

# 1 Производящие функции.

*Замечание.* Вообще производящая функция — не очень такое название. Потому что это скорее не функция, а способ записи бесконечных последовательностей.

Вот есть у нас  $2, 4, 8, 16, \dots$  и сразу понятно, что имеется ввиду. А вот когда мы видим  $1, 2, 5, 14, 42$  и знающие люди поймут, что это, скорее всего, числа Каталана. Но всё равно это не то чтобы однозначно определяет, что мы имеем ввиду.

Можно записать  $a_n = 2^n$  и сразу станет понятно, что это степени двойки. А когда мы запишем  $a_n = C_n$ , то поймут точно не все.

Короче, работать с такими вещами не очень приятно. И люди задумались: а как компьютеру дать бесконечные последовательности, чтобы он их понял.

И люди нашли инструмент из теории вероятности и статистики — собственно, производящие функции.

**Определение 1.** Пусть  $\{a_n\}_{n=0}^\infty$  — последовательность. Тогда **формальный степенной ряд** этой последовательности — это запись вида

$$A(t) = a_0t^0 + a_1t^1 + a_2t^2 + \dots + a_nt^n + \dots$$

*Замечание.* Вот есть у нас многочлен:  $t^2 + t - 7$ . Что такое  $t$ ? Ну, по сути, буква. Она не обладает значением, мы можем только потом уже рассмотреть многочлен как многочлен над каким-то кольцом, и уже значения его анализировать.

Вот и тут по сути мы имеем просто букву  $t$ , она ничему не равна.

**Утверждение.** Очевидно, формальные степенные ряды биективно сопоставляются последовательностям.

*Замечание.* Пока что формальный степенной ряд ничем не лучше просто последовательности. Но на самом деле с формальными степенными рядами можно производить полезные операции, которые позволят нам конечным количеством символов описать интересующие нас последовательности.

*Пример.* Какие формальные степенные ряды мы уже можем легко записать? Ну, те, которые соответствуют многочленам. То есть формальные степенные ряды тех последовательностей, которые имеют конечное количество ненулевых элементов.

**Определение 2.** Суммой формальных степенных рядов называется степенной ряд суммы их последовательностей.

Умножением формального ряда на число называется произведение его последовательности и этого числа.

*Замечание.* Понятно, что это согласуется с тем, как мы могли бы сложить ряды.

Но ведь ряды ещё можно умножать. Что получим?

**Определение 3.** Произведением формальных степенных рядов

$$A(t) = a_0t^0 + a_1t^1 + a_2t^2 + \dots + a_nt^n + \dots \quad B(t) = b_0t^0 + b_1t^1 + b_2t^2 + \dots + b_nt^n + \dots$$

называется ряд

$$(AB)(t) = C(t) = c_0 + c_1t^1 + c_2t^2 + \dots + c_nt^n + \dots \quad c_k = \sum_{j=0}^k a_j b_{k-j}$$

*Замечание.* А делить как? Ну, если  $\frac{A(t)}{B(t)} = C(t)$ , то  $A(t) = (BC)(t)$ . Ну,

$$a_0 = b_0c_0 \Rightarrow c_0 = \frac{a_0}{b_0}$$

$$a_1 = b_1c_0 + b_0c_1 \Rightarrow c_1 = \frac{a_1 - c_0b_1}{b_0}$$

**Определение 4. Частным формальных степенных рядов**

$$A(t) = a_0t^0 + a_1t^1 + a_2t^2 + \dots + a_nt^n + \dots \quad B(t) = b_0t^0 + b_1t^1 + b_2t^2 + \dots + b_nt^n + \dots$$

где  $b_0 \neq 0$  называется ряд

$$\left(\frac{A}{B}\right)(t) = C(t) = c_0 + c_1t^1 + c_2t^2 + \dots + c_nt^n + \dots \quad c_k = \frac{a_k - \sum_{j=0}^{k-1} c_j b_{k-j}}{b_0}$$

*Пример.*

$$\frac{1}{1-t-t^2}$$

Так,  $c_0 = 1$ ,

$$c_1 = \frac{a_1 - c_0 b_1}{b_0} = 1$$

$$c_2 = \frac{a_2 - c_1 b_1 - c_0 b_2}{b_0} = 2$$

Давайте в общем виде, учитывая тот факт, что  $b_k = 0$  для  $k > 2$ ,

$$c_n = a_n - c_{n-1}b_1 - c_{n-2}b_2 = c_{n-1} + c_{n-2}$$

Да это же числа Фибоначчи!

**Утверждение.** Если  $a_n \in \mathbb{Z}$ ,  $b_n \in \mathbb{Z}$ ,  $b_0 = \pm 1$ ,  $C(t) = \frac{A(t)}{B(t)}$ , то  $c_n \in \mathbb{Z}$ .

*Доказательство.* Очевидно. □

*Пример.* Давайте получим  $2^n$ ! Что нам хочется?

$$P(t) = 1 + 2t + 4t^2 + \dots + 2^n t^n + \dots$$

Хм-м-м-м. Может, так:

$$P(t) = (2t)^0 + (2t)^1 + (2t)^2 + \dots + (2t)^n + \dots$$

Хм-м-м, кажется, геометрическая прогрессия.

$$P(t) = \frac{1}{1-2t}$$

Ок?

*Замечание.* Ну, очень хочется так думать, но вообще так жить некорректно, неверно интерпретировать  $t$  как число. Потому что если мы будем, то мы придём в мат. анализ и вспомним о том, что у рядов есть радиус сходимости, и если ряд условно сходится или расходится, то мы проиграли.

Но почему производящие функции использовались в мат. статистике? Потому что к ним часто применяется следующий приём: давайте сделаем то, что делать нельзя, получим что-то, а потом как-нибудь иным способом докажем, что наш ответ норм.

*Пример.* Мы получили

$$P(t) = \frac{1}{1-2t}$$

Давайте проверим, что подходит.

$$a_0 = 1, a_{k \geq 1} = 0 \quad b_0 = 1, b_1 = -2, b_{n \geq 2} = 0$$

$$c_0 = \frac{a_0}{b_0} = 1 \quad c_1 = \frac{a_1 - c_0 b_1}{b_0} = 2 \quad c_n = -c_{n-1} b_1 = 2c_{n-1}$$

Действительно, подходит.

*Замечание.* Почему мы так делаем вообще? Потому что у нас для некоторых  $t$  верна формула

$$(2t)^0 + (2t)^1 + (2t)^2 + \dots + (2t)^n + \dots = \frac{1}{1-2t}$$

И если бы мы получили иной ряд на самом деле при делении, мы бы получили, что указанная выше формула не верна нигде. Есть патологические примеры (см. математический анализ, функция со всеми нулевыми производными, не равная тождественно нулю), но в целом обычно получается жить в ситуации, когда мы нарушаем правила математики, а потом доказываем, что получили верный ответ.

*Пример.* Хорошо, давайте построим числа Каталана. Мы знаем, что

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$$

Та-а-а-ак, что-то знакомое. Пусть  $C(t)$  — числа Каталана. Тогда из формулы выше хотелось бы, чтобы получилось

$$A(t) = C(t)C(t)$$

Где  $A(t)$  — числа Каталана со сдвинутыми коэффициентами. Как нам сдвинуть коэффициенты? Умножить на  $t$ :

$$C(t) = C(t)C(t)t$$

Это почти хорошо, разве что тут нулевой коэффициент получится ноль, а нам надо 1:

$$C(t) = C(t)C(t)t + 1$$

Так, ну, хорошо, начинаем делать грязь:

$$C(t) = \frac{1 \pm \sqrt{1-4t}}{t}$$

Тут всё плохо. Тут и  $\pm$ , и корень формального ряда и деление на  $t$ , а на  $t$  делить нельзя т.к. у него нулевой коэффициент ноль.

Ну, делаем грязь дальше. Что делать с корнем? По Тейлору раскладывать, конечно же

$$(1+x)^\alpha = 1 + \frac{\alpha}{1}x + \frac{\alpha(\alpha-1)}{2}x^2 + \dots + \frac{\alpha(\alpha-1)\dots(\alpha-n+1)}{n!}x^n + \dots$$

Это мы так легко не докажем, но у нас  $\alpha$  конкретное ( $\frac{1}{2}$ ), и вот верность этой формулы для него доказать можно довольно легко.

$$\sqrt{1-4t} = 1 - \frac{1/2}{1}4t + \frac{(1/2)(-1/2)}{2}16t^2 - \frac{(1/2)(-1/2)(-3/2)}{6}64t^3 + \frac{(1/2)(-1/2)(-3/2)(-5/2)}{24}256t^4 + \dots$$

Хорошо, давайте попытаемся это посчитать. Получим

$$1 - 2t - 4t^2 - 10t^3 - 28t^4 - \dots$$

Теперь оставшиеся две проблемы:  $\pm$  и деление на  $t$ . Посмотрим на второе. Почему мы не хотели делить на  $t$ ? Потому что у нас был  $a_0$ , который не поделить на 0. Но если свободного члена нет, то будет логично делить на  $t$ . Так понятно, что нам надо взять  $\pm$  как  $-$ . И мы, как нетрудно заметить, получим числа Каталана.

*Замечание.* Хорошо, какие ещё у нас есть операции? Ну, интегрирование и дифференцирование. Например, что будет, если мы хотим каждый коэффициент умножить на его номер? Ну, очевидно, так:

$$B(t) = A'(t) \cdot t$$

**Определение 5.** Производной формального степенного ряда называется понятно, что.

**Свойство 5.1.** Несложно проверить формулы производной произведения и производной частного для формальных степенных рядов.

**Определение 6.** Интегралом формального степенного ряда называется также понятно, что, разве что константа интегрирования равна нулю. Обозначение:

$$\int A(t)$$

*Замечание.* Из-за последнего (конкретной константы интегрирования) интегралами пользуются довольно нечасто.

**Свойство 6.1.** Несложно проверить, что верна формула интегрирования по частям.

*Замечание.* Ну что, подставляем один ряд в другой?

Пусть есть

$$A(t) = a_0 + a_1t + a_2t^2 + \dots + a_nt^n + \dots$$

$$B(t) = b_0 + b_1t + b_2t^2 + \dots + b_nt^n + \dots$$

Тогда что такое  $A(B(t))$ ?

$$C(t) = A(B(t)) = a_0 + a_1(b_0 + b_1t + b_2t^2 + \dots + b_nt^n + \dots) + a_2(b_0 + b_1t + b_2t^2 + \dots + b_nt^n + \dots)^2 + \dots$$

Ну и что с этим делать? У нас даже свободный член нормально не считается, там получится  $a_0 + a_1b_0 + a_2b_0^2 + \dots$ . Это вообще какая-то сумма ряда, а это, во-первых, матан, во-вторых, радиусы сходимости и прочий ужас.

Нам не нравится  $b_0$ , пусть подставлять можно только ряд с  $b_0 = 0$ . Тогда  $c_0 = a_0$ . Чему равно  $c_1$ ? Ну, во второй скобке там степени не ниже квадрата. Значит

$$c_1 = a_1b_1$$

А  $c_2$ ? Ну,

$$c_2 = a_1b_2 + a_2b_1^2$$

Пока непонятно, давайте запишем  $c_3$ :

$$c_3 = a_1b_3 + a_2(b_1b_2 + b_2b_1) + a_3b_1^3$$

Кринж какой-то, но уже что-то более понятное.

**Определение 7.** Пусть

$$A(t) = a_0 + a_1t + a_2t^2 + \dots + a_nt^n + \dots$$

$$B(t) = b_1t + b_2t^2 + \dots + b_nt^n + \dots$$

Тогда **подстановкой формального степенного ряда  $B$  в ряд  $A$**  называется ряд

$$C(t) = c_0 + c_1t + c_2t^2 + \dots + c_nt^n + \dots \quad c_n = \sum_{k=0}^n a_k \sum_{n=i_1+\dots+i_k} \prod_{j=1}^k b_{i_j}$$

**Теорема 1.** Пусть дана последовательность  $a_n$ . Следующие три условия эквивалентны:

1.  $A = \frac{P}{Q}$ , где  $P$  и  $Q$  — многочлены.
2.  $a_n$  задаётся рекуррентным соотношением:

$$\forall n \geq m \quad a_n = c_1a_{n-1} + c_2a_{n-2} + \dots + c_ka_{n-k}$$

Где  $m \geq k$  — некоторые натуральные числа,  $c_j$  — некоторые вещественные числа.

3. Для некоторого  $n_0 \in \mathbb{N}$  выполнено

$$\forall n > n_0 \quad a_n = \sum_{i=1}^s p_i(n) r_i^n$$

Где  $s$  — некоторое натуральное число,  $r_i \in \mathbb{C}$ .

Доказательство.  $1 \rightarrow 2$  Известно, что

$$A(t) = \frac{p_0 + p_1 t + \dots + p_m t^m}{q_0 + q_1 t + \dots + q_k t^k}$$

Известно, что  $q_0 \neq 0$ , иначе нельзя делить. Далее Н.У.О.  $q_0 = 1$  (иначе поделим числитель и знаменатель на  $q_0$ ).

Давайте делить:

$$\begin{aligned} a_0 &= p_0 \\ a_1 &= p_1 - q_1 a_0 \\ &\dots \end{aligned}$$

Дальше у нас что-то кончится раньше, либо  $k$ , либо  $m$ . Пусть, например,  $k$ :

$$\begin{aligned} a_k &= p_k - q_1 a_{k-1} - q_2 a_{k-2} - \dots - q_k a_0 \\ a_{k+1} &= p_{k+1} - q_1 a_k - q_2 a_{k-1} - \dots - q_k a_1 - \underbrace{q_{k+1} a_0}_0 \\ &\dots \end{aligned}$$

Потом у нас кончится  $m$ . И дальше будет то же самое, но без  $p_n$ . А это как раз условие 2.

$2 \rightarrow 1$  Известно

$$A(t) = a_0 + a_1 t + a_2 t^2 + \dots$$

Домножим это на некоторые штуки:

$$\begin{aligned} A(t) &= a_0 + a_1 t + a_2 t^2 + \dots \\ A(t)t &= a_0 t + a_1 t^2 + a_2 t^3 + \dots \\ A(t)t^2 &= a_0 t^2 + a_1 t^3 + a_2 t^4 + \dots \end{aligned}$$

А теперь давайте возьмём нашу рекурренту и заменим  $a_{n-j}$  на  $A(t)t^j$ :

$$A(t) - c_1 A(t)t - c_2 A(t)t^2 - \dots - c_k A(t)t^k$$

У нас тогда по рекуррентному соотношению все коэффициенты при  $t^{>m}$  будут ноль. То есть разность получится равной какому-то многочлену  $m$ -той степени. Ну так извините, получим

$$A(t) = \frac{P(t)}{1 - c_1 t - c_2 t^2 - \dots - c_k t^k}$$

□

*Замечание.* Из этого мы можем решить такую задачу: пусть есть последовательность, заданная линейной рекуррентой. Нам надо узнать, нельзя ли уменьшить рекурренте порядок.

Ну, как мы выяснили, порядок рекурренты равен степени знаменателя, а значит надо представить нашу последовательность как частное двух последовательностей и сократить максимально сильно. Чтобы сократить, надо научиться искать НОД двух многочленов (алгоритмом Евклида, например).

*Замечание.* Другая задачка: возьмём рекурренту и найдём  $a_n$ .

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

Запишем вот такое (очевидно, выполненное) равенство:

$$\begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_{n-k+2} \\ a_{n-k+1} \end{pmatrix} = \begin{pmatrix} c_1 & c_2 & \dots & c_{k-1} & c_k \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{pmatrix} \begin{pmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_{n-k+1} \\ a_{n-k} \end{pmatrix}$$

назовём первое  $\vec{a}_n$ , второе —  $C$ , третье —  $\vec{a}_{n-1}$ , всё, что нам надо, быстро возвести  $C$  в степень. Но можно решить это за  $k^2 \log n$ .

*Пример.* Давайте возьмём числа Фибоначчи:

$$f_n = f_{n-1} + f_{n-2}$$

И мы знаем формулу

$$\frac{1}{1-t-t^2}$$

И домножим числитель и знаменатель на  $1+t+t^2$ :

$$\frac{1+t+t^2}{1-3t^2+t^4}$$

Если взять это и получить отсюда рекурренту, получим

$$f_n = 3f_{n-2} - f_{n-4}$$

А если сделать это ещё раз, получим

$$f_n = 7f_{n-4} - f_{n-8}$$

То есть нам нужно значительно меньше считать.

*Замечание.* Более глобально,

$$A(t) = \frac{P(t)}{Q(t)} = \frac{P(t)Q(-t)}{Q(t)Q(-t)} = \frac{P_0(t^2) + P_1(t^2)t}{Q_1(t^2)}$$

Таким образом можно решить задачку по поиску  $a_n$  за  $(m+k)k \log n$

*Замечание.* Теперь давайте разбираться с третьим утверждением теоремы.

**Лемма 1.** Если  $a_n = n^k r^n$ , то  $A$  — дробно-рациональный формальный степенной ряд.

*Доказательство.* Индукция по  $k$ . Для  $r^n$  знаем  $(\frac{1}{1-rt})$ , переход: пусть  $n^{k-1}r^n = \frac{P(t)}{Q(t)}$ , то чему равно  $n^k r^n$ ? Ну, возьмём производную и домножим на  $t$ . Получим, как нетрудно заметить, отношение многочленов.  $\square$

*Замечание.* На практике это не очень применимо т.к. очень быстро растёт степень знаменателя. Давайте докажем, что  $n^k r^n$  представим как

$$\frac{P_k(t)}{(1-rt)^{k+1}}$$

Понятно, что  $P_0 = 1$ , и всё хорошо. Переход индукции:

$$\left( P_k(t) \frac{1}{(1-rt)^k} \right)' t = P_k'(t) \frac{t}{(1-rt)^k} + P_k(t) \frac{t}{(1-rt)^{k+1}}$$

*Доказательство.* Мы уже доказали, что из 1 следует 3 в теореме. Докажем обратно. Воспользуемся основной теоремой алгебры. Пусть  $\deg Q = k$ , тогда у него  $k$  корней с учётом кратности. Не умаляя общности  $Q = 1 - c_1t - c_2t^2 - \dots - c_kt^k$ . Пусть корни  $Q$  равны  $x_i$  и кратность у них  $s_i$ . Тогда  $\frac{P(t)}{Q(t)}$ , как мы знаем из математического анализа, можно разложить на простые дроби:

$$\frac{P(t)}{Q(t)} = \sum_{i=1}^l \frac{P_i(t)}{(1 - r_it)^{s_i}}$$

То есть нам достаточно доказать, что  $\frac{t^m}{(1-rt)^s}$  представим как  $a_n = p(n)r^n$ . Давайте сначала разберёмся для  $m = 0$ . И разберёмся индукцией по  $s$ . Для  $s = 1$  мы знаем,  $p(n) \equiv 1$ . Что для больших  $s$ ? Ну, продифференцируем нашу формулу

$$B'(t) = \left( \frac{1}{(1-rt)^s} \right)' = \frac{sr}{(1-rt)^{s+1}}$$

То есть  $\frac{1}{(1-rt)^{s+1}} = \frac{B'(t)}{sr}$ . И если раньше мы имели  $p(n)r^n$ , то теперь имеем  $\frac{p(n+1)(n+1)r^{n+1}}{sr}$ , и это то, что нам надо.

А если брать  $t^m$  в числителе, то получим  $a_n = p(n-m)r^{-m}r^n$ , если  $m \leq n$ .  $\square$

*Замечание.* Что, кстати, видим, так это то, что  $\deg p = k - 1$ .

*Замечание.* Также, как несложно заметить, асимптотика роста последовательности зависит только от наименьшего по модулю корня знаменателя. Остаётся лишь вопрос, что, если таковых несколько? Ну, рассмотрим

$$\frac{1}{1-t^2}$$

Если посмотреть на члены этой последовательности, то там чередуются нули и единицы. А в более общем случае, если взять

$$M = \text{lcm} \left( \frac{2\pi}{\arg r_i} \right)$$

То последовательность распадётся на  $M$  независимых, каждая из которых растёт с асимптотикой  $u^n (e^{i\varphi_i(n \% M)})$ , где  $u$  — модуль  $r_i$ .

## Использование производящих функций для работы с комбинаторными объектами.

*Замечание.* Давайте возьмём прямоугольник  $2 \times n$  и просуммируем все способы его замостить. И ещё для всех  $n$  просуммируем.

Все замещения, кроме пустого, начинаются либо с вертикальной доминошки, либо с двух горизонтальных. Давайте сгруппируем так слагаемые и вынесем за скобки начало. Нетрудно заметить, что в скобках все возможные замощения. Итого имеем, что сумма всех замощений  $S$  — пустое замощение плюс  $S$  на сумму вертикальной доминошки и горизонтальной в квадрате. Умеем ли мы делить один на разность пустого замощения и суммы вертикальной доминошки и горизонтальной в квадрате? Ну, почему нет. Получится сумма всех замощений, это мы уже знаем.

Но смотрите. Мы получили буквально формальные степенные ряды, если заменить количество доминошек на степень. И мы имеем, что фигура из  $k$  доминошек имеет «вес»  $d^k$ , и по сути мы тут считаем комбинаторные объекты заданного веса.

То же самое можно делать и для подвешенных двоичных деревьев, для любых комбинаторных объектов по сути. Но возникает вопрос: зачем? А на самом деле, тут мы используем производящие функции для работы с комбинаторными объектами.

*Замечание.* Что у нас по сути есть? У нас есть некоторая неделимая штука — «атом». И мы определённым образом производим действия с этими атомами, после чего получается объект с каким-то количеством атомов или «весом». А интересно нам в этой теории количество комбинаторных объектов заданного веса. Ну, эти количества задают какую-то последовательность, а значит можно сделать производящую функцию.

И как же эту функцию получить? Тут нам как раз поможет формальная сумма объектов. Если их просуммировать, а потом взять расчленив каждый на атомы, получится сумма  $u^{n_k}$ . И после приведения подобных как раз получится формальный степенной ряд.

**Определение 8.** Дизъюнктивным объединением комбинаторных объектов  $A$  и  $B$  называется комбинаторный объект, производящая функция которого равна сумме формальных степенных рядов  $A$  и  $B$ .

*Замечание.* Почему? Потому что если у нас есть два множества комбинаторных объектов не пересекаются, и объединение имеет своим весом сумму весов этих множеств.

**Определение 9.** Парой комбинаторных объектов  $A$  и  $B$  называется комбинаторный объект с производящей функцией, равным произведению производящих функций  $A$  и  $B$ .

*Замечание.* Опять же, почему? Ну, если считать вес пары равным сумме весов её компонент, то получится именно что

$$c_n = \sum_{i=0}^n a_i b_{n-i}$$

*Замечание.* Теперь давайте сделаем последовательность комбинаторных объектов  $A$ . Во-первых, если в  $A$  есть объект веса 0, его можно вставить куда угодно в список в любом количестве, а значит объектов любого веса будет бесконечное количество. Плохо.

Пусть не так. Тогда что будет? Вспомним Clojure и скажем, что у нас есть пустой список (он один), а если список не пуст, то у него есть хвост и голова. Голова — это  $A$ , хвост — это  $\text{Seq } A$ . Итого

$$\text{Seq } A = 1 + A \times \text{Seq } A \Leftrightarrow \text{Seq } A = \frac{1}{1 - A}$$

**Определение 10.** Пусть  $A$  — комбинаторный объект. Тогда  $\text{Seq } A$  — комбинаторный объект, производящая функция которого задаётся как

$$\frac{1}{1 - A}$$

*Пример.* Начинаем веселиться. Пусть у нас есть не один атом, а два. Разных. Ну, ничего интересного,  $B = \{u_1, u_2\}$  имеет производящую функцию  $B(t) = 2t$ . И дальше ничего концептуально нового. Можно рассмотреть  $\text{Seq } B$  и получить  $\frac{1}{1-2t}$ .

*Пример.* Можно рассмотреть что-нибудь концептуально ещё более сложное, например,  $A = \{u; (u; u)\}$ . Это  $t + t^2$ . Тогда  $\text{Seq } A = \frac{1}{1-t-t^2}$ . Откуда же тут числа Фибоначчи? Да понятно, откуда, у нас либо вертикальная доминошка  $(u)$ , либо пара горизонтальных  $(u; u)$ . Это ровно тот пример, что мы рассматривали изначально.

*Замечание.* Дальше мы хотим множество всех подмножеств. Понятно, что это, но непонятно, как возвести 2 в степень ряда. Ну, смотрите. Рассмотрим наши комбинаторные объекты  $A$ . Это какие-то  $A_1, A_2$  и так далее. Мы можем взять или не взять первый элемент. Это  $1 + A_1$ . Можем взять или не взять  $A_2$ :  $1 + A_2$ . Итого

$$\prod_{a \in A} (1 + a) = \prod_{a \in A} (1 + t^{w(a)}) = \prod_{k=0}^{\infty} (1 + t^k)^{a_k}$$

**Определение 11.** Множеством всех подмножеств (powerset) комбинаторного объекта  $A(t) = a_0 + a_1 t + \dots$  называется комбинаторный объект, производящая функция которого равна

$$\prod_{k=0}^{\infty} (1 + t^k)^{a_k}$$

*Замечание.* Теперь хочется мульти-множество. Мы можем взять каждый объект несколько раз. Тут, очевидно, тоже не может быть объектов веса ноль. Можем взять первый объект любое количество раз, второй — тоже и так далее. Получается

$$(1 + A_1 + A_1^2 + \dots) (1 + A_2 + A_2^2 + \dots) \dots$$



Что в итоге получается?

$$\frac{1}{1-A_1} \frac{1}{1-A_2} \dots$$

**Определение 12.** Мульти-множеством (multiset) комбинаторного объекта  $A(t) = a_0 + a_1 t + \dots$  называется комбинаторный объект, производящая функция которого равна

$$\prod_{k=1}^{\infty} \left( \frac{1}{1-t^k} \right)^{a_k}$$

**Утверждение.**

$$\text{MSet } A = \text{PSet}(\text{Seq } A - 1)$$

*Замечание.* Пусть  $A$  — комбинаторный объект. Давайте делать циклы из  $A$ . Как устроен цикл? Как список, но с точностью до циклического сдвига.

Мы знаем, что  $\text{Seq } A = 1 + A + A^2 + \dots$ . Понятно, что  $A^k$  не зависят друг от друга, значит можно рассмотреть  $A^k$  отдельно. Рассмотрим. Это кортеж из  $k$  элементов  $A$ . Воспользуемся леммой Бернсайда. Пусть  $I_{n,k,i}$  — количество массивов длины  $k$  веса  $n$ , в которых  $i$  — неподвижная точка. Тогда

$$C_{k,n} = \frac{1}{k} \sum_{i=0}^{k-1} I_{n,k,i}$$

$$I_{n,k,i} = \begin{cases} 0 & n \not\equiv \frac{k}{\gcd(k,i)} \\ A_{\frac{n \gcd(k,i)}{k}}^{\gcd(k,i)} & n \equiv \frac{k}{\gcd(k,i)} \end{cases}$$

Тогда  $C_n$  — это сумма  $C_{n,k}$ . Говорят, что все эти три штуки можно упростить до

$$C(t) = \sum_{k=1}^{\infty} \frac{\varphi(k)}{k} \ln \frac{1}{1-A(t^k)}$$

Где  $\varphi$  — функция Эйлера.

*Замечание.* Здесь дыра в сюжете.

### Производящие функции Дирихле.

*Замечание.* Сегодня в качестве формальной переменной мы будем использовать  $s$ , а не  $t$

**Определение 13.** Производящая функция Дирихле для последовательности  $a_1, a_2, \dots$  — это следующий формальный ряд

$$A(s) = \sum_{n=1}^{\infty} \frac{a_n}{n^s}$$

*Пример.* Начнём с последовательности  $1, 0, 0, \dots, 0, \dots$ . Для неё  $A(s) = 1$ .

*Пример.*  $1, 1, \dots, 1, \dots$ . Тогда  $A(s) = \zeta(s)$  — дзета-функция Римана.

*Пример.*  $1, 2, 3, 4, \dots$ . Тогда  $A(s) = \zeta(s-1)$ .

**Утверждение.** Если  $k \in \mathbb{Z}, b_n = n^k a_n$ , то  $B(s) = A(s-k)$ .

**Утверждение.** Несложно заметить, что сумма производящих функций Дирихле — производящая функция Дирихле суммы рядов.

*Замечание.* Отсюда мы легко умеем умножать производящие функции Дирихле на многочлен

*Замечание.* Пусть  $C(s) = A(s)B(s)$ , то как  $c_n$  зависит от  $a_n$  и  $b_n$ ?

$$C(s) = \left( \sum_{n=1}^{\infty} \frac{a_n}{n^s} \right) \left( \sum_{k=1}^{\infty} \frac{b_k}{n^s} \right) = \sum_{n=1}^{\infty} \sum_{k=1}^{\infty} \frac{a_n b_k}{(nk)^s} = \sum_{n=1}^{\infty} \sum_{m|n} \frac{a_n b_{m/n}}{n^s}$$

**Следствие 1.1.** Отсюда если  $A(s)\zeta(s) = B(s)$ , то  $b_n = \sum_{d|n} a_d$ .

**Следствие 1.2.** Отсюда последовательность  $d_n$  — количество делителей  $n$  соответствует производящей функции  $\zeta^2(s)$ .

*Замечание.* Давайте научимся делить. Попробуем взять  $\frac{1}{\zeta(s)}$ . Обозначим это за  $\mu(s)$ .

$$\sum_{d|1} \mu_d = 1 \Rightarrow \mu_1 = 1$$

$$\sum_{d|2} \mu_d = 0 \Rightarrow \mu_2 = -1$$

$$\sum_{d|3} \mu_d = 0 \Rightarrow \mu_3 = -1$$

$$\sum_{d|4} \mu_d = 0 \Rightarrow \mu_4 = 0$$

Заметим, что это единственным образом определяет  $\mu$ . Мы получим  $\mu_5 = -1, \mu_6 = 1$ . Для простых мы видим, что оно  $-1$ . А для составных непонятно.

**Теорема 2.** Пусть  $\mu(s) = \frac{1}{\zeta(s)}$ . Тогда

$$\mu_k = \begin{cases} 0 & \exists p \ p^2 | k \\ (-1)^\nu & k = p_1 p_2 \cdots p_\nu, p_i \text{ различные} \end{cases}$$

*Доказательство.* Докажем, что  $\mu$ , определённое той формулой подходит под  $\zeta(s)\mu(s) = 1$ . В  $n = 1$  понятно, пусть  $n > 1$ . Хотелось

$$0 = \sum_{d|n} \mu_d$$

Пусть  $n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ . Если квадрат есть, понятно, иначе пусть мы выбрали  $j$  простых множителей.

Сколько способов их выбрать?  $\binom{k}{j}$ . Получим

$$\sum_{d|n} \mu_d = \sum_{j=0}^k \binom{k}{j} (-1)^j = (1-1)^k = 0$$

□

**Определение 14.** Функция  $\mu(s)$  называется **функцией Мёбиуса**.

**Утверждение** (Формула обращения Мёбиуса). Пусть  $B(s) = A(s)\zeta(s)$ . Чтобы найти  $A$ , можно домножить обе части равенства на  $\mu$  и получить

$$a_n = \sum_{d|n} b_d \mu_{n/d}$$

**Определение 15.** Произведение  $\prod_{p \in \mathbb{P}} \sum_{k=0}^{\infty} \frac{1}{p^{ks}}$  называется **произведением Эйлера**.

**Теорема 3.**

$$\prod_{p \in \mathbb{P}} \sum_{k=0}^{\infty} \frac{1}{p^{ks}} = \zeta(s)$$

*Доказательство.* Что это за штука? Ну,

$$\left(1 + \frac{1}{2^s} + \frac{1}{4^s} + \dots\right) \left(1 + \frac{1}{3^s} + \frac{1}{9^s} + \dots\right) \left(1 + \frac{1}{5^s} + \frac{1}{25^s} + \dots\right) \dots$$

Ну, что у нас будет в начале? Ну, 1. Потом мы получим  $\frac{1}{2^s}$ ,  $\frac{1}{3^s}$  и  $\frac{1}{5^s}$ . Если взять несколько не-единиц, будет  $\frac{1}{6^s}$ ,  $\frac{1}{10^s}$ ,  $\frac{1}{15^s}$ . Ну, о'кей, а зачем всё это? А затем, что мы получим любое  $\frac{1}{n^s}$ , притом ровно один раз, потому что  $n$  единственным образом можно разложить на простые и взять единственным образом эти простые из скобок.  $\square$

*Замечание.* Давайте вот на что посмотрим:

$$\prod_{p \in \mathbb{P}} \sum_{k=0}^{\infty} \frac{1}{p^{ks}} = \prod_{p \in \mathbb{P}} \sum_{k=0}^{\infty} \frac{1}{(p^s)^k}$$

Это геометрическая прогрессия! Она равна

$$\prod_{p \in \mathbb{P}} \sum_{k=0}^{\infty} \frac{1}{(p^s)^k} = \prod_{p \in \mathbb{P}} \frac{1}{1 - p^{-s}}$$

На что надо умножить это, чтобы получилась единица?

**Следствие 1.1.**

$$\mu(s) = \prod_{p \in \mathbb{P}} \left(1 - \frac{1}{p^s}\right)$$

**Определение 16.** Будем обозначать взаимно простые числа значком  $\perp$ .

**Определение 17.** Рассмотрим  $f: \mathbb{N} \rightarrow \mathbb{R}$  **теоретико-численно мультипликативной** (далее просто мультипликативной), если

$$\forall u, v : u \perp v \quad f(uv) = f(u)f(v)$$

Последовательность называется мультипликативной, если она мультипликативна как функция индексов.

*Пример.*  $f(u) = 1$ , хорошая функция.

*Замечание.* Пусть  $a_n$  — мультипликативная последовательность. Пусть  $n = p_1^{\alpha_1} p_2^{\alpha_2} \dots$ . Тогда несложно заметить, что достаточно узнать  $a_{p_i^{k_i}}$ .

**Теорема 4.** Если  $a_n$  мультипликативно, то

$$A(s) = \prod_{p \in \mathbb{P}} \sum_{k=0}^{\infty} \frac{a_{p^k}}{p^{ks}}$$

**Определение 18.**  $\sigma_k(s) = \sum_{d|n} d^k$  называется **теоретико-числовыми моментами числа**.

*Пример.*  $\sigma_0(n)$  — количество делителей числа  $n$ . Чему равно  $\Sigma_0(s)$ ?

$$\Sigma_0(s) = \prod_{p \in \mathbb{P}} \sum_{k=0}^{\infty} \frac{k+1}{p^{ks}} = \prod_{p \in \mathbb{P}} \underbrace{\sum_{k=0}^{\infty} \frac{k+1}{(p^s)^k}}_{\frac{1}{(1-p^{-s})^2}} = \prod_{p \in \mathbb{P}} \frac{1}{(1-p^{-s})^2} = \zeta(s)^2$$

*Пример.* Рассмотрим функцию Эйлера  $\phi(n)$  — количество натуральных чисел, взаимно простых с  $n$  и меньших его.

$$\begin{aligned}\Phi(s) &= \prod_{p \in \mathbb{P}} \left( 1 + \sum_{k=1}^{\infty} \frac{p^k - p^{k-1}}{p^{ks}} \right) = \prod_{p \in \mathbb{P}} \left( 1 + \frac{p^1 - p^0}{p^s} + \frac{p^2 - p^1}{p^{2s}} + \dots \right) = \\ &= \prod_{p \in \mathbb{P}} \left( \underbrace{1 + \frac{p^1}{p^s} + \frac{p^2}{p^{2s}} + \dots}_{\sum_{k=0}^{\infty} \frac{1}{p^{k(s-1)}}} + \underbrace{-1 - \frac{p^0}{p^s} - \frac{p^1}{p^{2s}} - \dots}_{-\frac{1}{p} \sum_{k=1}^{\infty} \frac{1}{p^{k(s-1)}}} \right) = \\ &= \prod_{p \in \mathbb{P}} \frac{1 - p^{-s}}{1 - p^{-s+1}} = \mu(s) \zeta(s-1) = \frac{\zeta(s-1)}{\zeta(s)}\end{aligned}$$

## 2 Теория вычислимости.

*Замечание.* Глобально мы хотим анализировать как-то алгоритмы. А для этого надо формализовать понятие «программы» и «алгоритма». Ну, это достаточно просто. У нас есть какие-то символы  $\Pi$  и некоторое подмножество строк из них — корректные программы. Какие — пока не так важно. А зафиксируем то, что программа принимает на вход какой-то  $x$  — строку символов из  $\Sigma$  и возвращает что? Ну, мы для простоты будем возвращать истину или ложь. Этого нам хватит. Если программа крашится — будем считать, что это ложь.

Но ведь программа может никогда не завершиться. И это уже какой-то другой фундаментальный случай, его нельзя проэмулировать и узнать.

**Определение 1.** Будем считать, что **язык программы**  $(L(p))$  — множество слов, на которых она не зависает и выдаёт 1.

**Утверждение.** Несложно заметить, что  $L(p)$  — функция из программ в языки. И что она не инъективна (`return false;` и `while (true) {}` дают пустой язык, хотя программы разные). И что по соображениям мощности эта функция не сюръективна. Более того,  $2^{\Sigma^*}$  (все языки) — несчётное множество, а множество программ счётно.

*Замечание.* Впрочем, как мы обсуждали ранее, у невозможности задания почти всех языков программой есть и обратная сторона. Очень трудно описать несчётное количество языков счётным количеством строк на русском языке.

**Определение 2.** Язык называется **рекурсивно-перечислимым**, если он распознаётся некоторой программой. Множество таких языков —  $RE$ . Более современное название — просто **перечислимые** или **полуразрешимые**.

*Замечание.* Почему **полу**-разрешимые? Потому что программа умеет зависать, то есть мы нормально живём только в том случае, если слово есть в языке.

**Определение 3.** Язык называется **рекурсивным** (устаревшее) или **разрешимым** (современное), если существует программа, которая никогда не зависает и возвращает истину только на словах этого языка.

*Замечание.* Так. У нас есть два алфавита. Зачем? Давайте всё будет в одном алфавите. Компьютер нормально живёт с битовыми строками, и всё у него хорошо. Так что отныне  $\Sigma = \Pi$ .

*Замечание.* Ещё одно допущение. Зачем нам считать, что не любая программа корректна? Давайте любая программа будет корректна, а если нет, то подставим вместо него программу, которая всегда зависает.

*Замечание.* Так, хорошо. Ещё иногда удобно считать, что программа принимает не строку, а неотрицательное целое число. Между  $\mathbb{N}_0$  и  $\Sigma^*$  можно создать простую биекцию: градуированный лексикографический порядок.

**Определение 4.** Градуированный лексикографический порядок — упорядочивание сначала по длине, а среди равных длин — лексикографически. Он (как биекция между  $\Sigma^*$  и  $\mathbb{N}_0$ ) называется **естественным изоморфизмом**.

*Замечание.* Здесь и далее  $\mathbb{N}_0$ ,  $\Sigma^*$  и множество программ можно использовать рандомно, они все изоморфны, а значит можно говорить о программах, как о числах, о вводе как и программе и т.п.

**Определение 5.** Номер программы в естественном изоморфизме — **номер Гёделя**.

*Замечание.* Хотя вообще номер Гёделя — что-то более комплексное т.к. более глобально нам хочется сделать программу, которая считает число, конвертирует его в программу и запустит её. И вот в **любой** такой нумерации номера называются номерами Гёделя. Наш пример — именно такая нумерация.

*Замечание.* Всё это, конечно, хорошо, но что такое программа? А на самом деле не так важно. Есть два подхода к этому. Можно формально определить, что такое программа, углубиться в абстрактные вычислители и прочее. А второе — программистский здравый смысл. Мы с вами программисты, знаем кучу всего вообще, давайте считать, что у нас есть языки программирования есть, и нам разве что пофиг, что у нас ограниченная память, ограниченные целые числа и т.п. Но на что-то забыть всё же нельзя. На точность вещественных чисел, например, иначе парадоксы получатся.

Очень хорошо, а верно ли, что наши два подхода эквивалентны? Ну, вообще вопрос некорректный, потому что слева математический объект, а справа хрень какая-то, но если и на это забыть, то да, эквивалентны. Это **тезис Тьюринга — Чёрча**. По модулю ограниченности памяти наши штуки эквивалентны.

*Замечание.* Почему полуразрешимый язык называется перечислимым?

**Определение 6.** Язык называется **перечислимым**, если существует программа, которая, будучи запущенной на пустом вводе выводит все слова языка (возможно, за бесконечное время).

*Замечание.* Тут что-то схожее с производящими функциями. Мы не можем описать заданный член формального ряда, но можем описать первые  $n$ , когда они заданы рекуррентно. Ещё есть вопрос в том, что значит «выводит», как оно их разделяет, но это всё не важно.

*Пример.* Десятичные записи простых чисел перечислимы. Понятно, как их перечислить. Напишем простую проверку, является ли число простым и просто в **for** запишем её вызов и печать.

**Теорема 1.** Перечислимость и полуразрешимость эквивалентны.

*Доказательство.* Давайте полуразрешим перечислимый язык. Давайте возьмём перечислитель и всё, что он напечатает, перехватим и будем поочерёдно сравнивать с пришедшим нам словом.

Давайте наоборот. Заметим, что перебрать все слова и проверить каждое не получится (проверка может зависнуть). Давайте запускать программу с ограничением на время работы и сначала переберём все слова с ограничением на время работы  $TL = 1$ , потом  $TL = 2$  и т.д. Правда, тут вложенные бесконечные циклы, и в итоге до  $TL = 2$  мы не доберёмся. Поэтому давайте напомним вот что:

```
for TL in range(1, infinity):
    for x in map(word_by_number, range(TL)):
        if P(x):
            print(x)
```

Что мы тут использовали? Что мы можем вставить одну программу в другую, причём можно ещё и ограничить каким-то количеством шагов. □

**Теорема 2.** Любой разрешимый язык полуразрешим, но не обратное.

*Пример.* Рассмотрим вот такой язык: пусть  $U = \{(p; x) \mid p(x) = 1\}$  — множество пар из программы и ввода, что ввод удовлетворяет программе. Понятно, что он полуразрешим (запустим  $p(x)$ ). Но почему он не разрешим? От противного, пусть есть программа  $q$ , которая его разрешает. Сделаем тогда программу  $r(x) = \neg q(x; x)$ . Эта штука также никогда не зависает. Чудно. Что такое  $r(r)$ ? Если это 1, то  $q(r; r) = 0 \Rightarrow r(r; r) \neq 1 \Rightarrow r(r) \neq 1$ . Если 0, то  $q(r; r) = 1$ , а значит  $r(r) = 1$ . Ой.

**Теорема 3.** Если  $A$  и  $\bar{A}$  перечислимы, то  $A$  разрешим.

*Доказательство.* Ну, один из этих двоих не зависнет. А значит вводим  $TL$  и ждём того, кто первый пройдёт. В программировании можно воспользоваться тезисом Тьюринга — Чёрча и запустить два потока исполнения.  $\square$

**Утверждение.** Существуют неперечислимые языки.

*Доказательство.* Возьмём  $A = U$  — перечислим, но не разрешим. Тогда  $\bar{U}$  по теореме выше неперечислим.  $\square$

*Замечание.* А можно ли, чтобы программы не зависали? Ну, мы ещё выясним, что нельзя выполнить три условия: программы не зависят, не-зависающие программы эквивалентны обычными и чтобы по программе можно было понять, что она выводит.

**Определение 7.** Рассмотрим последовательность программ  $\{p_n\}$ . Говорят, что **последовательность программ вычислима**, если существует программа  $V(i; x)$ , которая по номеру  $i$  и входу  $x$  возвращает  $p_i(x)$ .

**Теорема 4.** Пусть  $\{p_n\}$  — последовательность программ. Тогда выполнено хотя бы одно из трёх:

1. Существуют  $i$  и  $x$  такие что  $p_i(x)$  зависит.
2. Существует такая программа  $q$ , что она не зависит ни на одном входе и для любого  $i$  существует  $x_i$  такое что  $q(x_i) \neq p_i(x_i)$
3.  $\{p_i\}$  не вычислима.

*Доказательство.* Пусть не так. Рассмотрим  $q(x) = \neg V(x; x)$  (поскольку  $\{p_i\}$  вычислима). Тогда, поскольку  $p_i$  не зависят,  $q$  она подходит под 2.  $\square$

**Утверждение** (Задача останова).  $HALT = \{(p; x) \mid p \text{ не зависит на } x\}$  не разрешим.

*Доказательство.* Ну, пусть мы разрешили задачу останова программой  $h$ . Пусть  $q(x)$  — программа, которая выглядит так:

```

if (h(x, x))
    while (true)
        ;
else
    return 0;

```

Понятно, что попытка проанализировать  $(q; q)$  приведёт нас к противоречию.  $\square$

**Определение 8.** Функция  $f: A \rightarrow \Sigma^*$  называется **вычислимой**, если существует такая программа  $p$ , что  $\forall x \in A \ f(x) = p(x)$  и  $\forall x \notin A \ p$  зависит.

**Определение 9.** Всюду определённая функция — та, у которой область определения равно  $\Sigma^*$ .

**Определение 10.** Говорят, что  $A$  **m-сводится** к  $B$ , если существует всюду определённая вычислимая функция  $f$  такая что  $x \in A \Leftrightarrow f(x) \in B$ . Обозначение:  $A \leq_m B$

*Замечание.* То есть все слова из  $A$  отображаются в язык  $B$ , а не из  $A$  — вне  $B$ . При этом функция нигде не зависит.

*Замечание.* «m» исторически расшифровывается как many-to-one, а современно как mapping.

**Теорема 5.** Если  $B$  разрешим и  $A \leq_m B$ , то  $A$  разрешим.

*Доказательство.* Ну, тривиально, нам нужна композиция функции и разрешителя  $B$ .  $\square$

**Теорема 6.** Если  $B$  перечислим и  $A \leq_m B$ , то  $A$  перечислим.

**Теорема 7.** Если  $A$  не разрешим/не перечислим,  $A \leq_m B$ , то  $B$  не разрешим/не перечислим.

**Теорема 8.**  $U \leq_m HALT$ .

*Доказательство.* Пусть  $f$  преобразует программу  $p$  и ввод  $x$  в

```
q = function(x)
  return "if (p(" + x + ") != 1)
    while (true)
      ;
    else
      return 1"
return (q, x)
```

□

**Утверждение.** Пусть  $HALT_\varepsilon$  — множество программ, не зависящих на  $\varepsilon$ . Тогда

$$HALT \leq_m HALT_\varepsilon$$

*Доказательство.* Программа сведения выглядит так:

```
return "function(unused) p(" + x + "), return 0"
```

□

**Определение 11.**  $FINITE = \{p \mid \exists \text{конечное число } x \ p(x) = 1\}$

**Утверждение.**  $HALT \leq_m \overline{FINITE}$ .

**Теорема 9** (Теорема Успенского — Райса). Никакое нетривиальное свойство перечислимых языков не разрешимо.

**Определение 12.**  $A \subset RE$  ( $RE$ , напомним, множество полуразрешимых языков) называется **свойством**.

**Определение 13.** Множество программ, распознающих языки с заданным свойством — **язык свойства**.

**Определение 14.** Отсюда мы сразу понимаем, что такое **разрешимое свойство** — то, язык которого разрешим.

**Определение 15.** Свойство **тривиально**, если оно равно  $RE$  или  $\emptyset$ .

*Доказательство.* Пусть  $A$  — нетривиальное свойство. То есть есть язык, который ему принадлежит, а есть тот, который не. Возьмём пустой язык. Не умаляя общности, пусть он не принадлежит  $A$ . И есть  $T \in A$ . У  $T$  есть полуразрешитель  $\text{in}T$ . Пусть  $A$  разрешим. Тогда есть программа  $\text{in}A(p)$ , которая проверяет, верно ли, что  $L(p) \in A$ . Напишем такую программу:

```
inU(p, x):
  q = "function(y)
    if (p(" + x + ") == 1)
      return inT(y)
    else
      return 0"
  return inA(q)
```

Утверждается, что это разрешитель универсального языка. Если  $p(x) = 1$ , то  $q(y) \equiv \text{in}T(y)$ . В противном случае  $q(y) = 0$ . Если  $p(x)$  зависит, то  $q(y)$  тоже зависит. Итого, если  $p(x) = 1$ , то  $q$  полураспознаёт  $T$ , а значит  $L(q) = T \in A$ . В противном случае  $q$  полураспознаёт пустой язык,  $L(q) = \emptyset \notin A$ . То есть  $p(x) = 1 \Leftrightarrow L(q) \in A \Leftrightarrow \text{in}A(q) = 1$ .  $\square$

*Замечание.* Здесь очень важное слово — «языков». Не «программ». Если вы прикручиваете свойство к программе, то теорема перестаёт работать.

*Замечание.* Сегодня на повестке дня «как написать программу, которая выводит свой собственный код, и зачем».

напишем простую программу, которая печатает ничего:

```
#include <iostream>

int main() {
    std::cout << "";
    return 0;
}
```

Идея первая — скопировать этот код внутрь кавычек

```
#include <iostream>

int main() {
    std::cout << "#include <iostream>

    int main() {
        std::cout << "";
        return 0;
    }";
    return 0;
}
```

Но он помимо к тому, что не работает, ещё и не компилируется. Ладненько, давайте поборемся со вторым, научимся экранировать:

```
std::string escape(const std::string& s) {
    std::string result;
    for (char c : s) {
        switch (c) {
            case '\n':
                result += "\\n";
                break;
            case '\"':
                result += "\\\"";
                break;
            case '\\':
                result += "\\\"";
                break;
            case '\\':
                result += "\\\"";
                break;
            default:
                result += c;
                break;
        }
    }
    return result;
}
```



```

    }
}
return result;
}

```

Давайте через эту штуку пропустим нашу программу. Получим что-то такое:

```
#include <iostream>\n#include <string>\n\nstd::string escape(const std::string& s) {\n    st
```

Теперь просто засунем это на место строки в исходный код:

```
#include <iostream>

std::string escape(const std::string& s) {
    std::string result;
    for (char c : s) {
        switch (c) {
            case '\\n':
                result += "\\n";
                break;
            case '\\\"':
                result += "\\\"";
                break;
            case '\\\'':
                result += "\\\'";
                break;
            case '\\\\':
                result += "\\\"";
                break;
            default:
                result += c;
                break;
        }
    }
    return result;
}

int main() {
    std::cout << "#include <iostream>\n#include <string>\n\nstd::string escape(const std::st
    return 0;
}

```

Ну, отличный план, только в том, что мы печатаем написано `std::cout << ""`;, а у нас на месте этого какая-то упоротая строка. Но как ни странно, эта проблема решается. В нашей текущей программе уже написано то, что мы должны вставить вместо `std::cout << ""`;. Давайте наша программа просто сделает то, что сделали мы: возьмёт этот исходный код, пропустит его через `escape` и подставит его на место `std::cout << ""`;

Обозначим за `$` особый спецсимвол, которым мы покажем, где у нас находится `std::cout << ""`;, чтобы знать, куда вставлять. Напишем пока что-то такое:

```
#include <iostream>
#include <string>

```

```
std::string escape(const std::string& s) {
    std::string result;
    for (char c : s) {
        switch (c) {
            case '\n':
                result += "\\n";
                break;
            case '\"':
                result += "\\\"";
                break;
            case '\\':
                result += "\\\"";
                break;
            case '\\\\':
                result += "\\\"";
                break;
            default:
                result += c;
                break;
        }
    }
    return result;
}

std::string get_other_source() {
    return "$";
}

std::string get_source() {
    std::string s = get_other_source();
    std::string r = escape(s);
    s.replace(s.find('$'), 1, r);
    return s;
}

int main() {
    std::cout << get_source();
    return 0;
}
```

Теперь пропустим нашу программу через `escape` и вместо доллара поставим результат:

```
#include <iostream>
#include <string>

std::string escape(const std::string& s) {
    std::string result;
    for (char c : s) {
        switch (c) {
            case '\n':
                result += "\\n";
                break;
            case '\"':
                result += "\\\"";
                break;
            case '\\':
                result += "\\\"";
                break;
            case '\\\\':
                result += "\\\"";
                break;
            default:
                result += c;
                break;
        }
    }
    return result;
}
```

```

        break;
    case '\\':
        result += "\\\";
        break;
    case '\\\\':
        result += "\\\";
        break;
    default:
        result += c;
        break;
    }
}
return result;
}

std::string get_other_source() {
    return "#include <iostream>\n#include <string>\n\nstd::string escape(const std::string&
}

std::string get_source() {
    std::string s = get_other_source();
    std::string r = escape(s);
    s.replace(s.find('$'), 1, r);
    return s;
}

int main() {
    std::cout << get_source();
    return 0;
}

```

Оно работает!

Заметим, что нам не надо именно выводить код. Мы можем, например, посчитать количество символов в исходном коде или что-нибудь ещё. То есть схема работы всегда одинаковая. Пишем **escape**, **get\_other\_source** и **source**, они одинаковые везде. После этого пишем в **main**'е то, что мы хотим сделать с исходным кодом. Когда мы кончили, пропускаем код программы через **escape** и вставляем вместо доллара.

**Теорема 10** (Теорема о рекурсии, теорема о рефлексии). *Для любой вычислимой функции двух аргументов  $V(x; y)$  существует вычислимая функция одного аргумента  $p$ , что  $\forall y \ V(p; y) = p(y)$ .*

*Замечание.* Что происходит? А вот что: программа может использовать свой исходный код (любым образом).

Доказывать эту теорему мы не будем, а дальше рассмотрим применения этой теоремы.

**Теорема 11.** *Универсальный язык не разрешим (снова). Но теперь мы докажем это при помощи теоремы о рекурсии.*

*Доказательство.* Пусть разрешим, то есть есть разрешимая функция  $U(p; x)$ , которая возвращает 1, если  $p(x) = 1$  и 0 иначе. Напишем две штуки:

```

p(x):
    if u(p, x):
        return 0;
    else:
        return 1;

```

Отличная программа. Теперь то же самое, но математически. Пусть  $V(p; x) = \neg U(p; x)$ . По теореме о рекурсии существует  $p$  такое что  $v(p; x) \equiv p(x)$ . Если  $p(x) = 1$ , то  $V(p; x) = 1$ , значит  $U(p; x) = 0$ , значит  $p(x) \neq 1$ .  $\square$

**Теорема 12.**  $HALT_\epsilon$  неразрешим.

*Доказательство.* Пусть разрешим программой  $h$ . Тогда вот то, что приведёт нас к противоречию:

```
p(x):
  if x == epsilon && h(p):
    while true {}
  else:
    return 1
```

 $\square$ 

**Теорема 13.** Любое нетривиальное свойство перечислимых языков неразрешимо.

*Доказательство.* Пусть есть свойство  $A$  нетривиальное и разрешимое. Пусть у нас есть языки  $Q$  и  $R$  с полурешителями  $q$  и  $r$ , где  $Q \in A$  и  $R \in A$ . И пусть у нас есть  $inA$  — разрешитель  $A$ . Напишем такую программу:

```
p(x):
  if inA(p):
    return r(x)
  else:
    return q(x)
```

 $\square$ 

*Замечание.* Перейдём уже к чему-то, что без теоремы о рекурсии плохо доказывается.

**Теорема 14** (Теорема о неподвижной точке). Для любой всюду определённой вычислимой функции  $f$  существует программа  $p$ , что  $f(p)$  и  $p$  ведут себя одинаково на любом вводе.

*Доказательство.* Рассмотрим  $f$ . Напишем следующую программу:

```
p(x):
  q = f(p)
  return q(x)
```

 $\square$ 

*Замечание.* Интересный факт: можно теорему о неподвижной точке рассматривать как базовую, а теорему о рекурсии — как её следствие.

**Теорема 15** (Первая теорема Гёделя о неполноте). В любой достаточно богатой формально непротиворечивой системе существует верное недоказуемое утверждение.

*Замечание.* С нашей точки зрения аксиомы — это строки, и мы можем получать из одних строк другие (при помощи естественного вывода). И система непротиворечива, если нельзя одновременно доказать  $\alpha$  и  $\neg\alpha$ . Причём тут нам пофиг вообще, какие у нас аксиомы нам единственное, что хочется — разрешимость. Т.е. по схеме аксиом и утверждению определить, подходит ли оно и по трём утверждениям выяснить, корректный ли  $\text{modus ropens}$ . В условиях вышеперечисленного мы можем проверить, корректное ли у нас доказательство.

Что такое «достаточно богатая» — вопрос. В математической логике вводят конкретные аксиомы, причём достаточно малое их количество, но мы не жадные и скажем, что система достаточно богата, если в ней сформулировать утверждение «программа на данном входе даёт единицу».

*Доказательство.* Рассмотрим такую программу:

```
p(x):
  s = "p(x) зависит"
  for y --- доказательство:
    if y доказывает s:
      return 1
```

Если  $p$  завершает работу, то она нашла доказательство, что она зависит. А у нас система непротиворечива. Значит она не завершает работу, но доказать мы это не можем.  $\square$

**Теорема 16** (Вторая теорема Гёделя о неполноте). *Никакая достаточно богатая непротиворечивая система не может доказать свою непротиворечивость.*

*Доказательство.* Рассмотрим первую теорему. Рассмотрим программу, если  $p(x)$  не зависит, то проблема, значит она зависит. А вот давайте попытаемся сформулировать эту же фразу в нашей системе. И если бы система могла доказать свою непротиворечивость, то из этого бы следовало, что  $p$  зависит, а значит у этого было бы доказательство. Но его нет.  $\square$

*Замечание.* Давайте закончим с математической логикой и перейдём к алгоритмам сжатия.

**Определение 16.**  $K(s) = \min\{\text{len}(p) \mid p(\varepsilon) = s\}$  называется **колмогоровской сложностью**.

**Теорема 17.** *Колмогоровская сложность невычислима.*

*Доказательство.* От противного. Рассмотрим программу

```
p(x):
  for s in Sigma*:
    if K(s) > len(p):
      print(s)
      return
```

Мы найдём такую строку, что  $K(s)$  больше длины  $p$ . То есть наименьшая длина программы, которая выводит  $s$  больше длины  $p$ . Но  $p$  же выводит  $s$ , противоречие. Причём такая строка  $s$  точно существует, потому что  $\text{len}(p)$  — какое-то число, количество программ размера меньше либо равных его конечно, а значит и количество строк, у которых сложность меньше  $\text{len}(p)$  конечно.  $\square$

**Теорема 18.** Пусть  $f$  всюду определённая вычислимая функция такой что  $\forall s \ K(s) \geq f(s)$ . Тогда  $\exists c \ f(s) \leq c$ .

*Доказательство.* Точно такое же, только вместо  $K$  надо подставить  $f$  в код. Поскольку  $K(s) \geq f(s) > \text{len } p$ , ситуация та же самая. Тогда

$$c = \max\{\text{len } p, f(s) : f(s) > \text{len } p\}$$

Если бы это множество было бесконечно, мы бы получили противоречие.  $\square$

**Определение 17.** **Busy Beaver (усердный бобёр)** — следующая функция.

$$\text{BB}(n) = \max\{\text{len } s \mid \exists p : \text{len } p \leq n, p(\varepsilon) = s\}$$

**Теорема 19.** Для любой всюду определённой вычислимой функции  $f$  существует конечное число таких  $n$ , что  $\text{BB}(n) \leq f(n)$

*Замечание.* Доказательство остаётся читателю как домашнее задание.