# 4. Семинар 28.02.2017

## 4.1. *Недетерминированные МТ

До этого мы рассматривали детерминированные машины Тьюринга, т.е. настоящие алгоритмы — строго определенные последовательности действий. Однако, в теории алгоритмов оказывается полезным такое неинтуитивное понятие, как **недерминированная машина Тьюринга** (недерминированный алгоритм). НМТ задается как и обычная МТ, только функция перехода *многозначная*: каждой паре (состояние, символ) может соответствовать несколько допустимых действий

$$\delta(q, a) \subset Q \times \Pi \times \{-1, 0, 1\}.$$

Последовательность шагов НМТ называется *допустимой*, если на каждом шаге $(q', a', i) \in \delta(q, a)$.

Недерминированная МТ $M$ *принимает язык* $L$, если для любого слова $x \in L$ существует такая допустимая последовательность шагов, что в конце концов $M$ останавливается в состоянии $Accept$, а для слов не из $L$ такой последовательности, соответственно, не существует. Заметим, что тут ситуация схожа с перечислимыми языками: НМТ может только *принять* слово, но не может его *отвергнуть*.

Очевидно, детерминированные МТ являются частным случаем недетерминированных. Тем не менее, с точки зрения вычислимости они эквивалентны: недерминированные МТ принимают все перечислимые языки, и только их. Действительно, для любого перечислимого языка существует принимающая его детерминированная МТ, а следовательно, и недетерминированная. С другой стороны, работу НМТ можно моделировать на ДМТ: достаточно для каждого шага хранить все конфигурации, в которые НМТ могла перейти к этому моменту, а их конечное число. но существенная разница между ДМТ и НМТ возникает при анализе сложности алгоритмов.

## 4.2. *Полиномиальная иерархия

Язык $L$ принадлежит классу $DTIME\big(f(n)\big)$, если существует такая детерминированная МТ $M$, которая разрешает $L$, что $T_M(n) = O\big(f(n)\big)$. Почему используется асимптотическая оценка, а не обычное неравенство? Дело в том, что имеет место **теорема об ускорении**: для любой многоленточной МТ $M$ и константы $c > 0$ можно построить эквивалентную многоленточную МТ $M'$ такую, что $T_{M'}(n) \leq cT_M(n) + O(n)$. Аналогично, $DSPACE\big(f(n)\big)$ — это класс языков, которые можно разрешить на ДМТ, используя $O\big(f(n)\big)$ памяти. Несколько труднее определить сложность в случае недетерминированного алгоритма. Будем говорить, что НМТ $M$, распознающая некоторый язык $L$, имеет временную сложность не больше $f(n)$, если для любого $x \in L$ НМТ принимает его за время, не большее $f(|x|)$, т.е. существует допустимая последовательность вычислений длины не больше $f(|x|)$, приводящая к состоянию $Accept$. Класс языков, принимаемых недетерминированными МТ за время $O\big(f(n)\big)$, обозначается $NTIME\big(f(n)\big)$. Аналогично определятся класс $NSPACE\big(f(n)\big)$. Заметим, что языки из всех этих классов являются разрешимыми, если $f(n)$ вычислима: достаточно сэмулировать все допустимые последовательности вычислений длины не больше $Cf(|x|)$, и если ни в одной из них машина не приняла $x$, то $x \notin L$.

Для любой вычислимой функции $f(n)$ имеют место следующие соотношения между раз-

личными классами:

$$DTIME(f(n)) \subset NTIME(f(n)) \subset DSPACE(f(n)) \subset NSPACE(f(n)),$$
$$NSPACE(f(n)) \subset \bigcup_{c>0} DTIME(2^{cf(n)}).$$

Если к тому же $f(n) \geq \log n$, то верна теорема Савича:

$$NSPACE(f(n)) \subset DSPACE(f^2(n)).$$

Как правило, считается, что если алгоритм работает за полиномиальное время, то он достаточно эффективный. Например, алгоритм, решающий какую-нибудь задачу полным перебором, будет работать экспоненциально долго, и будет неэффективным. Поэтому вводят следующие *классы языков*: $P$ — класс языков, разрешимых на детерминированной МТ за полиномиальное время, $NP$ — класс языков, принимаемых недетерминированной МТ за полиномиальное время, *co*-$NP$ — класс языков, дополнение которых лежит в $NP$. Через ранее введенные классы это выражается так:

$$P = \bigcup_{d\geq 0} DTIME(n^d),$$
$$NP = \bigcup_{d\geq 0} NTIME(n^d),$$
$$co\text{-}NP = \{L : \overline{L} \in NP\}.$$

Можно продолжать эту классификацию и дальше, т.к. оказывается, что $NP$ состоит из таких языков $L$, для любого слова $x \in L$ можно предъявить такой *сертификат* $y$ полиномиальной длины от $x$, т.е. $|y| = O\big(poly(|x|\big)$, что некоторая детерминированная полиномиальная МТ примет пару $(x, y)$. Эта конструкция аналогична классу $\Sigma_1$ (перечислимым множествам), и, добавляя больше кванторов, можно получать новые классы языков, которые образуют *полиномиальную иерархию*, по аналогии с арифметической иерархией. Таким же образом определяются классы $PSPACE$ и $NPSPACE$. Верны следующие соотношения:

$$P \subset NP \cap co\text{-}NP,$$
$$P \subset PSPACE,\ NP \subset NPSPACE,$$
$$PSPACE = NPSPACE.$$

Однако, вопрос о том, верно ли $P = NP$, остается открытым — это одна из “проблем тысячелетия”. Есть и другие вопросы, например, равны ли классы $NP$ и *co*-$NP$ (если $P = NP$ то это, конечно, так).

## 4.3. RAM модель

Теперь мы перейдем к другой модели вычислений, более близкой к реальным компьютерам. Это **random access machine** (RAM). RAM состоит из бесконечного набора *регистров* $\ldots x[-1], x[0], x[1], \ldots$ и списка команд (программы). В регистрах хранятся целые числа. Ко-

манды могут иметь различный вид, например:

$$
\begin{aligned}
&x[i] := 0,\\
&x[i] := x[i] + 1,\ x[i] := x[i] - 1,\\
&x[i] := x[i] + x[j],\ x[i] := x[i] - x[j],\\
&x[i] = x[x[j]],\ x[x[i]] := x[j],\\
&\text{if } x[i] \le 0 \text{ then GOTO } p,
\end{aligned}
$$

также можно добавить команды умножения, деления (с остатком), и т.д. Важная особенность — команды, которые дают возможность обращаться к произвольной ячейке памяти по ее *адресу* за константное время. Из-за этого свойства модель и называется *random access* machine. Машина выполняет команды по очереди, пока не встретит команду перехода GOTO. RAM завершает работу, когда попадет на пустую команду (если перед этим она была на последней команде в списке или в команде GOTO $p$ некорректный номер).

RAM и машина Тьюринга — эквивалентные модели, их можно моделировать друг на друге. Например, чтобы смоделировать МТ на RAM, достаточно хранить в ячейке $x[i]$ число $0$ или $1$, соответствующее символу в ячейке ленты МТ, и в выделенном регистре хранить положение головки. Переходы МТ, очевидно, легко смоделировать на RAM. В то же время, МТ может хранить целые числа и выполнять арифметические операции, поэтом на ней можно моделировать RAM.

Тем не менее, очевидно, RAM может выигрывать у МТ в быстродействии, за счет произвольного доступа к памяти. Чтобы определить время работы RAM, нужно присвоить каждой команде определенное время выполнения, вес. Тогда время работы $t(x)$ есть сумма времени выполнения всех команд, исполнявшихся RAM. Существуют различные способы это сделать. Например, можно присвоить всем командам одинаковое время выполнения — тогда мы получим *равномерный весовой критерий*. Его разумно применять, если мы считаем, что в ходе работы программы не будут возникать слишком большие числа. Например, мы анализируем алгоритм, который будет реализован на $C$, и знаем, что в реальных задачах хватит типа данных int. Однако надо учесть, что равномерный весовой критерий, вообще говоря, позволяет производить некоторые операции гораздо быстрее, чем это возможно в реальности: например, вектора или матрицы можно записать как очень большие целые числа, тогда их сложение в нашей модели будет занимать $O(1)$ времени. Другой распространенный подход — *логарифмический весовой критерий*. В списке команд нет умножения и деления, а сложение целых чисел с $n$ битами занимает $O(\log n)$ времени. Этот критерий соответствует случаю, когда программа работает с большими числами, выходящими за пределы разрядной сетки *(длинная арифметика)*.

Имеют место следующие соотношения между временем работы RAM и МТ.

***Теорема* 1.** *Пусть (многоленточная) машина Тьюринга $M$ делает $t_M(x)$ шагов на входе $x$. Тогда можно построить RAM $R$, которая будет работать за время $t_R(x) = O\big(t_M(x)\big)$.*

*Если дана RAM $R$ с логарифмическим весовым критерием, которая работает за время $t_R(x)$, то можно построить многоленточную МТ, работающую за $O\big(t_R^2(x)\big)$.*

Отсюда следует, что сложности RAM и МТ полиномиально связаны, поэтому применение модели RAM не изменит классы $P$ и $NP$. В дальнейшем мы будем неявно предполагать при анализе сложности, что рассматриваемые нами алгоритмы реализованы на RAM: либо с

равномерным весовым критерием, если мы не учитываем размеры чисел, либо с логарифмическим.

## 4.4. *Рекурсивные функции

**Рекурсия** — вызов функции (подпрограммы) из нее же самой. В математическом смысле, рекурсивная функция — это функция, которая определяется через ее же значения в других точках.

Важный класс рекурсивных функций — это **примитивно-рекурсивные** функции. Они получаются из базовых функций сложения ($x \mapsto x+1$) и проектирования ($(x_1, \ldots, x_k) \mapsto x_i$) с помощью операций подстановки и примитивной рекурсии. *Подстановка* заключается в том, что имея функции $f(y_1, \ldots, y_k)$ и $g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)$, можно сформировать функцию

$$(x_1, \ldots, x_n) \mapsto f\big(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)\big).$$

*Примитивная рекурсия* же из функций $f(x_1, \ldots, x_k)$ и $g(x_1, \ldots, x_k, y, z)$ формирует следующую функцию $h(\cdot)$:

$$\begin{aligned} h(x_1, \ldots, x_k, 0) &:= f(x_1, \ldots, x_k), \\ h(x_1, \ldots, x_k, y+1) &:= g\big(x_1, \ldots, x_k, y, h(x_1, \ldots, x_k, y)\big). \end{aligned}$$

Можно также сказать, что они реализуются программами с циклами **for** и операторами **if …then …else**, но без циклов **while**.

Более общее понятие — **частично-рекурсивная** функция. Она получается из базовых операций с помощью подстановки, примитивной рекурсии и *оператора минимизации* $\mu$:

$$g(x_1, \ldots, x_k) = \mu y\big(f(x_1, \ldots, x_k, y) = 0\big) := \min\{y : f(x_1, \ldots, x_k, y) = 0\}.$$

Оказывается, множество частично-рекурсивных функций в точности *совпадает с множеством всех вычислимых функций*.

## 4.5. Алгоритмы с рекурсией

Простейший пример рекурсии, которая возникает в алгоритмах — хвостовая рекурсия, когда рекурсивный вызов функции происходит в конце соответствующей подпрограммы. Например, рекурсивное вычисление факториала: $f(0) = 1$, $f(n) = nf(n-1)$, $n \geq 1$. Очевидно, хвостовую рекурсию можно заменить на обычный цикл. Более интересен случай, когда рекурсию нельзя так просто свести к циклам, например, если функция рекурсивно вызывается несколько раз в ходе подпрограммы. Важный пример подхода, когда возникает такого рода рекурсия — это парадигма **“разделяй и властвуй”**, которая заключается в разбиении задачи на несколько подзадач того же типа, но меньшего размера, их рекурсивного решения и комбинации полученных ответов.

Рассмотрим задачу сортировки: пусть даны $n$ чисел $a[0], \ldots, a[n-1]$, которые нужно упорядочить по возрастанию. Наивные алгоритмы, такие как сортировка вставками или пузырьком, решают эту задачу за время $O(n^2)$. Тем не менее, применяя подход “разделяй и властвуй”, можно получить более быстрые алгоритмы. Так, алгоритм сортировки слиянием (merge sort) заключается в следующем: разобьем текущий массив на две половины, рекурсивно отсортируем их, а потом “сольем”, т.е. объединим в один отсортированный массив.

**Algorithm 1** Сортировка слиянием

```
function merge_sort(a)
    if len(a) == 1 then
        return a;
    else
        n := len(a);
        x := merge_sort(a[0 ... len/2 − 1]);
        y := merge_sort(a[len/2 ... len − 1]);
        i := 0;
        j := 0;
        while i < len/2 or j < len/2 do          ▷ Считаем x[len/2] = y[len/2] = +∞
            if x[i] ≤ y[j] then
                z[i + j] := x[i];
                i := i + 1;
            else
                z[i + j] := y[j];
                j := j + 1;
        return z;
```

Проанализируем сложность этого алгоритма. Считаем, что все базовые операции (присваивание, сравнение и т.д.) занимают $O(1)$ времени. Пусть $T(n)$ — время работы при входе из $n$ элементов (как несложно видеть, оно не зависит от того, как были расположены элементы в исходно массиве). Объединение двух массивов занимает $cn$ времени, поэтому в итоге получается следующая рекуррента:

$$T(n) = 2T\big(\frac{n}{2}\big) + cn,\ n \geq 2,$$
$$T(1) = C.$$

Конечно, если $n$ нечетное, то нельзя разделить массив точно пополам, и правильнее было бы писать

$$T(n) = T\big(\lfloor\frac{n}{2}\rfloor\big) + T\big(\lceil\frac{n}{2}\rceil\big) + cn,$$

но будем для удобства считать, что $n = 2^k$ — как мы увидим позже, это не помешает правильно оценить сложность. Нас интересует асимптотическая оценка $T(n)$. Можно циклически подставлять рекуррентное соотношение, пока не дойдем до нижнего уровня ($n = 1$):

$$T(n) = 2T\big(\frac{n}{2}\big) + cn = 2\Big(2T\big(\frac{n}{4}\big) + c\frac{n}{2}\Big) + cn = 4T\big(\frac{n}{4}\big) + 2cn = \dots$$
$$\dots = 2^k T\big(\frac{n}{2^k}\big) + kcn = 2^k T(1) + cnk = Cn + cn\log n = \Theta(n\log n).$$

Покажем, что это верно для любых $n$, не только степеней 2: очевидно, $T(n)$ неубывающая функция, так что

$$T(n) = O\big(2^{\lceil\log n\rceil}\lceil\log n\rceil\big) = O\big(2^{\log n+1}(\log n + 1)\big) = O(n\log n),$$
$$T(n) = \Omega\big(2^{\lfloor\log n\rfloor}\lfloor\log n\rfloor\big) = \Omega\big(2^{\log n-1}(\log n - 1)\big) = \Omega(n\log n).$$

Таким образом, $T(n) = \Theta(n\log n)$ для всех $n$.

Другой способ анализа рекуррент такого рода — **дерево рекурсии**. Рассмотрим дерево, каждая вершина которого соответствует вызову экземпляра функции. Внутренние вершины помечены числом операций, которые потребовались для обработки результатов следующих рекурсивно вызванных функций (в нашем случае — для слияния двух массивов), а листы — числом операций на минимальной подзадаче, для которой уже не использовалась рекурсия. Кроме того, каждой вершине соответствует уровень, или глубина, $n$ — размер задачи, для которой была вызвана эта функция. Очевидно, время решения задачи (в худшем случае), зависит только от ее размера, поэтому все вершины на одном уровне эквивалентны. Суммарное время работы программы складывается из времени работы в каждой вершине.

Теперь перейдем к другой задаче. Пусть, опять же, дан набор чисел $a[0], \ldots, a[n-1]$. Рассмотрим задачу поиска *$k$-й порядковой статистики*, т.е. такого элемента $a_{(k)}$, который в отсортированном массиве стоит на $k$-м месте. Например, $a(n/2)$, т.е. средний элемент, — это *медиана* массива. Наивный подход заключается в том, чтобы отсортировать массив, и взять $k$-й элемент. Этот алгоритм имеет сложность $O(n \log n)$. Но, оказывается, можно решить эту задачу быстрее. Во-первых, допустим, что мы выбрали некоторый элемент $x$ в качестве опорного, и разбили массив на две части $b$ и $c$ так, что $b[i] < x$, $c[j] > x$. Если $len(b) < k$, то $x \leq a_{(k)}$, и $k$-я порядковая статистика не может находится в массиве $b$. Аналогично, если $len(c) \leq n - k$, то она не может находится в массиве $c$. В принципе, возможны и оба случая, когда искомой величиной оказывается $x$. Если мы можем выбрать $x$ так, чтобы размер массива $b$ и $c$ был не слишком большим, скажем, меньше чем $\alpha n$, $\alpha < 1$, то алгоритм будет работать быстро. Один из способов это сделать следующий: разобьем исходный массив на пятерки, и в каждой из них найдем медиану — это займет $O(n)$ времени; в полученном массиве из медиан $m$ размера $\frac{n}{5}$ рекурсивно найдем медиану $x$, и используем ее в качестве опорного элемента — разбиение исходного массива $a$ на $b$ и $c$ также занимает $O(n)$ времени. Как несложно видеть, в этом случае $len(b), len(c) \leq \frac{7}{10}n$ (с точностью до константы). Затем выбираем массив $b$ или $c$, в котором находится искомая порядковая статистика (например, если $len(b) < k$, то она точно не может находиться в массиве $b$), и рекурсивно запускаем поиск (если это массив $c$, то нужно соответственно изменить номер $k$).

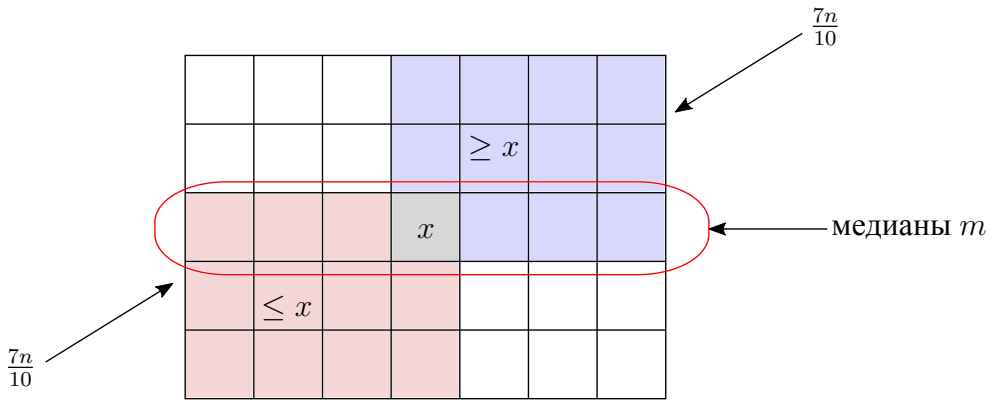


Рис. 1: Разбиение массива

Таким образом, получаем следующее рекуррентное соотношение:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + C_1 n, \; n \geq n_0,$$
$$T(n) \leq C_2, \; n < n_0.$$

Оказывается, решение этой рекурренты $T(n) = O(n)$, т.е. алгоритм работает в худшем случае за линейное время.