

Глава 1

Класс P .

1.1 Определение класса P .

Функция f – *полиномиального роста*, если для некоторых c и n_0 выполнено

$$f(n) < n^c \text{ при } n > n_0.$$

Время работы машины Тьюринга M на словах алфавита Σ (подмножество ленточного алфавита) *полиномиально*, если

$$T_M(n) = \max_{|\mathbf{x}|=n} T_M(\mathbf{x}), \text{ где } |\mathbf{x}| = |x_1| + \dots + |x_k|,$$

есть функция полиномиального роста (в частности, такая машина всегда останавливается на словах алфавита Σ ; можно также использовать $|\mathbf{x}| = \max |x_i|$ – получится эквивалентное определение).

Определение 1.1 *Класс P в широком смысле состоит из всех тотальных функций $f : (\Sigma^*)^k \rightarrow (\Sigma^*)^l$, вычислимых на машинах Тьюринга с полиномиальным временем работы. Класс P в узком (основном) смысле, состоит из всех предикатов $L : \Sigma^* \rightarrow \{0, 1\}$, вычислимых на машинах Тьюринга с полиномиальным временем работы.*

Замечание. Часто предикаты на множестве Σ^* отождествляют с их областями истинности

$$\{x \in \Sigma^* \mid L(x) = 1\}$$

и называют формальными языками. В этом случае элементы класса P (в узком смысле) есть языки, для которых условие “ $x \in L$ ” распознается за полиномиальное время.

Замечание. Алфавит Σ в этом определении не фиксирован – в класс P входят все такие функции и предикаты для всевозможных конечных алфавитов:

$$P = \bigcup_{\Sigma} P_{\Sigma}.$$

В то же время для каждого алфавита Σ двоичное кодирование букв порождает естественное вложение $P_{\Sigma} \mapsto P_{\{0,1\}}$, поэтому $P_{\{0,1\}}$ фактически уже содержит в себе все функции и предикаты из P .

Замечание. Определение класса P в значительной мере не зависит от модели вычисления. Если модель вычислений допускает компиляцию в машины Тьюринга и обратно с полиномиальным замедлением (а таковы все реальные языки программирования), то класс P , определенный посредством этой модели совпадает с нашим. В частности, совершенно несущественен выбор варианта машин Тьюринга (сколько каких лент, головок и т.п.).

Замечание. Имеется устойчивое общественное мнение, что предикаты и функции не из P реально вычислять гораздо труднее, чем предикаты и функции класса P . Практические задачи, которые лежат в классе P , но требуют для своего решения времени, оцениваемого полиномом большой степени, встречаются крайне редко. Обычно приходится сталкиваться с одной из двух реалий: либо задача решается за время – полином небольшой степени с не слишком большими коэффициентами, либо она требует экспоненциального времени, причем резкий рост временных затрат начинается уже на малых длинах входных данных. Эти различия наглядно проявляются в процессе эксперимента, поэтому факт принадлежности задачи классу P позволяет прогнозировать реальное поведение соответствующей программы.

1.2 Примеры: целочисленная арифметика.

Считаем, что числа представлены в двоичной записи. Тогда операции $+$, $-$, $*$, div , mod (работают “школьные” алгоритмы), $\text{nod}(a,b)$ (алгоритм Евклида: $\text{nod}(a,b) = \text{nod}(b, a \bmod b)$) принадлежат классу P .

Заметим, что $\exp(x) = 2^x$ не принадлежит классу P , т.к. длина результата уже не полиномиальна как функция от длины x (т.е. от $\log_2 x$).

Например, для $d = a \text{ div } b$ и $r = a \text{ mod } b$:

$d:=0;$

```

WHILE (a >= b) DO      // (log a) итераций
  x:=1;

  WHILE a-x*b >0 DO    // (log a) итераций.
    z:=x;              // Подбираем z -
    x:=x*2;            // наиб. степень 2 такую,
  DONE;                // что a=z*b+y и y>=0

  d:=d+z; a:=a-z*b;
DONE
r:=a.

```

Представим себе реализацию этого алгоритма на описанном выше варианте BASIC'a. Здесь 7 переменных, а в реализации будет $7 + const$. Здесь 10 операторов, а в реализации будет $10 \cdot const$. Здесь длины двоичных записей текущих значений переменных не более $s = (\log a + \log b) \cdot const$ и там – тоже. Здесь каждый оператор исполнялся не более $\log^2 a$ раз и там – тоже. В итоге для реализации будет $t = C_1 \cdot \log^2 a$, а $s = (\log a + \log b) \cdot C_2$. По (??) получаем, что при моделировании этого машиной Тьюринга время работы окажется ограниченным полиномом от $n = \log a + \log b$. Но такое значение n и есть длина входных данных, откуда $div, mod \in P$.

Алгоритм Евклида $d = \text{нод}(a, b)$:

```

IF b>a THEN x:= a; a:= b; b:= a FI; //делаем a>=b

WHILE b>0 DO
  x:= a mod b; // (a,b):= (b, a mod b)
  a:= b;
  b:= x
DONE;
d:= a.

```

На каждой итерации произведение ab уменьшается по крайней мере в 2 раза (т.к. $a \geq b + (a \text{ mod } b) \geq 2 \cdot (a \text{ mod } b)$), поэтому количество итераций не больше $\log_2 ab = \log_2 a + \log_2 b$, т.е. длины входных данных. Далее повторяем те же рассуждения про моделирование.

1.3 Примеры: арифметика остатков.

Операции $+$, $*$ в кольцах вычетов \mathbb{Z}_m легко вычисляются за полиномиальное время с помощью целочисленной арифметики. Алгоритм Евклида позволяет за полиномиальное время по a и m распознать обратимость элемента a в кольце \mathbb{Z}_m (проверяем условие $\text{nod}(a, m) = 1$). Более тонкие факты: функции $f(x, n) = x^{-1}$ в \mathbb{Z}_m (если не существует, то 0) и $\text{exp}(x, y, m) = x^y \bmod m$ также лежат в классе P .

$(x^y \bmod m)$. Сначала заметим, что при $y = 2^k$ это выражение можно вычислить с помощью k итераций оператора $x := x * x \bmod m$. Общий случай сводится к этому при помощи разложения

$$y = 2^{k_1} + \dots + 2^{k_n}, \quad k_1 < \dots < k_n, \quad n \leq k_n \leq |y|.$$

Достаточно следующего алгоритма:

```

exp:=1; z:=1;
WHILE y>0 DO                               // будет n итераций
  IF (y mod 2 =1) THEN
    u:= (x^z) mod m;                         // z есть степень двойки
    exp:=exp*u
  FI;
  z:=z*2;
  y:=y div 2
DONE.
```

$f(x, n) = x^{-1}$ в \mathbb{Z}_m . Если x – обратимый элемент кольца \mathbb{Z}_m , то $(x, m) = 1$ и достаточно найти разложение единицы $\alpha x + \beta m = 1$ (тогда $x^{-1} = \alpha \bmod m$). Для поиска такого разложения следует модифицировать алгоритм Евклида ($\text{nod}(x, m)$), т.е. реализацию выполняемого в цикле присваивания

```

r := a mod b ;
(a,b) := (b, r) ;
```

Будем хранить текущие компоненты пары (a, b) в виде целочисленных линейных комбинаций исходных данных:

$$a = \alpha x + \beta m, \quad b = \gamma x + \delta m,$$

(в начальный момент $\alpha = \delta = 1, \beta = \gamma = 0$). Тогда

$$r = a - (a \text{ div } b) \cdot b,$$

что позволяет найти коэффициенты разложения для следующей итерации. Алгоритм заканчивает работу на паре $((x, m), 0) = (1, 0)$, поэтому в результате будет построено требуемое разложение единицы. Каждая итерация увеличится на фиксированное число шагов, но количество итераций не изменится. Поэтому сохранится прежняя полиномиальная оценка на время работы.

1.4 Примеры: сложение и умножение матриц.

Для простоты ограничимся квадратными $(n \times n)$ матрицами с элементами из кольца \mathbb{Z}_{2^m} , представленными двоичными записями с лидирующими нулями длины $m = \text{const}$. На вход поступают число n и элементы матриц A, B (записанные подряд через разделитель). Длина входа есть полином от n , поэтому достаточно построить вычисление, ограниченное по времени другим полиномом от n . Рассмотрим, например, умножение.

Тьюрингово вычисление будет состоять в моделировании соответствующей программы обсуждавшегося ранее варианта BASIC'а со ссылками. Эта программа получает исходные данные в качестве начальных значений переменных $V_0, \dots, V(2n^2)$ и должна записать элементы произведения матриц в переменные $V(2n^2 + 1), \dots, V(3n^2)$. (Моделирующая машина Тьюринга должна проделать инициализацию переменных и в конце переписать результат на выходную ленту; очевидную реализацию этого мы опускаем.)

Сама BASIC-программа получается как результат моделирования стандартного метода перемножения матриц ($O(n^3)$ итераций)

```
FOR i=0 TO n-1 DO
  FOR j=0 TO n-1 DO
    FOR k=0 TO n-1 DO
      c[i,j]:=c[i,j]+a[i,k]*b[k,j]
    DONE
  DONE
DONE
```

Единственное, что требует разъяснений – реализация массивов (т.е. переменных в индексах) с помощью ссылок. Вместо выражения $a[i,k]$ в BASIC-программе надо использовать ссылку `ref d` на значение $a[i,k]$. Оно хранится в переменной $V(i \cdot n + k)$, поэтому переменной Vd надо предварительно присвоить значение $(i \cdot n + k)$. Таким образом, оператор `... a[i,k] ...` надо переводить так

```

100: Vd <- i * n
101: Vd <- Vd + k
102: ... (ref d) ...

```

где i , k , n обозначают переменные, хранящие значения i, k, n . Для остальных двух массивов поступаем аналогично, но учитываем, что соответствующие адреса есть $(n^2 + k \cdot n + j)$ и $(2 \cdot n^2 + i \cdot n + j)$. В итоге тело самого внутреннего цикла будет переведено фиксированным (конечным) набором операторов присваивания и временная оценка $t = O(n^3)$ при переводе всего блока сохранится. Суммарная длина содержимого всех переменных будет не более $s = 3 \cdot n^2 \cdot m + \text{const} \cdot \log n = O(n^2)$. Применяем (??) и получаем полиномиальную оценку на время соответствующего тьюрингова вычисления.

1.5 Примеры: связность в графе.

Граф G с множеством вершин $\{1, \dots, n\}$ задан матрицей смежности M . Это симметричная $n \times n$ матрица из 0 и 1; $m_{i,j} = 1$ т. и т.т., когда в графе есть ребро (i, j) . Например,

$$G : \begin{array}{cc} 1 & 2 \\ & \\ 3 & 4 \end{array} \quad M = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}.$$

Надо по M, i, j выяснить, связаны ли вершины i, j в графе G .

Эта задача легко сводится к вычислению n -ой степени 0,1-матрицы $A = (M + E)$, если в качестве операций $+, \cdot$ взять булевы операции \vee, \wedge (далеко не самый эффективный метод, но для доказательства принадлежности классу P работает). Степень матрицы A^q задает отношение связности за $\leq q$ шагов, поэтому для $(a_{i,j}^{(n)}) = A^n$

$$a_{i,j}^{(n)} = 1 \Leftrightarrow i \text{ и } j \text{ связаны в } G.$$

Вычислять A^n можно n -кратным перемножением матриц (см. предыдущий пример; использование булевых операций вместо $+, \cdot$ ничего не меняет).

Глава 2

Класс $P/Poly$.

2.1 Распознавание языков последовательностями булевых схем.

Напомним, что формальным языком мы назвали произвольное множество слов данного алфавита. Ограничимся языками $L \subset \{0, 1\}^*$. С каждым таким языком можно связать последовательность булевых функций $f_n(x_1, \dots, x_n)$, $n = 0, 1, \dots$ такую, что

$$x_1 \dots x_n \in L \Leftrightarrow f_n(x_1, \dots, x_n) = 1.$$

Схемной сложностью языка L называется функция

$$c_L(n) = c(f_n),$$

где через $c(\cdot)$ обозначена схемная сложность булевой функции.

Пример. Оценить сверху схемную сложность языка, состоящего из всех слов-перевертышей. Схема для f_n в этом случае может быть такая:

```
f <- ( x1 <-> xn )  
  
z <- ( x2 <-> x(n-1) )  
f <- ( f & z )  
  
z <- ( x3 <-> x(n-2) )  
f <- ( f & z )  
...
```

Поэтому $c_L(n) = O(n)$.

Следует заметить, что схемная сложность языка является огрубленной сложностной характеристикой – она не учитывает трудоемкость построения самой схемы. Но за счет огрубления становится принципиально возможным оценивать сложность произвольных языков (а не только разрешимых, как для машин Тьюринга).

Класс $P/Poly$ определяется как семейство всех таких языков $L \subset \{0, 1\}^*$, схемная сложность которых есть функция полиномиального роста, т.е.

$$c_L(n) < n^C \text{ при } n > n_0$$

для некоторых C и n_0 .

Класс $P/Poly$ замкнут относительно операций объединения, пересечения и дополнения (проверить) языков.

2.2 Континуальность класса $P/Poly$.

В отличие от класса P , в класс $P/Poly$ входят не только разрешимые языки. Мощность класса $P/Poly$ равна континууму. Каждой (не обязательно вычислимой) функции $\varphi : N \rightarrow \{0, 1\}$ можно инъективно сопоставить язык L схемной сложности $O(1)$:

$$x_1 \dots x_n \in L \Leftrightarrow \varphi(n) = 1.$$

Класс P представляет собой эффективизованную версию класса $P/Poly$: если к определению языка $L \in P/Poly$ добавить условие вычислимости за полиномиальное время отображения

$$n \xrightarrow{s} \begin{cases} \text{булева схема полиномиального размера,} \\ \text{вычисляющая } f_n, \end{cases}$$

то класс таких языков в точности совпадает с $P_{\{0,1\}}$.

Для доказательства принадлежности языков L , лежащих в модифицированном указанным образом (“равномерном”) классе $P/Poly$, к классу $P_{\{0,1\}}$, достаточно реализовать следующий алгоритм проверки условия “ $x \in L$ ”:

1. По входному слову $x = x_1 \dots x_n$ найти его длину n .
2. По n найти булеву схему S_n , правильно проверяющую условие “ $x \in L$ ” на словах длины n .

3. Вычислить $S_n(x_1, \dots, x_n)$ и выдать полученный результат в качестве ответа.

Для реализации можно взять композицию машины Тьюринга, вычисляющей отображение s и универсальной машины Тьюринга для языка булевых схем. Получится машина Тьюринга, которая распознает язык L за полиномиальное время.

Обратное включение обеспечивает конструкция из доказательства теоремы 2.1 (см. ниже), которая на самом деле строит отображение s вычислимым за полиномиальное время.

2.3 Включение $P \subset P/Poly$.

Ранее обсуждалось, что с точностью до кодировки исходных данных класс P (в узком смысле) совпадает с $P_{\{0,1\}}$ (ограничение алфавитом $\{0, 1\}$).

Теорема 2.1 $P_{\{0,1\}} \subset P/Poly$.

Доказательство. Пусть $L \in P$. Тогда существует одноленточная машина Тьюринга M , разрешающая L , и полином $p(n)$, ограничивающий сверху ее время работы на словах достаточно большой длины n . Этот же полином будет ограничивать и зону M . Занумеруем клетки ленты целыми числами, 0 – где стоит головка в начальный момент, входное слово – в клетках $1, \dots, n$, число n – достаточно большое.

Протоколом работы машины Тьюринга M за время T на слове $x_1 \dots x_n$ называется прямоугольная таблица следующего вида:

	$-S$		0	1		i			S
0	$\#, \emptyset$...	$\#, q_1$	x_1, \emptyset	x_2, \emptyset	...	x_n, \emptyset	...	$\#, \emptyset$
1	$\#, \emptyset$								$\#, \emptyset$
	\vdots								\vdots
j	$\#, \emptyset$					$\Gamma_{i,j}$			$\#, \emptyset$
	\vdots								\vdots
T	$\#, \emptyset$								$\#, \emptyset$

Номера столбцов соответствуют нумерации клеток ленты, номера строк – шагам вычисления. В клетке (i, j) таблицы записана пара $\Gamma_{i,j} = \langle a, q \rangle$, где a – содержимое i -той клетки ленты в момент $t = j$, а q есть соответствующее состояние машины, если в этот момент времени головка

обозревает клетку i , и \emptyset – в противном случае. Ширина таблицы $2 \cdot S + 1$ выбирается таким образом, чтобы головка машины в процессе работы все время находилась внутри куска ленты $|i| < S$. Если машина остановилась раньше момента T , то последние строки таблицы дублируют предыдущие. Для этого доказательства возьмем $T = S = p(n)$.

В клетке протокола может стоять произвольная пара $\langle a, q \rangle \in D$, где $D = \Sigma \times (Q \cup \{\emptyset\})$. Заметим также, что содержимое клетки $(i, j + 1)$ для $|i| < S$ функционально зависит от содержимого трех клеток над ней,

$$\begin{array}{|c|c|c|} \hline \Gamma_{i-1,j} & \Gamma_{i,j} & \Gamma_{i+1,j} \\ \hline & \Gamma_{i,j+1} & \\ \hline \end{array} \quad \Gamma_{i,j+1} = \varphi(\Gamma_{i-1,j}, \Gamma_{i,j}, \Gamma_{i+1,j}),$$

а содержимое клеток на верхней, левой и правой границах есть либо константа $\langle \#, \emptyset \rangle$ или $\langle \#, q_1 \rangle$, либо одна из следующих функций от n булевых переменных, представляющих входное слово:

$$\langle x_i, \emptyset \rangle = \pi_i(x_1, \dots, x_n).$$

Входное слово принадлежит языку L т. и т.т., когда в нижней строке протокола встречается пара $\langle 1, q_0 \rangle$.

Добавим к языку программирования булевых схем переменные по конечному множеству D (новый тип данных), константы $\langle \#, \emptyset \rangle$, $\langle \#, q_1 \rangle \in D$ и дополнительные "встроенные" операции $\varphi : D^3 \rightarrow D$, $\pi_i : \{0, 1\}^n \rightarrow D$, и $\chi : D \rightarrow \{0, 1\}$, где

$$\chi(\langle a, q \rangle) = 1 \Leftrightarrow \langle a, q \rangle = \langle 1, q_0 \rangle.$$

В расширенном языке нетрудно написать программу полиномиального размера, отвечающую на вопрос " $x_1 \dots x_n \in L$?". Она состоит из $(2 \cdot S + 1) \cdot T$ операторов присваивания, вычисляющих значения всех переменных $\Gamma_{i,j}$ по слоям сверху вниз и еще $2 \cdot (2 \cdot S + 1)$ операторов для вычисления ответа

$$f = \bigvee_{i=-S}^S \chi(\Gamma_{i,T}).$$

Остается промоделировать эту программу программой в исходном языке булевых схем (без типа D) с сохранением полиномиальной оценки на длину программы. Учитывая конечность множества D , можно выбрать число m из условия $2^{m-1} < |D| \leq 2^m$ и имитировать каждую переменную Γ типа D набором из m булевых переменных. Используя тип D дополнительные встроенные функции (и константы) можно заменить на конечные наборы булевых функций, вычисляющие биты

результата по битам аргументов. Каждую из последних (их конечное множество) реализуем булевой схемой (через \neg, \vee, \wedge например). Размеры всех этих схем ограничены некоторой константой d (т.к. их конечное число). Теперь каждый оператор программы в расширенном языке можно заменить на не более $m \cdot d$ операторов в исходном, производящих то же преобразование значений переменных, но в их двоичном представлении. Так как входные и выходные переменные нашей программы уже были булевыми, то это преобразование не изменит вычисляемой программой функции. При этом длина программы возрастет не более, чем в $m \cdot d$ раз, т.е. останется функцией полиномиального роста.

■