6         if (result[i] != 0)
7             continue;
8         for (int j = i * 2; j <= n; j += i) { // было j = i *
          ↪ i
9             result[j] = i; // 1 заменили на i
10        }
11    }
12 }
```

Теперь в `result[i]` будет лежать максимальный простой делитель числа i . Затем мы хотим разложить *val* на простые делители. Будем делать это так:

```

1  ...
2  vector<int> divs;
3  sieve(n, divs);
4  vector<int> prime;
5  int val;
6  cin >> val;
7  while (divs[val] != 0) {
8      prime.push_back(divs[val]);
9      val /= divs[val];
10 }
11 prime.push_back(val);
12 ...

```

Подумайте над тем, как работает этот код, и что в итоге будет лежать в *prime*.

Сложность. Сложность нахождения разложения числа, без учёта построения решета Эратосфена, получается равной $O(\text{количество простых делителей})$, что не может не радовать.

Это было решето Эратосфена. Вообще на него бывает довольно много задач, где используются те или иные фишки, поэтому его нужно уметь быстро и безошибочно писать в любой ситуации.

Алгоритм Евклида

Теперь давайте поговорим о нахождении наибольшего общего делителя двух чисел. НОД (gcd, greatest common divisor) — это наибольшее число, которое делит два данные числа.

Назовём функцию, выдающую нам НОД двух чисел, $gcd : (\mathbb{N}, \mathbb{N}) \rightarrow \mathbb{N}$. Заметим важное свойство этой функции.

Лемма. Если $a < b$, то: $gcd(a, b) = gcd(a, b - a)$

Покажем, что это так. Пусть $d = gcd(a, b)$, тогда $a = a_1d$, $b = b_1d$, и $b - a = d(b_1 - a_1)$. При том $(b_1 - a_1) \perp a_1$, так как иначе $gcd(a_1, b_1) \neq 1$ и наш d не является наибольшим общим делителем. Значит, $gcd(a, b - a) = d = gcd(a, b)$.

Наверное некоторые из вас знают алгоритм Евклида, в школе это могут рассказывать. На пальцах это можно объяснить так: у нас два аргумента функции уменьшаются, при этом всё ещё делятся на НОД, следовательно когда-то они уменьшатся настолько, что станут равны НОДу.

Но давайте сформулируем алгоритм Евклида чуточку строже.

Алгоритм Евклида. Пусть у нас есть два натуральных числа a и b , тогда существуют целые неотрицательные числа q и r , для которых $b = qa + r$ и $r < a$ — остаток от деления b на a , мы хотим найти $d = gcd(a, b)$. Алгоритм Евклида заключается в следующем.

Пусть у нас $a_0 = b$ и $a_1 = a$. Поделим a_0 на a_1 с остатком: $a_0 = q_1a_1 + a_2$; затем поделим a_1 на a_2 с остатком: $a_1 = q_2a_2 + a_3$ и т.д. В конце концов получим $a_{k-1} = q_k a_k$. Все числа a_0, a_1, \dots, a_k имеют вид $xa_0 + ya_1$, где $x, y \in \mathbb{Z}$.

Поэтому a_k делится на любой общий делитель чисел a_0 и a_1 . С другой стороны, $a_{k-2} = q_{k-1}a_{k-1} + a_k = (q_{k-1}q_k + 1)a_k$ и т.д., поэтому числа a_0 и a_1 делятся на a_k . Это значит, a_k — НОД чисел a_0 и a_1 .

Нахождение gcd

С теорией здесь можно почти закончить. Давайте теперь посмотрим, как написать приведённый алгоритм в коде. Опустим реализацию с вычитаниями, так как она работает долго, а если вам когда-нибудь понадобится её писать, вы с лёгкостью сможете изменить знак взятия остатка на знак минуса в своём коде.

Итак, у нас происходит череда взятий остатков в алгоритме Евклида. Когда же нужно остановиться? Как вы видели в последнем выражении алгоритма $a_{k-1} = q_k a_k$ остаток равен нулю. Значит и мы остановимся, когда получим 0 в остатке.

```
1 int gcd(int a, int b) {  
2     while (a != 0) {  
3         b %= a;  
4         swap(a, b);  
5     }  
6     return b;  
7 }
```

Сложность. Показать вычислительную сложность данного алгоритма не так просто. Поэтому сделаем это не строго, доверившись Габриэлю Ламе, заметившем, что наихудшим случаем для данного алгоритма будут являться числа Фибоначчи. Это числа вида $F_i = F_{i-1} + F_{i-2}$.

Если вызвать $\text{gcd}(F_n, F_{n-1})$, то оно посчитается за $n - 2$ шага. Оценивая грубо, можно сказать, что значения чисел Фибоначчи растут не быстрее, чем $\left(\frac{1 + \sqrt{5}}{2}\right)^n$. Обратная функция от возведения в степень

— взятие логарифма. Поэтому, если наше $\max(a, b) \leq \left(\frac{1 + \sqrt{5}}{2}\right)^n$, то сложность алгоритма в худшем случае будет $O(\log(\max(a, b)))$.

Давайте приведём ещё более короткую рекурсивную реализацию. Она полностью следует определению алгоритма.

```
1 int gcd(int a, int b) {  
2     return a == 0 ? b : gcd(b % a, a);  
3 }
```

В ней используется такая штука, которая зачастую помогает сократить код — тернарный оператор. Он соответствует:


```

1  if (/* какое-то условие */)
2      /* подставить результат1 */;
3  else
4      /* подставить результат2 */;

```

Используйте тернарный оператор только в тех случаях, когда читабельность кода от его использования только улучшается. Иначе ваш код никто не сможет понять.

Нахождение lcm

Наименьшее общее кратное (lcm, least common multiple) двух целых чисел — наименьшее натуральное число, которое делится на эти числа без остатка.

Вспомнив, что $a = a_1d$, $b = b_1d$ ($d = \gcd(a, b)$), перемножим: $ab = a_1b_1d^2$, значит $\text{lcm}(a, b) = a_1b_1d = \frac{ab}{d}$.

```

1  int lcm(int a, int b) {
2      return a / gcd(a, b) * b;
3  }

```

Расширенный алгоритм Евклида

Все эти НОДы, НОКи весёлые, простые, часто встречаются в различных задачах. Но давайте перейдём к чему-то посложнее, например, научимся решать простенькие диофантовы уравнения. Вообще диофантовы — это такие уравнения, переменные в которых принимают целочисленные значения.

Мы сейчас будем рассматривать случай линейных диофантовых уравнений **двух переменных с натуральными** коэффициентами:

$$ax + by = \gcd(a, b)$$

Где это встречается? Например, так выглядит уравнение прямой на плоскости.

Задача: решить уравнение $ax + by = \gcd(a, b)$ в целых числах.

Теорема. Такое уравнение всегда имеет решения.

Доказательство конструктивное, то есть докажем просто построив алгоритм решения таких уравнений.

Получим переменные x , y , идя обратно по алгоритму Евклида, от двух последних значений, к тем, из которых он запускался.

Заметим такой переход. Если мы знаем решение:

$$(b \% a)x_1 + ay_1 = \gcd(a, b)$$

И хотим получить решение

$$ax + by = \gcd(a, b)$$

То мы можем представить $(b \% a)$ как $(b - \lfloor \frac{b}{a} \rfloor \cdot a)$, а затем подставить в исходное:

$$\left(b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a\right)x_1 + ay_1 = \gcd(a, b)$$

Перегруппируем слагаемые, чтобы a были с a , b были с b :

$$a\left(y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1\right) + bx_1 = \gcd(a, b)$$

Таким образом, x и y можно выразить через полученные значения:

$$\begin{aligned}x &= y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1 \\y &= x_1\end{aligned}$$

Реализация.

Мы повыражали x и y через x_1, \dots, x_k и y_1, \dots, y_k . Посмотрим, как это будет выглядеть в коде.

Так как мы будем идти в обратном направлении, проще всего воспользоваться рекурсивной функцией. Для рекурсивной функции нужно указать точку завершения, иначе она будет вызывать сама себя до момента, когда стек памяти будет полностью заполнен.

Будем передавать в неё два дополнительных аргумента — ссылки на x , y .

```
1 int gcd(int a, int b, int& x, int& y) {  
2     if (a == 0) { // a * 0 + b * 1  
3         x = 0; // значение при a  
4         y = 1; // значение при b  
5         return b;  
6     }  
7     int x1, y1;  
8     // решим задачу попроще,  
9     // получим для неё x1, y1  
10    int d = gcd(b % a, a, x1, y1);  
11    x = (y1 - (b / a) * x1);  
12    y = x1;  
13    return d;  
14 }
```

Сложность. Очевидно $O(\log(\max(a, b)))$.

Вот такой вот алгоритм.

Про операции по модулю

Сложная часть закончилась, сейчас будет небольшое отступление в целом про арифметику остатков.

Давайте посмотрим на различные операции, которые можно выполнять при вычислениях в арифметике остатков.

Сначала посмотрим, как можно представить, что какие-то числа a и b делятся с остатком на число p :

$$a = q_ap + r_a$$

$$b = q_bp + r_b$$

Теперь посмотрим, как работают операции сложения, вычитания и умножения по модулю:

- $(a + b) \bmod p = (r_a + r_b) \bmod p$
- $(a - b) \bmod p = (r_a - r_b) \bmod p$
- $(a \cdot b) \bmod p = (r_a \cdot r_b) \bmod p$

Как видим, они работают хорошо.

А как обстоит дело с операцией деления? Это хороший вопрос, так как, например, $(2 \cdot 4) \% 5 = 3$. Наверное хотелось бы, чтобы при делении 3 на 2, мы получали ответ 4. То есть $(3/2) \% 5 = 4$. О том, как это делать мы говорим на следующей лекции про математику.

Итог

В этой лекции мы посмотрели немного, как применять теорию чисел в базовых олимпиадных алгоритмах:

- проверка числа на простоту
- факторизация числа
- решето Эратосфена
- нахождение НОД и НОК
- нахождение решения уравнений $ax + by = \gcd(a, b)$