Замечание. Две последние задачи похожи. Это сходство станет яснее, если написать матрицы-множители на сторонах 1–2, 2–3,...,$(s-1)$–s многоугольника, а на каждой хорде i–j написать произведение всех матриц, стягиваемых этой хордой.

8.1.6. Железная дорога с односторонним движением имеет n станций. Известны цены билетов от i-ой станции до j-ой (при $i < j$ — в обратную сторону проезда нет). Найти минимальную стоимость проезда от начала до конца (с учетом возможной экономии за счет пересадок).

Мы видели, что замена рекурсивной программы на заполнение таблицы значений иногда позволяет уменьшить число действий. Примерно того же эффекта можно добиться иначе: оставить программу рекурсивной, но в ходе вычислений запоминать уже вычисленные значения, а перед очередным вычислением проверять, нет ли уже готового значения.

8.1.7. Задано конечное множество с бинарной операцией (вообще говоря, не коммутативной и даже не ассоциативной). Имеется n элементов $a_1, \ldots, a_n$ этого множества и еще один элемент x. Проверить, можно ли так расставить скобки в произведении $a_1 \times \ldots \times a_n$, чтобы в результате получился x. Число операций должно не превосходить Cn^3 для некоторой константы C (зависящей от числа элементов в выбранном конечном множестве).

Решение. Заполняем таблицу, в которой для каждого участка $a_i \cdots a_j$ нашего произведения хранится список всех возможных его значений (при разной расстановке скобок).

По существу этот же приём применяется в полиномиальном алгоритме проверки принадлежности слова произвольному контекстно-свободному языку (см. главу 13).

Следующая задача (задача о рюкзаке) уже упоминалась в главе 3.

8.1.8. Имеется n положительных целых чисел $x_1 \ldots x_n$ и число N. Выяснить, можно ли получить N, складывая некоторые из чисел $x_1 \ldots x_n$. Число действий должно быть порядка Nn.

[Указание. После i шагов хранится множество тех чисел на отрезке $0 \ldots N$, которые представимы в виде суммы некоторых из $x_1 \ldots x_i$.]

8.2. Стек отложенных заданий

Другой прием устранения рекурсии продемонстрируем на примере задачи о ханойских башнях.

8.2.1. Написать нерекурсивную программу для нахождения последовательности перемещений колец в задаче о ханойских башнях.

Решение. Вспомним рекурсивную программу, перекладывающую i верхних колец с m на n:

```
procedure move(i,m,n: integer);
| var s: integer;
begin
| if i = 1 then begin
| | writeln ('сделать ход ', m, '->', n);
| end else begin
| | s:=6-m-n; {s - третий стержень: сумма номеров равна 6}
| | move (i-1, m, s);
| | writeln ('сделать ход ', m, '->', n);
| | move (i-1, s, n);
| end;
end;
```

Видно, что задача «переложить i верхних дисков с m-го стержня на n-ый» сводится к трём задачам того же типа: двум задачам с i-1 дисками и к одной задаче с единственным диском. Занимаясь этими задачами, важно не позабыть, что́ ещё осталось сделать.

Для этой цели заведём *стек отложенных заданий*, элементами которого будут тройки $\langle \mathtt{i}, \mathtt{m}, \mathtt{n}\rangle$. Каждая такая тройка интерпретируется как заказ «переложить i верхних дисков с m-го стержня на n-ый». Заказы упорядочены в соответствии с требуемым порядком их выполнения: самый срочный — вершина стека. Получаем такую программу:

```
procedure move(i,m,n: integer);
begin
| сделать стек заказов пустым
| положить в стек тройку <i,m,n>
```

```
  {инвариант: осталось выполнить заказы в стеке}
  while стек непуст do begin
  | удалить верхний элемент, переложив его в <j,p,q>
  | if j = 1 then begin
  | | writeln ('сделать ход', p, '->', q);
  | end else begin
  | | s:=6-p-q;
  | |      {s - третий стержень: сумма номеров равна 6}
  | | положить в стек тройки <j-1,s,q>, <1,p,q>, <j-1,p,s>
  | end;
  end;
end;
```

(Заметим, что первой в стек кладётся тройка, которую надо выполнять последней.) Стек троек может быть реализован как три отдельных стека. (Кроме того, в паскале есть специальный тип, называемый «запись» (`record`), который может быть применён.)

8.2.2. (Сообщил А.К.Звонкин со ссылкой на Анджея Лисовского.) Для задачи о ханойских башнях есть и другие нерекурсивные алгоритмы. Вот один из них: простаивающим стержнем (не тем, с которого переносят, и не тем, на который переносят) должны быть все стержни по очереди. Другое правило: поочерёдно перемещать наименьшее кольцо и не наименьшее кольцо, причем наименьшее — по кругу.

8.2.3. Использовать замену рекурсии стеком отложенных заданий в рекурсивной программе печати десятичной записи целого числа.

Решение. Цифры добываются с конца и закладываются в стек, а затем печатаются в обратном порядке.

8.2.4. Написать нерекурсивную программу, печатающую все вершины двоичного дерева.

Решение. В этом случае стек отложенных заданий будет содержать заказы двух сортов: «напечатать данную вершину» и «напечатать все вершины поддерева с данным корнем» (при этом `nil` считается корнем пустого дерева). Таким образом, элемент стека есть пара: ⟨тип заказа, номер вершины⟩.

Вынимая элемент из стека, мы либо сразу исполняем его (если это заказ первого типа), либо помещаем в стек три порождённых им заказа — в одном из шести возможных порядков.

8.2.5. Что изменится, если требуется не печатать вершины двоичного дерева, а подсчитать их количество?

Решение. Печатание вершины следует заменить прибавлением единицы к счётчику. Другими словами, инвариант таков: (общее число вершин) = (счётчик) + (сумма чисел вершин в поддеревьях, корни которых лежат в стеке).

8.2.6. Для некоторых из шести возможных порядков возможны упрощения, делающие ненужным хранение в стеке элементов двух видов. Указать некоторые из них.

Решение. Если требуемый порядок таков:

корень, левое поддерево, правое поддерево,

то заказ на печатание корня можно не закладывать в стек, а выполнять сразу.

Несколько более сложная конструкция применима для порядка

левое поддерево, корень, правое поддерево.

В этом случае все заказы в стеке, кроме самого первого (напечатать поддерево) делятся на пары:

напечатать вершину `x`, напечатать «правое поддерево» `x`

(= поддерево с корнем в правом сыне `x`). Объединив эти пары в заказы специального вида и введя переменную для отдельного хранения первого заказа, мы обойдемся стеком однотипных заказов.

То же самое, разумеется, верно, если поменять местами левое и правое — получается ещё два порядка.

Замечание. Другую программу печати всех вершин дерева можно построить на основе программы обхода дерева, разобранной в главе 3. Там используется команда «вниз». Поскольку теперешнее представление дерева с помощью массивов `l` и `r` не позволяет найти предка заданной вершины, придётся хранить список всех вершин на пути от корня к текущей вершине. Смотри также главу 9.

8.2.7. Написать нерекурсивный вариант программы быстрой сортировки. Как обойтись стеком, глубина которого ограничена $C \log n$, где n — число сортируемых элементов?

Решение. В стек кладутся пары $\langle i, j\rangle$, интерпретируемые как отложенные задания на сортировку соответствующих участков массива. Все эти заказы не пересекаются, поэтому размер стека не может превысить n. Чтобы ограничиться стеком логарифмической глубины, будем придерживаться такого правила: глубже в стек помещать больший из возникающих двух заказов. Пусть $f(n)$ – максимальная глубина стека, которая может встретиться при сортировке массива из не более чем n элементов таким способом. Оценим $f(n)$ сверху таким способом: после разбиения массива на два участка мы сначала сортируем более короткий (храня в стеке более длинный про запас), при этом глубина стека не больше $f(n/2)+1$, затем сортируем более длинный, так что

$$f(n) \leqslant \max(f(n/2)+1, f(n-1)),$$

откуда очевидной индукцией получаем $f(n) = O(\log n)$.

8.3. Более сложные случаи рекурсии

Пусть функция f с натуральными аргументами и значениями определена рекурсивно условиями

$$\begin{aligned} f(0) &= a, \\ f(x) &= h(x, f(l(x))) \quad (x > 0) \end{aligned}$$

где a — некоторое число, а h и l — известные функции. Другими словами, значение функции f в точке x выражается через значение f в точке $l(x)$. При этом предполагается, что для любого x в последовательности

$$x, l(x), l(l(x)), \ldots$$

рано или поздно встретится 0.

Если дополнительно известно, что $l(x) < x$ для всех x, то вычисление f не представляет труда: вычисляем последовательно $f(0), f(1), f(2), \ldots$

8.3.1. Написать нерекурсивную программу вычисления f для общего случая.

Решение. Для вычисления $f(x)$ вычисляем последовательность

$$l(x), l(l(x)), l(l(l(x))), \ldots$$

до появления нуля и запоминаем её, а затем вычисляем значения f в точках этой последовательности, идя справа налево.

Ещё более сложный случай из следующей задачи вряд ли встретится на практике (а если и встретится, то проще рекурсию не устранять, а оставить). Но тем не менее: пусть функция f с натуральными аргументами и значениями определяется соотношениями

$$\begin{aligned} f(0) &= a, \\ f(x) &= h(x, f(l(x)), f(r(x))) \quad (x > 0), \end{aligned}$$

где a — некоторое число, а l, r и h — известные функции. Предполагается, что если взять произвольное число и начать применять к нему функции l и r в произвольном порядке, то рано или поздно получится 0.

8.3.2. Написать нерекурсивную программу вычисления f.

Решение. Можно было бы сначала построить дерево, у которого в корне находится x, а в сыновьях вершины i стоят $l(i)$ и $r(i)$ — если только i не равно нулю. Затем вычислять значения функции, идя от листьев к корню. Однако есть и другой способ.

Обратной польской записью (или *постфиксной записью*) выражения называют запись, где знак функции стоит после всех ее аргументов, а скобки не используются. Вот несколько примеров:

$f(2)$	$2\ f$
$f(g(2))$	$2\ g\ f$
$s(2, t(7))$	$2\ 7\ t\ s$
$s(2, u(2, s(5, 3)))$	$2\ 2\ 5\ 3\ s\ u\ s$

Постфиксная запись выражения позволяет удобно вычислять его с помощью *стекового калькулятора*. Этот калькулятор имеет стек, который мы будем представлять себе расположенным горизонтально (числа вынимаются и кладутся справа), и клавиши — числовые и функциональные. При нажатии на клавишу с числом это число кладётся в стек. При нажатии на функциональную клавишу соответствующая функция применяется к нескольким аргументам у вершины стека. Например, если в стеке были числа

$$2\ 3\ 4\ 5\ 6$$

и нажата функциональная клавиша s, соответствующая функции от двух аргументов, то в стеке окажутся числа

$$2\ 3\ 4\ s(5,6)$$

Перейдём теперь к нашей задаче. В процессе вычисления значения функции f мы будем работать со стеком чисел, а также с последовательностью чисел и символов `f`, `l`, `r`, `h`, которую мы будем интерпретировать как последовательность нажатий клавиш на стековом калькуляторе. Инвариант такой:

> если стек чисел представляет собой текущее состояние стекового калькулятора, то после нажатия всех клавиш последовательности в стеке останется единственное число, и оно будет искомым ответом.

Пусть нам требуется вычислить значение $f(x)$. Тогда вначале мы помещаем в стек число x, а последовательность содержит единственный символ `f`. (При этом инвариант соблюдается.) Далее с последовательностью и стеком выполняются такие преобразования:

старый стек	старая последовательность	новый стек	новая последовательность
X	x P	$X\ x$	P
$X\ x$	`l` P	$X\ l(x)$	P
$X\ x$	`r` P	$X\ r(x)$	P
$X\ x\ y\ z$	`h` P	$X\ h(x,y,z)$	P
$X\ 0$	`f` P	$X\ a$	P
$X\ x$	`f` P	X	x x `l` `f` x `r` `f` `h` P

Здесь x, y, z — числа, X — последовательность чисел, P — последовательность чисел и символов `f`, `l`, `r`, `h`. В последней строке предполагается, что $x \neq 0$. Эта строка соответствует равенству

$$f(x) = h(x, f(l(x)), f(r(x))).$$

Преобразования выполняются, пока последовательность не станет пуста. В этот момент в стеке окажется единственное число, которое и будет ответом.

Замечание. Последовательность по существу представляет собой стек отложенных заданий (вершина которого находится слева).

9. Разные алгоритмы на графах

9.1. Кратчайшие пути

В этом разделе рассматриваются различные варианты одной задач. Пусть имеется n городов, пронумерованных числами от 1 до n. Для каждой пары городов с номерами i, j в таблице a[i][j] хранится целое число — цена прямого авиабилета из города i в город j. Считается, что рейсы существуют между любыми городами, a[i][i] = 0 при всех i, a[i][j] может отличаться от a[j][i]. Наименьшей стоимостью проезда из i в j считается минимально возможная сумма цен билетов для маршрутов (в том числе с пересадками), ведущих из i в j. (Она не превосходит a[i][j], но может быть меньше.)

В предлагаемых ниже задачах требуется найти наименьшую стоимость проезда для некоторых пар городов при тех или иных ограничениях на массив a и на время работы алгоритма.

9.1.1. Предположим, что не существует замкнутых маршрутов, для которых сумма цен отрицательна. Доказать, что в этом случае маршрут с наименьшей стоимостью существует.

Решение. Маршрут длиной больше n всегда содержит цикл, поэтому минимум можно искать среди маршрутов длиной не более n, а их конечное число.

Во всех следующих задачах предполагается, что это условие (отсутствие циклов с отрицательной суммой) выполнено.

9.1.2. Найти наименьшую стоимость проезда из 1-го города во все остальные за время $O(n^3)$.

Решение. Обозначим через МинСт(1,s,k) наименьшую стоимость проезда из 1 в s менее чем с k пересадками. Тогда выполняется такое соотношение:

$$\texttt{МинСт}(1,s,k+1) = \min\Bigl(\texttt{МинСт(1,s,k)}, \min_{\texttt{i=1..n}} \texttt{МинСт(1,i,k)}+\texttt{a[i][s]}\Bigr)$$

Как отмечалось выше, искомым ответом является МинСт(1,i,n) для всех i = 1...n.

```
k:= 1;
for i := 1 to n do begin x[i] := a[1][i]; end;
{инвариант: x[i] = МинСт(1,i,k)}
while k <> n do begin
| for s := 1 to n do begin
| | y[s] := x[s];
| | for i := 1 to n do begin
| | | if y[s] > x[i]+a[i][s] then begin
| | | | y[s] := x[i]+a[i][s];
| | | end;
| | end
| | {y[s] = МинСт(1,s,k+1)}
| end;
| for i := 1 to n do begin x[s] := y[s]; end;
| k := k + 1;
end;
```

Приведенный алгоритм называют алгоритмом динамического программирования, или алгоритмом Форда–Беллмана.

9.1.3. Доказать, что программа останется правильной, если не заводить массива y, а производить изменения в самом массиве x (заменив в программе все вхождения буквы y на x и затем удалить ставшие лишними строки).

Решение. Инвариант будет таков:

$$\texttt{МинСт(1,i,n)} \leqslant \texttt{x[i]} \leqslant \texttt{МинСт(1,i,k)}.$$

Этот алгоритм может быть улучшен в двух отношениях: можно за то же время $O(n^3)$ найти наименьшую стоимость проезда $i \to j$ для *всех* пар i,j (а не только при $i = 1$), а можно сократить время работы до $O(n^2)$. Правда, в последнем случае нам потребуется, чтобы все цены a[i][j] были неотрицательны.

9.1.4. Найти наименьшую стоимость проезда $i \to j$ для всех i,j за время $O(n^3)$.

Решение. Для $k = 0 \ldots n$ через $A(i,j,k)$ обозначим наименьшую стоимость маршрута из i в j, если в качестве пересадочных разрешено использовать только пункты с номерами не больше k. Тогда

$$A(i,j,0) = a[i][j],$$

$$A(i,j,k+1) = \min\Big(A(i,j,k), A(i,k+1,k) + A(k+1,j,k)\Big)$$

(два варианта соответствуют неиспользованию и использованию пункта $k+1$ в качестве пересадочного; отметим, что в нем незачем бывать более одного раза).

Этот алгоритм называют алгоритмом Флойда.

9.1.5. Известно, что все цены неотрицательны. Найти наименьшую стоимость проезда $1 \to i$ для всех $i = 1 \ldots n$ за время $O(n^2)$.

Решение. В процессе работы алгоритма некоторые города будут *выделенными* (в начале — только город 1, в конце — все). При этом:

- для каждого выделенного города i хранится наименьшая стоимость пути $1 \to i$; при этом известно, что минимум достигается на пути, проходящем только через выделенные города;
- для каждого невыделенного города i хранится наименьшая стоимость пути $1 \to i$, в котором в качестве промежуточных используются только выделенные города.

Множество выделенных городов расширяется на основании следующего замечания: если среди всех невыделенных городов взять тот, для которого хранимое число минимально, то это число является истинной наименьшей стоимостью. В самом деле, пусть есть более короткий путь. Рассмотрим первый невыделенный город на этом пути — уже до него путь длиннее! (Здесь существенна неотрицательность цен.)

Добавив выбранный город к выделенным, мы должны скорректировать информацию, хранимую для невыделенных городов. При этом достаточно учесть лишь пути, в которых новый город является последним пунктом пересадки, а это легко сделать, так как минимальную стоимость проезда в новый город мы уже знаем.

При самом бесхитростном способе хранения множества выделенных городов (в булевском векторе) добавление одного города к числу выделенных требует времени $O(n)$.

Этот алгоритм называют алгоритмом Дейкстры.

Отыскание кратчайшего пути имеет естественную интерпретацию в терминах матриц. Пусть A — матрица цен одной авиакомпании, а B — матрица цен другой. Пусть мы хотим лететь с одной пересадкой, причём сначала самолётом компании A, а затем — компании B. Сколько нам придется заплатить, чтобы попасть из города `i` в город `j`?

9.1.6. Доказать, что эта матрица вычисляется по обычной формуле для произведения матриц, только вместо суммы надо брать минимум, а вместо умножения — сумму.

9.1.7. Доказать, что таким образом определенное произведение матриц ассоциативно.

9.1.8. Доказать, что задача о кратчайших путях эквивалентна вычислению A^{∞} для матрицы цен A: в последовательности $A, A^2, A^3, \ldots$ все элементы, начиная с некоторого, равны искомой матрице стоимостей кратчайших путей. (Если нет отрицательных циклов!)

9.1.9. Начиная с какого элемента можно гарантировать равенство в предыдущей задаче?

Обычное (не модифицированное) умножение матриц тоже может оказаться полезным, только матрицы должны быть другие. Пусть есть не все рейсы (как раньше), а только некоторые, `a[i][j]` равно `1`, если рейс есть, и `0`, если рейса нет. Возведем матрицу `a` (обычным образом) в степень `k` и посмотрим на её `i`-`j`-ый элемент.

9.1.10. Чему он равен?

Ответ. Числу различных способов попасть из `i` в `j` за `k` рейсов (с `k-1` пересадками).

Случай, когда есть не все рейсы, можно свести к исходному, введя фиктивные рейсы с бесконечно большой (или достаточно большой) стоимостью. Тем не менее возникает такой вопрос. Число реальных рейсов может быть существенно меньше n^2, поэтому интересны алгоритмы, которые работают эффективно в такой ситуации. Исходные данные естественно представлять тогда в такой форме: для каждого города известно число выходящих из него рейсов, их пункты назначения и цены.

9.1.11. Доказать, что алгоритм Дейкстры можно модифицировать так, чтобы для n городов и m рейсов (всего) он требовал не более $C(n + m)\log n$ операций.

[Указание. Что надо сделать на каждом шаге? Выбрать невыделенный город с минимальной стоимостью и скорректировать цены для всех городов, в которые из него есть маршруты. Если бы кто-то сообщал нам, для какого города стоимость минимальна, то хватило бы $C(n + m)$ действий. А поддержание сведений о том, какой элемент в массиве минимален (см. задачу на с. 124) обходится ещё в множитель $\log n$.]

9.2. Связные компоненты, поиск в глубину и ширину

Наиболее простой случай задачи о кратчайших путях — если все цены равны 0 или $+\infty$. Другими словами, мы интересуемся возможностью попасть из i в j, но за ценой не постоим. В других терминах: мы имеем ориентированный граф (картинку из точек, некоторые из которых соединены стрелками) и нас интересуют вершины, доступные из данной.

Для этого случая задачи о кратчайших путях приведенные в предыдущем разделе алгоритмы — не наилучшие. В самом деле, более быстрая рекурсивная программа решения этой задачи приведена в главе 7, а нерекурсивная — в главе 6. Сейчас нас интересует такая задача: не просто перечислить все вершины, доступные из данной, но перечислить их в определенном порядке. Два популярных случая — поиск в ширину и в глубину.

Поиск в ширину.

Надо перечислить все вершины ориентированного графа, доступные из данной, в порядке увеличения длины пути от неё. (Тем самым мы решим задачу о кратчайших путях, когда цены рёбер равны 1 или $+\infty$.)

9.2.1. Придумать алгоритм решения этой задачи с числом действий не более $C\cdot$ (число рёбер, выходящих из интересующих нас вершин).

Решение. Эта задача рассматривалась в главе 6, с. 123. Здесь мы приведём подробное решение. Пусть `num[i]` — количество рёбер, выходящих из `i`, `out[i][1]`, ..., `out[i][num[i]]` — вершины, куда ведут рёбра. Вот программа, приведённая ранее:

```
procedure Доступные (i: integer);
  {напечатать все вершины, доступные из i, включая i}
  var  X: подмножество 1..n;
       P: подмножество 1..n;
       q, v, w: 1..n;
       k: integer;
begin
  ...сделать X, P пустыми;
  writeln (i);
  ...добавить i к X, P;
  {(1) P = множество напечатанных вершин; P содержит i;
   (2) напечатаны только доступные из i вершины;
   (3) X - подмножество P;
   (4) все напечатанные вершины, из которых выходит
       ребро в ненапечатанную вершину, принадлежат X}
  while X непусто do begin
    ...взять какой-нибудь элемент X в v;
    for k := 1 to num [v] do begin
      w := out [v][k];
      if w не принадлежит P then begin
        writeln (w);
        добавить w в P;
        добавить w в X;
      end;
    end;
  end;
end;
```

Тогда нам было безразлично, какой именно элемент множества `X` выбирается. Если мы будем считать `X` очередью (первым пришёл — первым ушёл), то эта программа напечатает все вершины, доступные из `i`, в порядке возрастания их расстояния от `i` (числа рёбер на кратчайшем пути из `i`). Докажем это.

Обозначим через $V(k)$ множество всех вершин, расстояние которых от `i` (в описанном смысле) равно k. Имеет место такое соотношение:

$$V(k+1) = (\text{концы ребер с началами в } V(k)) \setminus (V(0) \cup \ldots\ldots \cup V(k))$$

Докажем, что для любого $k = 0, 1, 2\ldots$ в ходе работы программы будет такой момент (после очередной итерации цикла `while`), когда

в очереди стоят все элементы $V(k)$ и только они;
напечатаны все элементы $V(0), \ldots, V(k)$

(Для $k = 0$ — это состояние перед циклом.) Рассуждая по индукции, предположим, что в очереди скопились все элементы $V(k)$. Они будут просматриваться в цикле, пока не кончатся (поскольку новые элементы добавляются в конец, они не перемешаются со старыми). Концы ведущих из них рёбер, если они уже не напечатаны, печатаются и ставятся в очередь — то есть всё как в записанном выше соотношении для $V(k+1)$. Так что когда все старые элементы кончатся, в очереди будут стоять все элементы $V(k+1)$.

Поиск в глубину.

Рассматривая поиск в глубину, удобно представлять себе ориентированный граф как образ дерева. Более точно, пусть есть ориентированный граф, одна из вершин которого выделена. Будем предполагать, что все вершины доступны из выделенной по ориентированным путям. Построим дерево, которое можно было бы назвать «универсальным накрытием» нашего графа. Его корнем будет выделенная вершина графа. Из корня выходят те же стрелки, что и в графе — их концы будут сыновьями корня. Из них в дереве выходят те же стрелки, что и в графе и так далее. Разница между графом и деревом в том, что пути в графе, ведущие в одну и ту же вершину, в дереве «расклеены». В других терминах: вершина дерева — это путь в графе, выходящий из корня. Её сыновья — это пути, продолженные на одно ребро. Заметим, что дерево бесконечно, если в графе есть ориентированные циклы.

Имеется естественное отображение дерева в граф (вершин в вершины). При этом каждая вершина графа имеет столько прообразов, сколько путей в неё ведет. Поэтому обход дерева (посещение его вершин в том или ином порядке) одновременно является и обходом графа — только каждая вершина посещается многократно.

Будем предполагать, что для каждой вершины графа выходящие из неё рёбра упорядочены (например, пронумерованы). Тем самым для каждой вершины дерева выходящие из неё рёбра также упорядочены. Будем обходить дерево так: сначала корень, а потом поддеревья (в порядке ведущих в них рёбер). Такой обход дерева рассматривался нами в главе 7. Ему соответствует обход графа. Если выкинуть из этого обхода повторные посещения уже посещённых вершин, то получится то, что называется «поиск в глубину».

Другими словами, на путях, выходящих из выделенной вершины, введём порядок: путь предшествует своему продолжению; если два пути расходятся в некоторой вершине, то меньшим считается тот, который выходит из нее по меньшему ребру. Вершины теперь упорядочиваются в соответствии с минимальными путями, в них ведущими. Обход вершин графа в указанном порядке называется *поиском в глубину*.

9.2.2. Написать программу поиска в глубину.

[Указание. Возьмём программу обхода дерева (корень $\to$ левое поддерево $\to$ правое поддерево) из главы 7 или из главы 8 и используем её применительно к обстоятельствам. Главное изменение: не надо посещать вершины повторно. Так что если мы попали в уже посещённую вершину, то можно с ней ничего не делать. (Если путь не минимален среди ведущих в данную вершину, то и все его продолжения не минимальны — их просматривать не надо).]

Замечание. Напомним, что в главе 8 упоминались две возможности устранения рекурсии в программе обхода дерева (с.155). Оба варианта можно использовать для поиска в глубину.

Поиск в глубину лежит в основе многих алгоритмов на графах, порой в несколько модифицированном виде.

9.2.3. Неориентированный граф называется *двудольным*, если его вершины можно раскрасить в два цвета так, что концы любого ребра — разного цвета. Составить алгоритм проверки, является ли заданный граф двудольным, в котором число действий не превосходит C $\cdot$ (число рёбер+число вершин).

[Указание. (а) Каждую связную компоненту можно раскрашивать отдельно. (б) Выбрав цвет одной вершины и обходя ее связную компоненту, мы определяем единственно возможный цвет остальных.]

Замечание. В этой задаче безразлично, производить поиск в ширину или в глубину.

9.2.4. Составить нерекурсивный алгоритм топологической сортировки ориентированного графа без циклов. (Рекурсивный алгоритм смотри на с. 139.)

Решение. Предположим, что граф имеет вершины с номерами `1...n`, для каждой вершины `i` известно число `num[i]` выходящих из неё рёбер и номера вершин `dest[i][1],...,dest[i][num[i]]`, в которые эти рёбра ведут. Будем условно считать, что рёбра перечислены «слева направо»: левее то ребро, у которого номер меньше. Нам надо напечатать все вершины в таком порядке, чтобы конец любого ребра был напечатан перед его началом. Мы предполагаем, что в графе нет ориентированных циклов — иначе такое невозможно.

Для начала добавим к графу вершину `0`, из которой ребра ведут в вершины `1,...,n`. Если её удастся напечатать с соблюдением правил, то тем самым все вершины будут напечатаны.

Алгоритм хранит путь, выходящий из нулевой вершины и идущий по рёбрам графа. Переменная `l` отводится для длины этого пути. Путь образован вершинами `vert[1]...vert[l]` и рёбрами, имеющими номера `edge[1]...edge[l]`. Номер `edge[s]` относится к нумерации рёбер, выходящих из вершины `vert[s]`. Тем самым для всех `s` должны выполняться неравенство

$$\texttt{edge[s]} \leqslant \texttt{num[vert[s]]}$$

и равенство

$$\texttt{vert[s+1]} = \texttt{dest [vert[s]] [edge[s]]}.$$

Заметим, что конец последнего ребра нашего пути (то есть вершина `dest[vert[l]][edge[l]]`, не включается в массив `vert`. Кроме того, для последнего ребра мы делаем исключение, разрешая ему указывать «в пустоту», т. е. разрешаем `edge[l]` равняться `num[vert[l]]+1`.

В процессе работы алгоритм будет печатать номера вершин, при этом соблюдая требование «вершина напечатана только после тех вершин, в которые из неё ведут рёбра». Кроме того, будет выполняться такое требование (И):

> вершины пути, кроме последней (`vert[1]...vert[l]`) не напечатаны, но свернув с пути налево, мы немедленно упираемся в напечатанную вершину

Вот что получается:

```
l:=1; vert[1]:=0; edge[1]:=1;
while not( (l=1) and (edge[1]=n+1)) do begin
| if edge[l]=num[vert[l]]+1 then begin
| | {путь кончается в пустоте, поэтому все вершины,
| |     следующие за vert[l], напечатаны - можно
| |     печатать vert[l]}
| | writeln (vert[l]);
| | l:=l-1; edge[l]:=edge[l]+1;
| end else begin
| |  {edge[l] <= num[vert[l]], путь кончается в
| |     вершине}
| |  lastvert:= dest[vert[l]][edge[l]]; {последняя}
| |  if lastvert напечатана then begin
| |  | edge[l]:=edge[l]+1;
| |  end else begin
| |  | l:=l+1; vert[l]:=lastvert; edge[l]:=1;
| |  end;
| end;
end;
{путь сразу же ведет в пустоту, поэтому все вершины
 левее, то есть 1..n, напечатаны}
```

9.2.5. Доказать, что если в графе нет циклов, то этот алгоритм заканчивает работу.

Решение. Пусть это не так. Каждая вершина может печататься только один раз, так что с некоторого момента вершины не печатаются. В графе без циклов длина пути ограничена (вершина не может входить в путь дважды), поэтому подождав ещё, мы можем дождаться момента, после которого путь не удлиняется. После этого может разве что увеличиваться `edge[l]` — но и это не беспредельно.

9.2.6. Доказать, что время работы этого алгоритма не превосходит O(число вершин + число рёбер).

9.2.7. Как модифицировать алгоритм так, чтобы он отыскивал один из циклов, если таковые имеются, и производил топологическую сортировку, если циклов нет?

10. Сопоставление с образцом

10.1. Простейший пример

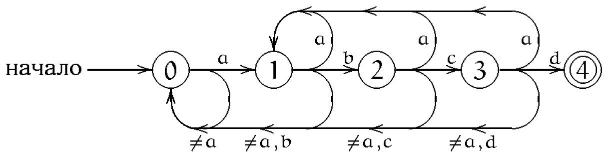
10.1.1. Имеется последовательность символов `x[1]...x[n]`. Определить, имеются ли в ней идущие друг за другом символы `abcd`. (Другими словами, требуется выяснить, есть ли в слове `x[1]...x[n]` подслово `abcd`.)

Решение. Имеется примерно `n` (если быть точным, `n-3`) позиций, на которых может находиться искомое подслово в исходном слове. Для каждой из позиций можно проверить, действительно ли там оно находится, сравнив четыре символа. Однако есть более эффективный способ. Читая слово `x[1]...x[n]` слева направо, мы ожидаем появления буквы `a`. Как только она появилась, мы ищем за ней букву `b`, затем `c`, и, наконец, `d`. Если наши ожидания оправдываются, то слово `abcd` обнаружено. Если же какая-то из нужных букв не появляется, мы оказываемся у разбитого корыта и начинаем всё сначала.

Этот простой алгоритм можно описать в разных терминах. Используя терминологию так называемых *конечных автоматов*, можно сказать, что при чтении слова `x` слева направо мы в каждый момент находимся в одном из следующих состояний: «начальное» (0), «сразу после `a`»(1), «сразу после `ab`» (2), «сразу после `abc`» (3) и «сразу после `abcd`» (4). Читая очередную букву, мы переходим в следующее состояние по правилу

Текущее состояние	Очередная буква	Новое состояние
0	a	1
0	кроме a	0
1	b	2
1	a	1
1	кроме a,b	0
2	c	3
2	a	1
2	кроме a,c	0
3	d	4
3	a	1
3	кроме a,d	0

Как только мы попадем в состояние 4, работа заканчивается.



Соответствующая программа очевидна (мы указываем новое состояние, даже если оно совпадает со старым; эти строки можно опустить):

```
i:=1; state:=0;
{i - первая непрочитанная буква, state - состояние}
while (i <> n+1) and (state <> 4) do begin
| if state = 0 then begin
| | if x[i] = a then begin
| | | state:= 1;
| | end else begin
| | | state:= 0;
| | end;
| end else if state = 1 then begin
| | if x[i] = b then begin
| | | state:= 2;
| | end else if x[i] = a then begin
| | | state:= 1;
| | end else begin
| | | state:= 0;
| | end;
| end else if state = 2 then begin
| | if x[i] = c then begin
| | | state:= 3;
| | end else if x[i] = a then begin
| | | state:= 1;
| | end else begin
| | | state:= 0;
| | end;
```

```
  end else if state = 3 then begin
    if x[i] = d then begin
      state:= 4;
    end else if x[i] = a then begin
      state:= 1;
    end else begin
      state:= 0;
    end;
  end;
end;
answer := (state = 4);
```

Иными словами, мы в каждый момент храним информацию о том, какое максимальное начало нашего образца abcd является концом прочитанной части. (Его длина и есть то «состояние», о котором шла речь.)

Терминология, нами используемая, такова. *Слово* — это любая последовательность символов из некоторого фиксированного конечного множества. Это множество называется *алфавитом*, его элементы — *буквами*. Если отбросить несколько букв с конца слова, останется другое слово, называемое *началом* первого. Любое слово также считается своим началом. *Конец* слова — то, что останется, если отбросить несколько первых букв. Любое слово считается своим концом. *Подслово* — то, что останется, если отбросить буквы и с начала, и с конца. (Другими словами, подслова — это концы начал, или, что то же, начала концов.)

В терминах индуктивных функций (см. раздел 1.3) ситуацию можно описать так: рассмотрим функцию на словах, которая принимает два значения «истина» и «ложь» и истинна на словах, имеющих abcd своим подсловом. Эта функция не является индуктивной, но имеет индуктивное расширение

$$x \mapsto \text{длина максимального начала слова abcd, являющегося концом } x$$

10.2. Повторения в образце — источник проблем

10.2.1. Можно ли в предыдущих рассуждениях заменить слово abcd на произвольное слово?

Решение. Нет, и проблемы связаны с тем, что в образце могут быть повторяющиеся буквы. Пусть, например, мы ищем вхождения слова `ababc`. Вот появилась буква `a`, за ней идет `b`, за ней идет `a`, затем снова `b`. В этот момент мы с нетерпением ждем буквы `c`. Однако — к нашему разочарованию — вместо нее появляется другая буква, и наш образец `ababc` не обнаружен. Однако нас может ожидать утешительный приз: если вместо `c` появилась буква `a`, то не всё потеряно: за ней могут последовать буквы `b` и `c`, и образец-таки будет найден.

Вот картинка, поясняющая сказанное:

```
x y z a b a b a b c ... ←  входное слово
      a b a b c         ←  мы ждали образца здесь
          a b a b c     ←  а он оказался здесь
```

Таким образом, к моменту

```
x y z a b a b |       ←  входное слово
      a b a b | c     ←  мы ждали образца здесь
          a b | a b c ←  а он оказался здесь
```

есть два возможных положения образца, каждое из которых подлежит проверке. Тем не менее по-прежнему возможен конечный автомат, читающий входное слово буква за буквой и переходящий из состояния в состояние в зависимости от прочитанных букв.

10.2.2. Указать состояния соответствующего автомата и таблицу перехода (новое состояние в зависимости от старого и читаемой буквы).

Решение. По-прежнему состояния будут соответствовать наибольшему началу образца, являющемуся концом прочитанной части слова. Их будет шесть: 0, 1 (a), 2 (ab), 3 (aba), 4 (abab), 5 (ababc). Таблица перехода:

Текущее состояние	Очередная буква	Новое состояние
0	a	1 (a)
0	кроме a	0
1 (a)	b	2 (ab)
1 (a)	a	1 (a)
1 (a)	кроме a,b	0
2 (ab)	a	3 (aba)
2 (ab)	кроме a	0
3 (aba)	b	4 (abab)
3 (aba)	a	1 (a)
3 (aba)	кроме a,b	0
4 (abab)	c	5 (ababc)
4 (abab)	a	3 (aba)
4 (abab)	кроме a,c	0

Для проверки посмотрим, к примеру, на вторую снизу строку. Если прочитанная часть кончалась на abab, а затем появилась буква a, то теперь прочитанная часть кончается на ababa. Наибольшее начало образца (ababc), являющееся её концом — это aba.

Философский вопрос: мы говорили, что трудность состоит в том, что есть несколько возможных положений образца, каждое из которых может оказаться истинным. Им соответствуют несколько начал образца, являющихся концами входного слова. Но конечный автомат помнит лишь самое длинное из них. Как же остальные?

Философский ответ. Дело в том, что самое длинное из них определяет все остальные — это его концы, одновременно являющиеся его началами.

Не составляет труда для любого конкретного образца написать программу, осуществляющую поиск этого образца описанным способом. Однако хотелось бы написать программу, которая ищет произвольный образец в произвольном слове. Это можно делать в два этапа: сначала по образцу строится таблица переходов конечного автомата, а затем читается входное слово и состояние преобразуется в соответствии с этой таблицей. Подобный метод часто используется для более сложных задач поиска (см. далее), но для поиска подслова существует более простой и эффективный алгоритм, называемый алгоритмом Кнута–Морриса–Пратта. (Ранее сходные идеи были предложены Ю.В. Матиясевичем.) Но прежде нам понадобятся некоторые вспомогательные утверждения.

10.3. Вспомогательные утверждения

Для произвольного слова X рассмотрим все его начала, одновременно являющиеся его концами, и выберем из них самое длинное. (Не считая, конечно, самого слова X.) Будем обозначать его $l(X)$.

Примеры: $l(\mathtt{aba}) = \mathtt{a}$, $l(\mathtt{abab}) = \mathtt{ab}$, $l(\mathtt{ababa}) = \mathtt{aba}$, $l(\mathtt{abc}) =$ пустое слово.

10.3.1. Доказать, что все слова $l(X)$, $l(l(X))$, $l(l(l(X)))$ и т.д. являются началами слова X.

Решение. Каждое из них (согласно определению) является началом предыдущего.

По той же причине все они являются концами слова X.

10.3.2. Доказать, что последовательность предыдущей задачи обрывается (на пустом слове).

Решение. Каждое слово короче предыдущего.

10.3.3. Доказать, что любое слово, одновременно являющееся началом и концом слова X (кроме самого X) входит в последовательность $l(X), l(l(X)), \ldots$

Решение. Пусть слово Y есть одновременно начало и конец X. Слово $l(X)$ — самое длинное из таких слов, так что Y не длиннее $l(X)$. Оба эти слова являются началами X, поэтому более короткое из них является началом более длинного: Y есть начало $l(X)$. Аналогично, Y есть конец $l(X)$. Рассуждая по индукции, можно предполагать, что утверждение задачи верно для всех слов короче X, в частности, для слова $l(X)$. Так что слово Y, являющееся концом и началом $l(X)$, либо равно $l(X)$, либо входит в последовательность $l(l(X))$, $l(l(l(X)))$,..., что и требовалось доказать.

10.4. Алгоритм Кнута–Морриса–Пратта

Алгоритм Кнута–Морриса–Пратта (КМП) получает на вход слово

$$\texttt{X} = \texttt{x[1]x[2]}\ldots\texttt{x[n]}$$

и просматривает его слева направо буква за буквой, заполняя при этом массив натуральных чисел `l[1]`...`l[n]`, где

$$\texttt{l[i]} = \text{длина слова } l(\texttt{x[1]}\ldots\texttt{x[i]})$$

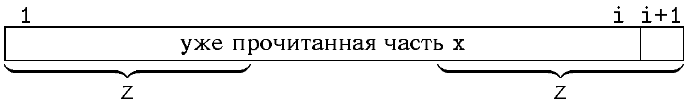
(функция l определена в предыдущем пункте). Словами: `l[i]` есть длина наибольшего начала слова `x[1]`...`x[i]`, одновременно являющегося его концом.

10.4.1. Какое отношение всё это имеет к поиску подслова? Другими словами, как использовать алгоритм КМП для определения того, является ли слово `A` подсловом слова `B`?

Решение. Применим алгоритм КМП к слову `A#B`, где `#` — специальная буква, не встречающаяся ни в `A`, ни в `B`. Слово `A` является подсловом слова `B` тогда и только тогда, когда среди чисел в массиве `l` будет число, равное длине слова `A`.

10.4.2. Описать алгоритм заполнения таблицы `l[1]`...`l[n]`.

Решение. Предположим, что первые `i` значений `l[1]`...`l[i]` уже найдены. Мы читаем очередную букву слова (т.е. `x[i+1]`) и должны вычислить `l[i+1]`.



Другими словами, нас интересуют начала Z слова `x[1]`...`x[i+1]`, одновременно являющиеся его концами — из них нам надо выбрать самое длинное. Откуда берутся эти начала? Каждое из них (не считая пустого) получается из некоторого слова Z' приписыванием буквы `x[i+1]`. Слово

Z' является началом и концом слова `x[1]...x[i]`. Однако не любое слово, являющееся началом и концом слова `x[1]...x[i]`, годится — надо, чтобы за ним следовала буква `x[i+1]`.

Получаем такой рецепт отыскания слова Z. Рассмотрим все начала слова `x[1]...x[i]`, являющиеся одновременно его концами. Из них выберем подходящие — те, за которыми идет буква `x[i+1]`. Из подходящих выберем самое длинное. Приписав в его конец `x[i+1]`, получим искомое слово Z.

Теперь пора воспользоваться сделанными нами приготовлениями и вспомнить, что все слова, являющиеся одновременно началами и концами данного слова, можно получить повторными применениями к нему функции l из предыдущего раздела. Вот что получается:

```
i:=1; l[1]:= 0;
{таблица l[1]..l[i] заполнена правильно}
while i <> n do begin
| len := l[i]
| {len - длина начала слова x[1]..x[i], которое является
|    его концом; все более длинные начала оказались
|    неподходящими}
| while (x[len+1] <> x[i+1]) and (len > 0) do begin
| | {начало не подходит, применяем к нему функцию l}
| | len := l[len];
| end;
| {нашли подходящее или убедились в отсутствии}
| if x[len+1] = x[i+1] do begin
| | {x[1]..x[len] - самое длинное подходящее начало}
| | l[i+1] := len+1;
| end else begin
| | {подходящих нет}
| | l[i+1] := 0;
| end;
| i := i+1;
end;
```

10.4.3. Доказать, что число действий в приведённом только что алгоритме не превосходит $C\mathtt{n}$ для некоторой константы C.

Решение. Это не вполне очевидно: обработка каждой очередной буквы может потребовать многих итераций во внутреннем цикле. Однако каждая такая итерация уменьшает `len` по крайней мере на `1`, и в этом случае `l[i+1]` окажется заметно меньше `l[i]`. С другой стороны, при увеличении `i` на единицу величина `l[i]` может возрасти не более чем на `1`, так что часто и сильно убывать она не может — иначе убывание не будет скомпенсировано возрастанием.

Более точно, можно записать неравенство

$$\mathtt{l[i+1]} \leqslant \mathtt{l[i]} - (\text{число итераций на } \mathtt{i}\text{-м шаге}) + 1$$

или

$$(\text{число итераций на } \mathtt{i}\text{-м шаге}) \leqslant \mathtt{l[i]} - \mathtt{l[i+1]} + 1$$

Остаётся сложить эти неравенства по всем `i` и получить оценку сверху для общего числа итераций.

10.4.4. Будем использовать этот алгоритм, чтобы выяснить, является ли слово `X` длины `n` подсловом слова `Y` длины `m`. (Как это делать с помощью специального разделителя `#`, описано выше.) При этом число действий будет не более $C(\mathtt{n}+\mathtt{m})$, и используемая память тоже. Придумать, как обойтись памятью не более $C\mathtt{n}$ (что может быть существенно меньше, если искомый образец короткий, а слово, в котором его ищут — длинное).

Решение. Применяем алгоритм КМП к слову `A#B`. При этом вычисление значений `l[1]`, ..., `l[n]` проводим для слова `X` длины `n` и запоминаем эти значения. Дальше мы помним только значение `l[i]` для текущего `i` — кроме него и кроме таблицы `l[1]`...`l[n]`, нам для вычислений ничего не нужно.

На практике слова `X` и `Y` могут не находиться подряд, поэтому просмотр слова `X` и затем слова `Y` удобно оформить в виде разных циклов. Это избавляет также от хлопот с разделителем.

10.4.5. Написать соответствующий алгоритм (проверяющий, является ли слово `X` = `x[1]`...`x[n]` подсловом слова `Y` = `y[1]`...`y[m]`).

Решение. Сначала вычисляем таблицу `l[1]`...`l[n]` как раньше. Затем пишем такую программу:

```
j:=0; len:=0;
{len - длина максимального начала слова X, одновременно
      являющегося концом слова y[1]..y[j]}
while (len <> n) and (j <> m) do begin
| while (x[len+1] <> y[j+1]) and (len > 0) do begin
| | {начало не подходит, применяем к нему функцию l}
| | len := l[len];
| end;
| {нашли подходящее или убедились в отсутствии}
| if x[len+1] = y[j+1] do begin
| | {x[1]..x[len] - самое длинное подходящее начало}
| | len := len+1;
| end else begin
| | {подходящих нет}
| | len := 0;
| end;
| j := j+1;
end;
{если len=n, слово X встретилось; иначе мы дошли до конца
   слова Y, так и не встретив X}
```

10.5. Алгоритм Бойера–Мура

Этот алгоритм делает то, что на первый взгляд кажется невозможным: в типичной ситуации он читает лишь небольшую часть всех букв слова, в котором ищется заданный образец. Как так может быть? Идея проста. Пусть, например, мы ищем образец abcd. Посмотрим на четвертую букву слова: если, к примеру, это буква e, то нет никакой необходимости читать первые три буквы. (В самом деле, в образце буквы e нет, поэтому он может начаться не раньше пятой буквы.)

Мы приведем самый простой вариант этого алгоритма, который не гарантирует быстрой работы во всех случаях. Пусть x[1]...x[n] — образец, который надо искать. Для каждого символа s найдем самое правое его вхождение в слово X, то есть наибольшее k, при котором x[k] $=$ s. Эти сведения будем хранить в массиве pos[s]; если символ s вовсе не встречается, то нам будет удобно положить pos[s] = 0 (мы увидим дальше, почему).

10.5.1. Как заполнить массив pos?

Решение.

```
положить все pos[s] равными 0
for i:=1 to n do begin
  pos[x[i]]:=i;
end;
```

В процессе поиска мы будем хранить в переменной last номер буквы в слове, против которой стоит последняя буква образца. Вначале last = n (длина образца), затем last постепенно увеличивается.

```
last:=n;
{все предыдущие положения образца уже проверены}
while last <= m do begin {слово не кончилось}
| if x[m] <> y[last] then begin {последние буквы разные}
| | last := last + (n - pos[y[last]]);
| | {n - pos[y[last]]  - это минимальный сдвиг образца,
| |    при котором напротив y[last] встанет такая же
| |    буква в образце. Если такой буквы нет вообще,
| |    то сдвигаем на всю длину образца}
| end else begin
| | если нынешнее положение подходит, т.е. если
| | x[1]..x[n] = y[last-n+1]..y[last],
| | то сообщить о совпадении;
| | last := last+1;
| end;
end;
```

Знатоки рекомендуют проверку совпадения проводить справа налево, т.е. начиная с последней буквы образца (в которой совпадение заведомо есть). Можно также немного сэкономить, произведя вычитание заранее и храня не pos[s], а n-pos[s], т.е. число букв в образце справа от последнего вхождения буквы s.

Возможны разные модификации этого алгоритма. Например, можно строку `last:=last+1` заменить на `last:=last+(n-u)`, где `u` — координата второго справа вхождения буквы `x[n]` в образец.

10.5.2. Как проще всего учесть это в программе?

Решение. При построении таблицы `pos` написать

```
for i:=1 to n-1 do...
```

(далее как раньше), а в основной программе вместо `last:=last+1` написать

```
last:= last+n-pos[y[last]];
```

Приведенный нами упрощённый вариант алгоритма Бойера–Мура в некоторых случаях требует существенно больше `n` действий (число действий порядка `mn`), проигрывая алгоритму Кнута–Морриса–Пратта.

10.5.3. Привести пример ситуации, в которой образец не входит в слово, но алгоритму требуется порядка `mn` действий, чтобы это установить.

Решение. Пусть образец имеет вид `baaa...aa`, а само слово состоит только из букв `a`. Тогда на каждом шаге несоответствие выясняется лишь в последний момент.

Настоящий (не упрощённый) алгоритм Бойера–Мура гарантирует, что число действий не превосходит $C(\mathtt{m}+\mathtt{n})$ в худшем случае. Он использует идеи, близкие к идеям алгоритма Кнута–Морриса–Пратта. Представим себе, что мы сравнивали образец со входным словом, идя справа налево. При этом некоторый кусок Z (являющийся концом образца) совпал, а затем обнаружилось различие: перед Z в образце стоит не то, что во входном слове. Что можно сказать в этот момент о входном слове? В нем обнаружен фрагмент, равный Z, а перед ним стоит не та буква, что в образце. Эта информация может позволить сдвинуть образец на несколько позиций вправо без риска пропустить его вхождение. Эти сдвиги следует вычислить заранее для каждого конца Z нашего образца. Как говорят знатоки, всё это (вычисление таблицы сдвигов и её использование) можно уложить в $C(\mathtt{m}+\mathtt{n})$ действий.

10.6. Алгоритм Рабина

Этот алгоритм основан на простой идее. Представим себе, что в слове длины m мы ищем образец длины n. Вырежем окошечко размера n и будем двигать его по входному слову. Нас интересует, не совпадает ли слово в окошечке с заданным образцом. Сравнивать по буквам долго. Вместо этого фиксируем некоторую функцию, определённую на словах длины n. Если значения этой функции на слове в окошечке и на образце различны, то совпадения нет. Только если значения одинаковы, нужно проверять совпадение по буквам.

Что мы выигрываем при таком подходе? Казалось бы, ничего — ведь чтобы вычислить значение функции на слове в окошечке, всё равно нужно прочесть все буквы этого слова. Так уж лучше их сразу сравнить с образцом. Тем не менее выигрыш возможен, и вот за счёт чего. При сдвиге окошечка слово не меняется полностью, а лишь добавляется буква в конце и убирается в начале. Хорошо бы, чтобы по этим данным можно было рассчитать, как меняется функция.

10.6.1. Привести пример удобной для вычисления функции.

Решение. Заменим все буквы в слове и образце их номерами, представляющими собой целые числа. Тогда удобной функцией является сумма цифр. (При сдвиге окошечка нужно добавить новое число и вычесть пропавшее.)

Для каждой функции существуют слова, к которым она применима плохо. Зато другая функция в этом случае может работать хорошо. Возникает идея: надо запасти много функций и в начале работы алгоритма выбирать из них случайную. (Тогда враг, желающий подгадить нашему алгоритму, не будет знать, с какой именно функцией ему бороться.)

10.6.2. Привести пример семейства удобных функций.

Решение. Выберем некоторое число p (желательно простое, смотри далее) и некоторый вычет x по модулю p. Каждое слово длины n будем рассматривать как последовательность целых чисел (заменив буквы кодами). Эти числа будем рассматривать как коэффициенты многочлена степени $n-1$ и вычислим значение этого многочлена по модулю p в точке x. Это и будет одна из функций семейства (для каждой пары p и x получается, таким образом, своя функция). Сдвиг окошка на 1 соответствует вычитанию старшего члена (x^{n-1} следует вычислить заранее), умножению на x и добавлению свободного члена.

Следующее соображение говорит в пользу того, что совпадения не слишком вероятны. Пусть число p фиксировано и к тому же простое, а X и Y — два различных слова длины n. Тогда им соответствуют различные многочлены (мы предполагаем, что коды всех букв различны — это возможно, если p больше числа букв алфавита). Совпадение значений функции означает, что в точке x эти два различных многочлена совпадают, то есть их разность обращается в 0. Разность есть многочлен степени $n-1$ и имеет не более $n-1$ корней. Таким образом, если n много меньше p, то случайному x мало шансов попасть в неудачную точку.

10.7. Более сложные образцы и автоматы

Мы можем искать не конкретное слово, а подслова заданного вида. Например, можно искать слова вида `a?b`, где вместо `?` может стоять любая буква (иными словами, нас интересует буква `b` на расстоянии 2 после буквы `a`).

10.7.1. Указать конечный автомат, проверяющий, есть ли во входном слове фрагмент вида `a?b`.

Решение. Читая слово, следует помнить, есть ли буква `a` на последнем месте и на предпоследнем — пока не встретим искомый фрагмент. Автомат имеет состояния 00, 01, 10, 11, их смысл таков:

00	на предпоследнем и последнем местах нет `a`
01	на предпоследнем нет, на последнем есть
10	не предпоследнем есть, на последнем нет
11	есть и там, и там

Таблица переходов автомата:

Текущее состояние	Очередная буква	Новое состояние
00	a	01
00	не a	00
01	a	11
01	не a	10
10	a	01
10	b	найдено
10	не a и не b	00
11	a	11
11	b	найдено
11	не a и не b	10

Другой стандартный знак в образце — это звездочка (*), на место которой может быть подставлено любое слово. Например, образец ab*cd означает, что мы ищем подслово ab, за которым следует что угодно, а затем (на любом расстоянии) идёт cd.

10.7.2. Указать конечный автомат, проверяющий, есть ли во входном слове образец ab*cd (в описанном только что смысле).

Решение.

Текущее состояние	Очередная буква	Новое состояние
начальное	a	a
начальное	не a	начальное
a	b	ab
a	a	a
a	не a и не b	начальное
ab	c	abc
ab	не c	ab
abc	d	найдено
abc	c	abc
abc	не c и не d	ab

Еще один вид поиска — это поиск любого из слов некоторого списка.

10.7.3. Дан список слов $X_1, \ldots, X_k$ и слово Y. Определить, входит ли хотя бы одно из слов X_i в слово Y (как подслово). Количество действий не должно превосходить константы, умноженной на суммарную длину всех слов (из списка и того, в котором происходит поиск).

Решение. Очевидный способ состоит в том, чтобы каждое слово из списка проверять отдельно (с помощью одного из рассмотренных алгоритмов). Однако при этом мы не укладываемся в заданное число действий (из-за умножения k на длину слова Y).

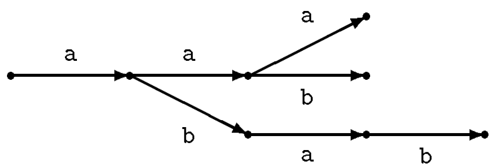
Посмотрим на дело с другой стороны. Каждому образцу из списка соответствует конечный автомат с некоторым множеством состояний. Эти автоматы можно объединить в один, множеством состояний которого будет произведение множеств состояний всех тех автоматов. Это — очень большое множество. Однако на самом деле большинство его элементов недоступны (не могут появиться при чтении входного слова) и за счет этого получается экономия. Примерно эту идею (но в изменённом виде) мы и будем использовать.

Вспомним алгоритм Кнута–Морриса–Пратта. В нём, читая входное слово, мы хранили наибольшее начало образца, являющееся концом прочитанной части. Теперь нам следует хранить для каждого из образцов наибольшее его начало, являющееся концом прочитанной части. Решающим оказывается такое замечание: достаточно хранить самое длинное из них — все остальные по нему восстанавливаются (как наибольшие начала образцов, являющиеся его концами).

Склеим все образцы в дерево, объединив их совпадающие начальные участки. Например, набору образцов

$$\{\mathtt{aaa}, \mathtt{aab}, \mathtt{abab}\}$$

соответствует дерево



Формально говоря, вершинами дерева являются все начала всех образцов, а сыновья вершины получаются приписыванием буквы.

Читая входное слово, мы двигаемся по этому дереву: текущая вершина — это наибольшая (самая правая) из вершин, являющихся концом прочитанной части (= наибольший конец прочитанной части, являющийся началом одного из образцов).

Определим функцию l, аргументами и значениями которой являются вершины дерева. Именно, $l(P)$ = наибольшая вершина дерева, являющаяся концом P. (Напомним, вершины дерева — это слова.) Нам понадобится такое утверждение:

10.7.4. Пусть P — вершина дерева. Докажите, что множество всех вершин, являющихся концами P, равно $\{l(P), l(l(P)), \ldots\}$

Решение. См. доказательство аналогичного утверждения для алгоритма Кнута–Морриса–Пратта.

Теперь ясно, что нужно делать, находясь в вершине P и читая букву z входного слова. Надо просматривать последовательно вершины P, $l(P)$, $l(l(P))$,..., пока не обнаружится такая, из которой выходит стрелка с буквой z. Та вершина, в которую эта стрелка ведёт, и будет нашим следующим положением.

Остается понять, как для каждой вершины дерева вычислить указатель на значение функции l в этой вершине. Это делается как раньше, при этом значения l для более коротких слов используются при вычислении очередного значения функции l. Это означает, что вершины дерева следует просматривать в порядке возрастания их длины. Нетрудно понять, что всё это можно уложить в требуемое число действий (хотя константа зависит от числа букв в алфавите). Относящиеся к этому подробности см. в главе 9.

Можно поинтересоваться, какие свойства слов распознаются с помощью конечных автоматов. Оказывается, что существует просто описываемый класс образцов, задающий все такие свойства — класс *регулярных выражений*.

Определение. Пусть фиксирован конечный алфавит Γ, не содержащий символов Λ, ε, (,), * и | (они будут использоваться для построения регулярных выражений и не должны перемешиваться с буквами). Регулярные выражения строятся по таким правилам:

(а) буква алфавита Γ — регулярное выражение;

(б) символы Λ, ε — регулярные выражения;

(в) если $A, B, C, \ldots, E$ — регулярные выражения, то $(ABC\ldots E)$ — регулярное выражение;

(г) если $A, B, C, \ldots, E$ — регулярные выражения, то $(A|B|C|\ldots|E)$ — регулярное выражение;

(д) если A — регулярное выражение, то $A*$ — регулярное выражение.

Каждое регулярное выражение задает множество слов в алфавите Γ по таким правилам:

(а) букве соответствует одноэлементное множество, состоящее из однобуквенного слова, состоящего из этой буквы;

(б) символу ε соответствует пустое множество, а символу Λ — одноэлементное множество, единственным элементом которого является пустое слово;

(в) регулярному выражению $(ABC\ldots E)$ соответствует множество всех слов, которые можно получить, если к слову из A приписать слово из B, затем из C,..., затем из E (*конкатенация* множеств);

(г) регулярному выражению $(A|B|C|\ldots|E)$ соответствует объединение множеств, соответствующих выражениям $A, B, C, \ldots, E$;

(д) регулярному выражению A* соответствует *итерация* множества, соответствующего выражению A, то есть множество всех слов, которые можно так разрезать на куски, что каждый кусок принадлежит множеству, соответствующему выражению A. (В частности, пустое слово всегда содержится в A*.)

Множества, соответствующие регулярным выражениям, называются *регулярными*. Вот несколько примеров:

Выражение	Множество
(a|b)*	все слова из букв a и b
(aa)*	слова из чётного числа букв a
(Λ|a|b|aa|ab|ba|bb)	все слова длины не более 2 из букв a,b

10.7.5. Написать регулярное выражение, которому соответствует множество всех слов из букв a и b, в которых число букв a чётно.

Решение. Выражение b* задает все слова без буквы a, а выражение (b* a b* a b*) — все слова ровно с двумя буквами a. Остается объединить эти множества, а потом применить итерацию:

((b* a b* a b*) | b*)*

Другой вариант ответа:

(b* a b* a)* b*

10.7.6. Написать регулярное выражение, которое задаёт множество всех слов из букв a, b, c, в которых слово bac является подсловом.

Решение. ((a|b|c)*bac(a|b|c)*)

Теперь задачу о поиске образца в слове можно переформулировать так: проверить, принадлежит ли слово множеству, соответствующему данному регулярному выражению.

10.7.7. Какие выражения соответствуют образцам a?b и ab*cd, рассмотренным ранее? (В образце символ * используется не в том смысле, что в регулярных выражениях!) Предполагается, что алфавит содержит буквы a, b, c, d, e.

Решение.

```
((a|b|c|d|e)*a(a|b|c|d|e)b(a|b|c|d|e)*)

((a|b|c|d|e)*ab(a|b|c|d|e)*cd(a|b|c|d|e)*)
```

10.7.8. Доказать, что для всякого регулярного выражения можно построить конечный автомат, который распознает соответствующее этому выражению множество слов.

Решение. Нам потребуется новое понятие — понятие *источника*, или *недетерминированного конечного автомата*. Представим себе ориентированный граф — картинку из нескольких точек (вершин) и некоторых стрелок, соединяющих эти точки (рёбер). Пусть на некоторых рёбрах написаны буквы (не обязательно на всех). Пусть также среди вершин выбраны две — начальная Н и конечная К. Такая картинка называется *источником*.

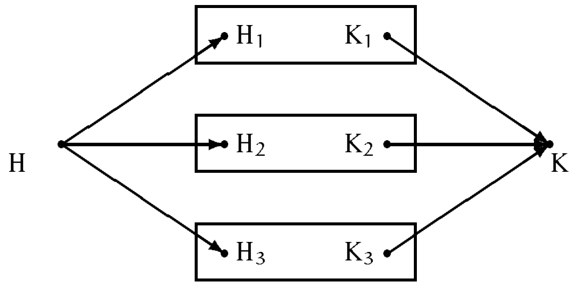
Будем двигаться различными способами из Н в К, читая буквы по дороге (на тех стрелках, где они есть). Каждому пути из Н в К, таким образом, соответствует некоторое слово. А источнику в целом соответствует множество слов — тех слов, которые можно прочесть на путях из Н в К.

Замечание. Если нарисовать состояния конечного автомата в виде точек, а переходы при чтении букв изобразить в виде стрелок, то станет ясно, что конечный автомат — это частный случай источника. (С дополнительными требованиями: (а) на всех стрелках, за исключением ведущих в К, есть буквы; (б) для любой точки на выходящих из неё стрелках каждая буква встречается ровно один раз.)

Мы будем строить конечный автомат по регулярному выражению в два приёма. Сначала мы построим источник, которому соответствует то же самое множество слов. Затем для произвольного источника построим автомат, который проверяет, принадлежит ли слово соответствующему множеству.

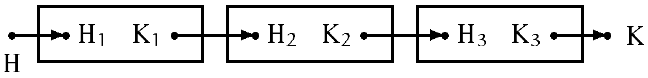
10.7.9. По регулярному выражению построить источник, задающий то же множество.

Решение. Индукция по построению регулярного выражения. Буквам соответствуют графы из одной стрелки. Объединение реализуется так:

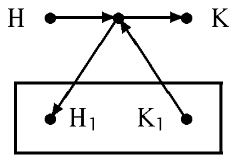


Нарисована картинка для объединения трёх множеств, прямоугольники — это источники, им соответствующие; указаны начальные и конечные вершины. На новых стрелках (их 6) букв не написано.

Конкатенации соответствует картинка



Наконец, итерации соответствует картинка



10.7.10. Дан источник. Построить конечный автомат, проверяющий, принадлежит ли входное слово соответствующему множеству (то есть можно ли прочесть это слово, идя из Н в К).

Решение. Состояниями автомата будут множества вершин источника. Именно, прочтя некоторое начало X входного слова, мы будем помнить множество всех вершин источника, в которые можно пройти из начальной, прочитав на пути слово X.

Оказывается, что регулярные выражения, автоматы и источники распознают одни и те же множества. Чтобы убедиться в этом, нам осталось решить такую задачу:

10.7.11. Дан источник. Построить регулярное выражение, задающее то же множество, что и этот источник.

Решение. (Сообщено участниками просеминара по логике.) Пусть источник имеет вершины $1, \ldots, k$. Будем считать, что 1 — это начало, а k — конец. Через $D_{i,j,s}$ обозначим множество всех слов, которые можно прочесть на пути из i в j, если в качестве промежуточных пунктов разрешается использовать только вершины $1, \ldots, s$. Согласно определению, источнику соответствует множество $D_{1,k,k}$.

Индукцией по s будем доказывать регулярность всех множеств $D_{i,j,s}$ при всех i и j. При $s = 0$ это очевидно (промежуточные вершины запрещены, поэтому каждое из множеств состоит только из букв).

Из чего состоит множество $D_{i,j,s+1}$? Отметим на пути моменты, в которых он заходит в $(s+1)$-ую вершину. При этом путь разбивается на части, каждая из которых уже не заходит в неё. Поэтому легко сообразить, что

$$D_{i,j,s+1} = D_{i,j,s} \,|\, (D_{i,s+1,s} \quad D_{s+1,s+1,s}* \quad D_{s+1,j,s})$$

(вольность записи: мы используем для операций над множествами обозначения как для регулярных выражений). Остаётся воспользоваться предположением индукции.

10.7.12. Где ещё используется то же самое рассуждение?

Ответ. В алгоритме Флойда вычисления цены кратчайшего пути, см. главу 9.

10.7.13. Доказать, что класс множеств, задаваемых регулярными выражениями, не изменился бы, если бы мы разрешили использовать не только объединение, но и отрицание (а следовательно, и пересечение — оно выражается через объединение и отрицание).

Решение. Для автоматов переход к отрицанию очевиден.

Замечание. На практике важную роль играет число состояний автомата. Оказывается, что тут всё не так просто, и переход от источника к автомату требует экспоненциального роста числа состояний. Подробное рассмотрение связанных с этим теоретических и практических вопросов — дело особое (см. книгу Ахо, Ульмана и Сети о компиляторах).

11. Представление множеств. Хеширование

11.1. Хеширование с открытой адресацией

В главе 6 было указано несколько представлений для множеств, элементами которых являются целые числа произвольной величины. Однако в любом из них хотя бы одна из операций проверки принадлежности, добавления и удаления элемента требовала количества действий, пропорционального числу элементов множества. На практике это бывает слишком много. Существуют способы, позволяющие получить для всех трёх упомянутых операций оценку $C \log n$. Один из таких способов мы рассмотрим в следующей главе. В этой главе мы разберем способ, которые хотя и приводит к Cn действиям в худшем случае, но зато «в среднем» требует значительно меньшего их числа. (Мы не будем уточнять слов «в среднем», хотя это и можно сделать.) Этот способ называется *хешированием*.

Пусть нам необходимо представлять множества элементов типа `T`, причем число элементов в них заведомо меньше `n`. Выберем некоторую функцию `h`, определенную на значениях типа `T` и принимающую значения `0...n-1`. Было бы хорошо, чтобы эта функция принимала на элементах будущего множества по возможности более разнообразные значения. (Худший случай — это когда её значения на всех элементах хранимого множества одинаковы.) Эту функцию будем называть *хеш-функцией*, или, как ещё говорят, *функцией расстановки*.

Введём два массива

```
val:  array [0..n-1] of T;
used: array [0..n-1] of boolean;
```

(мы позволяем себе писать `n-1` в качестве границы в определении типа, хотя в паскале это не разрешается). В этих массивах будут храниться элементы множества: оно равно множеству всех `val[i]` для тех `i`, для которых `used[i]`, причём все эти `val[i]` различны. По возможности мы будем хранить элемент `t` на месте `h(t)`, считая это место «исконным» для элемента `t`. Однако может случиться так, что новый элемент, который мы хотим добавить, претендует на уже занятое место (для которого `used` истинно). В этом случае мы отыщем ближайшее справа свободное место и запишем элемент туда. («Справа» значит «в сторону увеличения индексов»;

дойдя до края, мы перескакиваем в начало.) По предположению, число элементов всегда меньше n, так что пустые места заведомо будут.

Формально говоря, в любой момент должно соблюдаться такое требование: для любого элемента множества участок справа от его исконного места до его фактического места полностью заполнен.

Благодаря этому проверка принадлежности заданного элемента t осуществляется легко: встав на h(t), двигаемся направо, пока не дойдём до пустого места или до элемента t. В первом случае элемент t отсутствует в множестве, во втором — присутствует. Если элемент отсутствует, то его можно добавить на найденное пустое место. Если присутствует, то можно его удалить (положив used = false).

11.1.1. В предыдущем абзаце есть ошибка. Найдите её и исправьте.

Решение. Дело в том, что при удалении требуемое свойство «отсутствия пустот» может нарушиться. Поэтому будем делать так. Создав дыру, будем двигаться направо, пока не натолкнёмся на элемент, стоящий не на исконном месте, или на ещё одно пустое место. Во втором случае на этом можно успокоиться. В первом случае посмотрим, не нужно ли найденный элемент поставить на место дыры. Если нет, то продолжаем поиск, если да, то затыкаем им старую дыру. При этом образуется новая дыра, с которой делаем всё то же самое.

11.1.2. Написать программы проверки принадлежности, добавления и удаления.

Решение.

```
function принадлежит (t: T): boolean;
| var i: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| принадлежит := used [i] and (val [i] = t);
end;
```

```
procedure добавить (t: T);
| var i: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| if not used [i] then begin
| | used [i] := true;
| | val [i] := t;
| end;
end;

procedure исключить (t: T);
| var i, gap: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| if used [i] and (val [i] = t) then begin
| | used [i] := false;
| | gap := i;
| | i := (i + 1) mod n;
| | {gap - дыра, которая может закрыться одним из i,i+1,...}
| | while used [i] do begin
| | | if i = h (val[i]) then begin
| | | | {на своем месте, ничего не делать}
| | | end else if dist(h(val[i]),i) < dist(gap,i) then begin
| | | | {gap...h(val[i])...i, ничего не делать}
| | | end else begin
| | | | used [gap] := true;
| | | | val [gap] := val [i];
| | | | used [i] := false;
| | | | gap := i;
```

```
      end;
      i := (i + 1) mod n;
    end;
  end;
end;
```

Здесь dist(a,b) — измеренное по часовой стрелке (слева направо) расстояние от a до b, то есть

$$\texttt{dist(a,b)} = (\texttt{b}-\texttt{a}+\texttt{n}) \bmod \texttt{n}.$$

(Мы прибавили n, так как функция mod правильно работает при положительном делимом.)

11.1.3. Существует много вариантов хеширования. Один из них таков: обнаружив, что исконное место (обозначим его i) занято, будем искать свободное не среди $i+1, i+2, \ldots$, а среди $r(i), r(r(i)), r(r(r(i))), \ldots$, где r — некоторое отображение $\{0, \ldots, n-1\}$ в себя. Какие при этом будут трудности?

Ответ. (1) Не гарантируется, что если пустые места есть, то мы их найдем. (2) При удалении неясно, как заполнять дыры. (На практике во многих случаях удаление не нужно, так что такой способ также применяется. Считается, что удачный подбор функции r может предотвратить образование «скоплений» занятых ячеек.)

11.1.4. Пусть для хранения множества всех правильных русских слов в программе проверки орфографии используется хеширование. Что нужно добавить, чтобы к тому же уметь находить английский перевод любого правильного слова?

Решение. Помимо массива val, элементы которого являются русскими словами, нужен параллельный массив их английских переводов.

11.2. Хеширование со списками

На хеш-функцию с k значениями можно смотреть как на способ свести вопрос о хранении одного большого множества к вопросу о хранении нескольких меньших. Именно, если у нас есть хеш-функция с k значениями, то любое множество разбивается на k подмножеств (возможно, пустых), соответствующих возможным значениям хэш-функции. Вопрос о проверке принадлежности, добавлении или удалении для большого множества сводится к такому же вопросу для одного из меньших (чтобы узнать, для какого, надо посмотреть на значение хеш-функции).

Эти меньшие множества удобно хранить с помощью ссылок; их суммарный размер равен числу элементов хешируемого множества. Следующая задача предлагает реализовать этот план.

11.2.1. Пусть хеш-функция принимает значения `1...k`. Для каждого значения хеш-функции рассмотрим список всех элементов множества с данным значением хеш-функции. Будем хранить эти `k` списков с помощью переменных

```
Содержание: array [1..n] of T;
Следующий: array [1..n] of 1..n;
ПервСвоб: 1..n;
Вершина: array [1..k] of 1..n;
```

так же, как мы это делали для `k` стеков ограниченной суммарной длины. Напишите соответствующие программы. (Удаление по сравнению с открытой адресацией упрощается.)

Решение. Перед началом работы надо положить `Вершина[i]` = `0` для всех `i` = `1...k`, и связать все места в список свободного пространства, положив `ПервСвоб` = `1` и `Следующий[i]` = `i+1` для `i` = `1...n-1`, а также `Следующий[n]` = `0`.

```
function принадлежит (t: T): boolean;
| var i: integer;
begin
| i := Вершина[h(t)];
| {осталось искать в списке, начиная с i}
| while (i <> 0) and (Содержание[i] <> t) do begin
| | i := Следующий[i];
| end; {(i=0) or (Содержание [i] = t)}
| принадлежит := (i<>0) and (Содержание[i]=t);
end;
```

```
procedure добавить (t: T);
| var i: integer;
begin
| if not принадлежит(t) then begin
| | i := ПервСвоб;
| | {ПервСвоб <> 0 - считаем, что не переполняется}
| | ПервСвоб := Следующий[ПервСвоб]
| | Содержание[i]:=t;
| | Следующий[i]:=Вершина[h(t)];
| | Вершина[h(t)]:=i;
| end;
end;

procedure исключить (t: T);
| var i, pred: integer;
begin
| i := Вершина[h(t)]; pred := 0;
| {осталось искать в списке, начиная с i;  pred -
|    предыдущий, если он есть, и 0, если нет}
| while (i <> 0) and (Содержание[i] <> t) do begin
| | pred := i; i := Следующий[i];
| end; {(i=0) or (Содержание [i] = t)}
| if i <> 0 then begin
| | {Содержание[i]=t, элемент есть, надо удалить}
| | if pred = 0 then begin
| | | {элемент оказался первым в списке}
| | | Вершина[h(t)] := Следующий[i];
| | end else begin
| | | Следующий[pred] := Следующий[i]
| | end;
| | {осталось вернуть i в список свободных}
| | Следующий[i] := ПервСвоб;
| | ПервСвоб:=i;
| end;
```

```
end;
```

11.2.2. (Для знакомых с теорией вероятностей.) Пусть хеш-функция с k значениями используется для хранения множества, в котором в данный момент n элементов. Доказать, что математическое ожидание числа действий в предыдущей задаче не превосходит $C(1+n/k)$, если добавляемый (удаляемый, искомый) элемент t выбран случайно, причём все значения $h(t)$ имеют равные вероятности (равные $1/k$).

Решение. Если $l(i)$ — длина списка, соответствующего хеш-значению i, то число операций не превосходит $C(1 + l(h(t)))$; усредняя, получаем искомый ответ, так как $\sum_i l(i) = n$.

Эта оценка основана на предположении о равных вероятностях. Однако в конкретной ситуации всё может быть совсем не так, и значения хеш-функции могут «скучиваться»: для каждой конкретной хеш-функции есть «неудачные» ситуации, когда число действий оказывается большим. Приём, называемый *универсальным хешированием*, позволяет обойти эту проблему. Идея состоит в том, что берётся семейство хеш-функций, причем любая ситуация оказывается неудачной лишь для небольшой части этого семейства.

Пусть H — семейство функций, каждая из которых отображает множество T в множество из k элементов (например, $0 \ldots k-1$). Говорят, что H — *универсальное семейство хеш-функций*, если для любых двух различных значений s и t из множества T вероятность события $h(s) = h(t)$ для случайной функции h из семейства H равна $1/k$. (Другими словами, те функции из H, для которых $h(s) = h(t)$, составляют $1/k$-ую часть всех функций в H.)

Замечание. Более сильное требование к семейству H могло бы состоять в том, чтобы для любых двух различных элементов s и t множества T значения $h(s)$ и $h(t)$ случайной функции h являются независимыми случайными величинами, равномерно распределёнными на $0 \ldots k-1$.

11.2.3. Пусть $t_1, \ldots, t_n$ — произвольная последовательность различных элементов множества T. Рассмотрим количество действий, происходящих при помещении элементов $t_1, \ldots, t_n$ в множество, хешируемое с помощью функции h из универсального семейства H. Доказать, что среднее количество действий (усреднение — по всем h из H) не превосходит $Cn(1 + n/k)$.

Решение. Обозначим через m_i количество элементов последовательности, для которых хеш-функция равна i. (Числа $m_0, \ldots, m_{k-1}$ зависят, конечно, от выбора хеш-функции.) Количество действий, которое мы хотим оценить, с точностью до постоянного множителя равно $m_0^2 + m_1^2 + \ldots + m_{k-1}^2$. (Если s чисел попадают в одну хеш-ячейку, то для этого требуется примерно $1+2+\ldots+s$ действий.) Эту же сумму квадратов можно записать как число пар $\langle p, q\rangle$, для которых $h(t_p) = h(t_q)$. Последнее равенство, если его рассматривать как событие при фиксированных p и q, имеет вероятность $1/k$ при $p \neq q$, поэтому среднее значение соответствующего члена суммы равно $1/k$, а для всей суммы получаем оценку порядка n^2/k, а точнее $n + n^2/k$, если учесть члены с $p = q$.

Эта задача показывает, что на каждый добавляемый элемент приходится в среднем $C(1 + n/k)$ операций. В этой оценке дробь n/k имеет смысл «коэффициента заполнения» хеш-таблицы.

11.2.4. Доказать аналогичное утверждение для произвольной последовательности операций добавления, поиска и удаления (а не только для добавления, как в предыдущей задаче).

[Указание. Будем представлять себе, что в ходе поиска, добавления и удаления элемент проталкивается по списку своих коллег с тем же хеш-значением, пока не найдёт своего двойника или не дойдёт до конца списка. Будем называть i-j-столкновением столкновение t_i с t_j. (Оно либо произойдёт, либо нет — в зависимости от h.) Общее число действий примерно равно числу всех происшедших столкновений плюс число элементов. При $t_i \neq t_j$ вероятность i-j-столкновения не превосходит $1/k$. Осталось проследить за столкновениями между равными элементами. Фиксируем некоторое значение x из множества T и посмотрим на связанные с ним операции. Они идут по циклу: добавление — проверки — удаление — добавление — проверки — удаление — ... Столкновения происходят между добавляемым элементом и следующими за ним проверками (до удаления включительно), поэтому общее их число не превосходит числа элементов, равных x.]

Теперь приведем примеры универсальных семейств. Очевидно, для лю-