Основные алгоритмы, комментарии по второму листку

Шибаев Иннокентий

February 15, 2021

1 По поводу алгоритма Евклида

Алгоритм Евклида берет на вход два числа a и b и выдает их наибольший общий делитель НОД(a, b) (greatest common divisor $gcd(a, b)$). Его рекурсивная запись представлена ниже в Алгоритме 1.

Алгоритм 1: Алгоритм Евклида

Input : $a, b \in \mathbb{N} \cup \{0\}$
Output: Наибольший общий делитель НОД(a, b)

```
Function gcd(a, b) :
    if b == 0 then
        return a
    return gcd(b, a mod b)
```

Разберем что здесь происходит и в процессе докажем корректность и найдем сложность этого алгоритма.

Доказательство корректности. На каждом шаге от одного числа берется остаток по модулю другого, таким образом числа строго уменьшаются (кроме, возможно, первого шага, если $a < b$, но тогда монотонное уменьшение начнется со второго вызова, когда мы фактически вызовем $gcd(b, a \mod b) = gcd(b, a)$). Если на каком-то шаге число a делится без остатка на b то следующий вызов gcd будет иметь вид $gcd(b, 0)$, что, по условию в начале, вернет b.

Из вышеописанного строгого убывания и ограниченности снизу (и того что мы работаем с целыми числами) мы получаем что этот процесс конечен, рано или поздно алгоритм перестанет идти вглубь и начнет "разматываться".

Осталось заметить, что при этом процессе сохраняется определенный инвариант – НОД$(a, b) =$ НОД$(b, a \mod b)$, это следует из того что $a = tb + c$ где $0 \leqslant c < b$, $c = a \mod b$, и при этом т.к. $a, b \vdots$ НОД(a, b) получаем что $a \mod b = c \vdots$ НОД(a, b) (т.к. левая часть в уравнении $a = tb + c$ делится на НОД(a, b)).

Итак, алгоритм конечен, и возвращаемое им значение всегда делится на НОД(a, b), при этом, как мы знаем, выход совершается только тогда, когда $b == 0$, т.е. когда на предыдущем шаге одно число разделилось без остатка на другое – а это значит что то что мы возвращаем в точности равно искомому наибольшему общему делителю, тем самым корректность доказана. □

Оценка сложности. Напоминаю, что нам нужно искать сложность как функцию от длины входа – в данном случае на вход алгоритму подаются числа a, b, т.е. длина входа это $\lceil \log_2(a) \rceil + \lceil \log_2(b) \rceil$ (если мы считаем что числа подаются в бинарном виде). На каждом шаге алгоритм берет одно число по модулю другого, тем самым переходя от чисел длин $(\lceil \log_2(a) \rceil, \lceil \log_2(b) \rceil)$ к числам длины $(\lceil \log_2(b) \rceil, \lceil \log_2(c) \rceil)$, где $\lceil \log_2(c) \rceil \leqslant \lceil \log_2(b) \rceil$.

При этом заметим, что происходит одно из двух: либо $\lceil \log_2(c) \rceil == \lceil \log_2(b) \rceil$, т.е. (в виду того что мы работаем с двоичной системой) число b менее чем в 2 раза превосходит c (иначе они были бы разной длины!) и тогда на следующем шаге получится что число b уменьшится больше чем в 2 раза (когда мы перейдем к паре $(c, b \bmod c)$), а значит его длина уменьшится как минимум на 1.

Другой вариант, это случай, когда $\lceil \log_2(c) \rceil < \lceil \log_2(b) \rceil$. Но тогда на следующем шаге число b опять же уменьшится не менее чем в 2 раза, т.е. длина опять же уменьшится как минимум на 1.

Таким образом, за каждые максимум 2 шага суммарная длина чисел уменьшается на 1. Таким образом алгоритм отработает максимум за $C \cdot (\lceil \log_2(a) \rceil + \lceil \log_2(b) \rceil)$, и, значит, его сложность имеет вид $O(\lceil \log_2(a) \rceil + \lceil \log_2(b) \rceil)$. Это линейный от длины входа алгоритм, значит полиномиальный. □

Однако нас на самом деле несколько больше интересует расширенная версия данного алгоритма представленная ниже (Алгоритм 2), которая находит, помимо НОД(a, b), еще и целочисленные коэффициенты x, y такие, что $ax + by = \text{НОД}(a, b)$.

Алгоритм 2: Расширенный алгоритм Евклида

Input : $a, b \in \mathbb{N} \cup \{0\}$
Output: Наибольший общий делитель НОД(a, b), $x, y \in \mathbb{Z}$ такие, что $ax + by = \text{НОД}(a, b)$

```
Function gcd_ext(a, b) :
    if b == 0 then
        return (a,1,0)
    (d, x, y) = gcd_ext(b, a mod b)
    return (d, y, x − (a div b) · b)
```

Опять же, разберемся с корректностью и скоростью работы данного алгоритма.

Доказательство корректности. Мы уже доказали выше корректность алгоритма Евклида для поиска НОД(a, b), и здесь мы эту часть никак не трогаем, так что осталось только показать что получаемые x, y являются решениями уравнения $ax+by = \text{НОД}(a, b)$.

И здесь нам поможет тот факт что мы работаем с рекурсивным алгоритмом, которые очень удобно перекладываются на принципы мат. индукции (дальнейшее основывается на доказательстве с сайта e-maxx).

Предположим что мы имеем решение x_1, y_1 для $bx_1 + (a \bmod b)y_1 = \text{НОД}(a, b)$, и мы хотим восстановить решение x, y для уравнения $ax + by = \text{НОД}(a, b)$. Мы знаем, что $a \bmod b = a - (a \operatorname{div} b) \cdot b$. Подставим это в первое уравнение

$$\begin{aligned} bx_1 + (a \bmod b)y_1 &= \text{НОД}(a, b) \\ \Rightarrow bx_1 + (a - (a \operatorname{div} b) \cdot b)\, y_1 &= \text{НОД}(a, b) \\ \Rightarrow ay_1 + b\,(x_1 - (a \operatorname{div} b) \cdot y_1) &= \text{НОД}(a, b) \end{aligned}$$

и сравнивая это с уравнением $ax + by = \text{НОД}(a, b)$ получаем формулы

$$\begin{cases} x = y_1 \\ y = x_1 - (a \operatorname{div} b) \cdot y_1 \end{cases}$$

для пересчета решения для исходного уравнения. Осталось только получить базу индукции – при $b == 0$ мы получаем уравнение вида $ax + 0y = \text{НОД}(a, b)$ и таким образом $x = 1, y = 0$ (что мы и возвращаем в алгоритме). □

Оценка сложности. А здесь все еще проще – мы не делаем отдельных вызовов, так что применяя то же рассуждение что и в предыдущем случае мы получаем оценку вида $C \cdot (\lceil \log_2(a) \rceil + \lceil \log_2(b) \rceil)$ (с несколько большей константой, в связи с дополнительными действиями перед возвращением значений), и, таким образом, сложность алгоритма остается той же – $O(\lceil \log_2(a) \rceil + \lceil \log_2(b) \rceil)$. □

Зачем это нужно? С помощью расширенного алгоритма Евклида (2) можно решать диофантовы уравнения вида $ax+by=c$. Такое уравнение имеет решение тогда и только тогда, когда $c \vdots \text{НОД}(a,b)$. Действительно, если $c = t \cdot \text{НОД}(a,b)$, то решив уравнение $ax_1 + by_1 = \text{НОД}(a,b)$ с помощью вышеописанного алгоритма мы можем положить $x = tx_1$ и $y = ty_1$ тем самым получив решение исходного уравнения.

Если же c не кратно наибольшему общему делителю то решения не существует т.к. левая часть уравнения делится на $\text{НОД}(a,b)$.

Рассмотрим задачу подобную задаче 1 из второго листка

Задача 1.1. Найдите с помощью расширенного алгоритма Евклида обратный остаток $19^{-1} \pmod{147}$.

Решение (Вариант 1, напрямую применить расширенный алгоритм Евклида)**.** Итак, нам надо найти x такой, что $19x = 1 \pmod{147}$, т.е. решить уравнение вида

$$19x + 147y = 1.$$

Заметим, что 19 и 147 взаимно просты, т.е. $\text{НОД}(19, 147) = 1$, т.е. данное уравнение имеет решение.

Попробуем применить алгоритм расписанный выше напрямую. Для этого начнем заполнять следующую таблицу в которой будем отмечать то с какими параметрами вызывается очередной уровень рекурсии (каждому вызову соответствует одна строка, красным показаны новые значения):

a	x	b	y	d
19		147		
147		19		

a	x	b	y	d
19		147		
147		19		
19		14		

a	x	b	y	d
19		147		
147		19		
19		14		
14		5		

a	x	b	y	d
19		147		
147		19		
19		14		
14		5		
5		4		

a	x	b	y	d
19		147		
147		19		
19		14		
14		5		
5		4		
4		1		

a	x	b	y	d
19		147		
147		19		
19		14		
14		5		
5		4		
4		1		
1		0		

На начальном этапе расширенный алгоритм Евклида не отличается от обычного – мы просто делим первое число на второе с остатком и меняем местами, повторяя до тех пор пока второе не станет равным 0. Теперь мы можем начать подъем вверх. Значения для x, y внизу получаются по алгоритму просто как 1 и 0 соответственно, после чего мы начинаем пересчитывать предыдущие x и y по формулам которые мы вывели выше ($x = y_1$ и $y = x_1 - (a \text{ div } b) \cdot y_1$, $d = d_1$) где x_1, y_1 и d_1 это в данной таблице просто x, y и d находящиеся на уровень ниже тех что мы хотим вычислить). То что в этих формулах слева мы будем выделять красным, а то что справа – синим:

a	x	b	y	d
19		147		
147		19		
19		14		
14		5		
5		4		
4		1		
1	1	0	0	1

a	x	b	y	d
19		147		
147		19		
19		14		
14		5		
5		4		
4	0	1	1	1
1	1	0	0	1

a	x	b	y	d
19		147		
147		19		
19		14		
14		5		
5	1	4	-1	1
4	0	1	1	1
1	1	0	0	1

a	x	b	y	d
19		147		
147		19		
19		14		
14	-1	5	3	1
5	1	4	-1	1
4	0	1	1	1
1	1	0	0	1

a	x	b	y	d
19		147		
147		19		
19	3	14	-4	1
14	-1	5	3	1
5	1	4	-1	1
4	0	1	1	1
1	1	0	0	1

a	x	b	y	d
19		147		
147	-4	19	$3-7\cdot(-4)$	1
19	3	14	-4	1
14	-1	5	3	1
5	1	4	-1	1
4	0	1	1	1
1	1	0	0	1

a	x	b	y	d
19	31	147	-4	1
147	-4	19	31	1
19	3	14	-4	1
14	-1	5	3	1
5	1	4	-1	1
4	0	1	1	1
1	1	0	0	1

(причем последний переход, как видим, это просто перемена мест, т.к. фактически самое первое действие что мы делали в ходе алгоритма это поменяли местами a и b). В результате получили, что $x = 31$ и $y = -4$, таким образом $19 \cdot 31 - 147 \cdot 4 = 1$. Значит $19^{-1} \pmod{147} = 31$. □

При реализации (имеется в виду реализации в коде) этот метод на самом деле удобен, и, чаще всего, его пишут именно так. Но он не очень удобен для того чтобы выполнять его руками, поэтому предлагается другой способ, не требующий вообще использования формул пересчета.

Решение (Вариант 2, через обычный алгоритм Евклида)**.** Нам надо решить уравнение вида

$$19x + 147y = 1.$$

Теперь будем заполнять таблицу из трех колонок (x, y, $ax+by$) следующим образом. В начале впишем туда значения при $x = 0,\ y = 1$ и наоборот:

x	y	$ax + by$
0	1	147
1	0	19

Теперь посмотрим на третий столбец. Там фактически сейчас стоят значения b и a. Если мы запустим алгоритм Евклида (т.е. будем каждый раз брать верхнее по модулю нижнего и записывать его следующим) то в конце-концов в этом столбце внизу мы получим НОД(a, b). Идея состоит в том чтобы повторять эти же действия со строками слева (которые отвечают значениям x и y). "Повторять" здесь имеет следующее значение – когда в третьем столбце мы делим одно значение на другое по модулю нам слева надо отнять одну строку от другой столько раз, сколько мы фактически отнимаем в третьем столбце второе значение от первого совершая деление с остатком, т.е. к примеру для чисел 147 и 19 мы должны от первой строки отнять вторую 7 раз (т.к. $147 \operatorname{div} 19 = 7$)

x	y	$ax + by$
0	1	147
1	0	19
-7	1	14

и теперь мы переходим уже к двум последним строкам. Продолжая это мы получим

x	y	$ax+by$
0	1	147
1	0	19
-7	1	14
8	-1	5

x	y	$ax+by$
0	1	147
1	0	19
-7	1	14
8	-1	5
-23	3	4

x	y	$ax+by$
0	1	147
1	0	19
-7	1	14
8	-1	5
-23	3	4
31	-4	1

Мы получили тот же результат что и предыдущим методом ($x = 31$), но заметно быстрее, при этом нам не понадобилось совершать "обратный проход", как в прошлом случае, и мы не использовали никаких формул пересчета. □

Еще тут можно заметить что во втором случае мы используем только последние две строки на каждом шаге работы, поэтому это можно реализовать циклом, и поддерживать только $O(1)$ значений (заметим что в прошлом случае в виду того что нам надо поддерживать весь стек лучше чем его глубина, т.е. $O\left(\lceil \log_2(a) \rceil + \lceil \log_2(b) \rceil\right)$ по памяти у нас сделать не получится.

2 По поводу зависимости временной сложности от модели вычислений

```
1 Function QPower(b, n) :
2     if n > 1 then
3         x = QPower(b, ⌊n/2⌋)
4         if n нечётно then
5             return b × x × x
6         return x × x
7     return b
```

(Данный раздел носит больше факультативный характер, и вроде особо не нужен для решения задач в этом курсе, так что можете скипнуть :))

Вообще про модели вычислений есть отдельный курс, называется AMB, но хочется сказать пару вещей по этому поводу раз уж мы разобрали очень удобную для этого задачу:

Задача 2.1. Оцените временную сложность алгоритма `QPower`, вычисляющего b^n. Считайте, что арифметические операции стоят $O(1)$.

Эта задача была решена на семинаре, было показано что время работы данного алгоритма в этом случае имеет вид $O(\log n)$, т.е. линейно по входу. Предлагается рассмотреть несколько иной вариант этой задачи:

Задача 2.2. Оцените временную сложность алгоритма `QPower`, вычисляющего b^n. Считайте, что умножение выполняется методом "в столбик", т.е. для перемножения двух чисел длиной a и b надо потратить $O(ab)$ операций.

Решение. Итак, пусть у нас для вычислено $x = \texttt{QPower}(b, \lfloor n/2 \rfloor)$, и теперь нам надо сделать какие-то перемножения и отправить это "наверх". Есть два варианта – четное и нечетное n, разберем более сложный случай нечетного n.

В этом случае нам надо вычислить произведение $b \times x \times x$. Заметим, что если длина b имеет вид $\lceil \log_2(b) \rceil$, то длина x (из того что алгоритм корректно вычисляет b^n) имеет вид

$$\left\lceil \log_2\left(b^{\lfloor n/2 \rfloor}\right) \right\rceil = \left\lceil \left\lfloor \frac{n}{2} \right\rfloor \log_2(b) \right\rceil \leqslant \left\lfloor \frac{n}{2} \right\rfloor \lceil \log_2(b) \rceil$$

(где равенство достигается если b это степень двойки). Соответственно, чтобы перемножить число b и число x надо затратить (из условия что у нас перемножение в столбик)

$$\mathrm{C}\left\lceil \log_2(b) \right\rceil \cdot \left\lfloor \frac{n}{2} \right\rfloor \left\lceil \log_2(b) \right\rceil = \mathrm{C}\left\lfloor \frac{n}{2} \right\rfloor \left\lceil \log_2(b) \right\rceil^2$$

операций, а чтобы возвести x в квадрат

$$\mathrm{C}\left(\left\lfloor \frac{n}{2} \right\rfloor \left\lceil \log_2(b) \right\rceil\right)^2 = \mathrm{C}\left\lfloor \frac{n}{2} \right\rfloor^2 \left\lceil \log_2(b) \right\rceil^2$$

операций. Допустим в начале мы перемножаем $b \times x$ и затем уже умножаем результат на x, тогда нам понадобится (с учетом того что мы перемножаем уже числа длины $\left\lfloor \frac{n}{2} \right\rfloor + 1$ и $\left\lfloor \frac{n}{2} \right\rfloor$):

$$\mathrm{C}\left\lfloor \frac{n}{2} \right\rfloor \left\lceil \log_2(b) \right\rceil^2 + \mathrm{C}\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right)\left\lfloor \frac{n}{2} \right\rfloor \left\lceil \log_2(b) \right\rceil^2 = \mathrm{C}\left(\left\lfloor \frac{n}{2} \right\rfloor^2 + 2\left\lfloor \frac{n}{2} \right\rfloor\right)\left\lceil \log_2(b) \right\rceil^2$$

(замечу что если перемножать в другом порядке то получится то же, ассоциативность все же есть, а вот если затронуть произведения матриц на вектора то там вещи вроде $ABCx$ где A, B, C – матрицы а x – вектор считают именно в порядке $Cx \to B(Cx) \to A(B(Cx))$, т.к. умножение вектора на матрицу занимает порядка кол-ва элементов матрицы действий, а перемножение матриц (в случае квадратных) уже куб от размера матрицы).

Давайте для простоты положим что $n = 2^m$, а $b = 2^l$. Тогда на каждом шаге не надо будет даже умножать на b, и останется только первое слагаемое, и таким образом затраты по времени будут иметь вид

$$\sum_{i=0}^{m-1} \mathrm{C}\left(2^i\right)^2 l^2 = Cl^2 \sum_{i=0}^{m-1} 2^{2i} = Cl^2 \frac{2^{2m}-1}{4-1} = C\frac{n^2-1}{3}\log^2 b = O\left(n^2 \log^2 b\right)$$

А теперь сравним это с самым простым что мы можем придумать – с последовательным умножением на b просто n раз подряд. Тогда первое перемножение занимает $C\log^2 b$, второе $C\log b \cdot \log b^2 = 2C\log^2 b$, третье $3C\log^2 b$ и т.д., т.е.

$$\sum_{i=1}^{n-1} \mathrm{C} \cdot i \log^2 b = C\frac{n(n-1)}{2}\log^2 b = O\left(n^2 \log^2 b\right)$$

т.е. в такой модели вычислений алгоритм бинарного возведения в степень вообще говоря не особо-то полезен (хотя у него константа получше – там деление на 3, а в самой прямой версии всего на 2). Но все же, мы получили что в другой модели вычислений данный алгоритм не дает улучшения по порядку скорости работы!

Как это можно улучшить? Есть различные методы быстрого перемножения двух длинных чисел (весь этот раздел называется, кстати говоря, "длинная арифметика"), к примеру алгоритм Карацубы который перемножает два числа длины n за $O\left(n^{\log_2 3}\right)$, или основанный на дискретном преобразовании Фурье (за $O(n \log n)$, хотя, точнее, за $O(n \log n \log\log n)$), первый довольно простой, и мы его даже будем скорее всего рассматривать впоследствии.