Уральский федеральный университет
имени Первого президента России Б.Н.Ельцина

СОЛОДУШКИН
Святослав Игоревич

# ФУНДАМЕНТАЛЬНЫЕ ЗАДАЧИ ИНФОРМАТИКИ. СКРИПТЫ

Для студентов высших учебных заведений, обучающихся по направлениям 230700.62 — «Прикладная информатика», 010300.62 — «Фундаментальная информатика и информационные технологии», 010200.62 — «Математика и компьютерные науки», 090301.65 — «Компьютерная безопасность».

УДК 004.02

Рецензенты

Кафедра прикладной математики и технической графики, ФГБОУ ВПО Уральская государственная архитектурно-художественная академия.

Ложников Андрей Борисович, научный сотрудник Института математики и механики УрО РАН, кандидат физ.-мат. наук.

Солодушкин С.И. **Фундаментальные задачи информатики. Скрипты.** — Издательство Уральского федерального университета, 57 с., — ISBN 1-23-345678-9.

Рассмотрены некоторые классические задачи фундаментальной информатики, для каждой задачи излагаются формальные и содержательные постановки и методы решения этих задач. Особое внимание уделено разбору специально подобранных примеров, которые всесторонне иллюстрируют вводимые понятия, работу алгоритмов и применения стандартов на практике.

Для студентов высших учебных заведений, специализирующихся в области прикладной и фундаментальной информатики, разработке программного обеспечения

# Предисловие

Базовые университетские курсы, посвященные фундаментальным задачам информатики, имеют своей целью формирование программистского мировоззрения и общего кругозора в области информационных технологий. К числу таких курсов относится и читаемый автором на математико-механическом факультете УрФУ курс «Введение в специальность. Скрипты». Цель курса — знакомство с фундаментальными задачами информатики, как то сжатие данных или поиск подстроки в строке. В качестве языка для реализации изучаемых алгоритмов выбран легкий в освоении и одновременно с тем выразительный язык скриптового типа JScript. В курсе рассматриваются задачи, над решением которых несколько десятилетий назад работали выдающиеся инженеры, математики и программисты — Шеннон, Дейкстра, Хэмминг, Хаффман и другие классики Computer Science. Предложенные ими подходы оказались исключительно удачными и позволяют решать многие классы возникающих на практике задач; на сегодняшний день они общепризнаны, а потому являются обязательными для изучения студентами, специализирующимися в области математики, программирования, фундаментальной и прикладной информатики.

Настоящее учебное пособие призвано помочь студентам в освоении курса «Введение в специальность. Скрипты» и отражает структуру курса. Пособие разбито на главы. Каждая глава соответствует рассматриваемой на занятиях теме и содержит описание и формальную постановку задачи, необходимые теоретические сведения, указания к решению и иногда листинги программ. Кроме того, в конце глав приводятся вопросы для самоконтроля.

При подготовке учебного пособия автор в основном обращался к первоисточникам, т. е. к оригинальным научным статьям и документации.

# 1. Алгоритм сжатия RLE

## 1.1. Необходимость сжатия данных «на лету»

Хранимые файлы и передаваемые по сети данные часто содержат длинные цепочки повторяющихся байтов. В целях уменьшения размеров файлов и снижения загрузки сети применяют специальные алгоритмы сжатия. Рассмотрим пример.

Клиентская машина отправила по сети пакет запроса на сервер точного времени и ждет пакет ответа. Пакеты содержат заголовки и поля данных; некоторые поля данных заполнены нулями как значениями по умолчанию. В результате реальный пакет может содержать весьма длинные последовательности нулей. Ниже приведен шестнадцатеричный дамп SNTP [1] пакета.

```
D:\network\ntp>perl ntime.pl -d pool.ntp.org

0000  0B 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0020  00 00 00 00 00 00 00 00  D5 1C E8 21 DF 40 00 00
```

Дамп пакета — это снимок, образ, иными словами, содержимое пакета. Поскольку пакет состоит из последовательности байтов, удобно представлять содержимое пакета в шестнадцатеричном виде, т. е. по два символа на байт. Самое меньшее число, которое может быть в байте, — это 0, самое большое — 255; этим значениям соответствуют 00 и FF. Байты нумеруются с нуля, и в начале каждой строки указан номер (в шестнадцатеричном виде) первого в этой строке байта. Для удобства восприятия байты записываются по шестнадцать штук в строку, а в середине делается дополнительный пробел.

Здесь 39 расположенных подряд байта имеют значение 0, это байты с номерами от 1 до 39 включительно (их шестнадцатеричные номера $1_{16}$ и $27_{16}$).

Естественно желание представить пакет в более компактном виде. Если таких пакетов будет передаваться много, то экономия будет весьма ощутима. Существует немало алгоритмов сжатия файлов, хранящихся на дисках или иных долговременных носителях; для лучшего сжатия эти алгоритмы требуют построения специальных таблиц и нескольких проходов по файлу, и в итоге работают небыстро. Работа с сетевыми пакетами имеет некоторые особенности.

1. Байты поступают к контроллеру сетевой карты непрерывным потоком.
2. Поступившие к контроллеру сетевой карты байты буферизуются на непродолжительное время и отправляются в сеть.

Таким образом, нельзя, просмотрев конец отправляемых пакетов, вернуться в начало и что-то переделать, так как это начало может быть уже отослано. Кроме того, алгоритм должен сжимать поступающий поток байтов очень быстро (со скоростью поступления этих пакетов на контроллер сетевой карты), иначе работа по сети лишь замедлится.

[1] SNTP (англ. Simple Network Time Protocol) — протокол синхронизации времени по компьютерной сети.

Выражаясь образно, можно сказать, что пакеты «летят в сеть». Алгоритмы, предназначенные для сжатия не «долгоживущих» на диске файлов, а таких «летящих» последовательностей байтов, называются алгоритмами сжатия на лету.

## 1.2. Описание алгоритма RLE

Алгоритм RLE (англ. Run-length encoding) — простой алгоритм сжатия данных, который оперирует сериями данных, т. е. последовательностями, в которых один и тот же символ встречается несколько раз подряд. При кодировании строка одинаковых символов, составляющих серию, заменяется строкой, которая содержит сам повторяющийся символ и количество его повторов.

Сначала изложим основные идея на двух простых примерах, а потом строго сформулируем правила кодирования и декодирования.

Пусть есть строка

$$\texttt{Priv}\underbrace{\texttt{ee}\ldots\texttt{e}}_{65\text{раз}}\texttt{t, I wait you here t}\underbrace{\texttt{oo}\ldots\texttt{o}}_{69\text{раз}}\ \texttt{l}\underbrace{\texttt{oo}\ldots\texttt{o}}_{38\text{раз}}\texttt{ng}$$

Если некоторый символ встречается много (пока не уточняем сколько) раз подряд, образуя цепочку, заменим цепочку на специальную трехбайтовую последовательность <Escape_symbol, Counter, Symbol>:

1. Escape_symbol — экранирующий символ (иначе — эспкейп-символ), показывающий, что следующие за ним два байта надо интерпретировать специальным образом. Без ограничения общности в качестве Escape_symbol можно выбрать символ #.

2. Counter — число повторов (длина цепочки). Отводя под число повторов один байт, мы автоматически разрешаем цепочки длиной не более 255 символов. Число повторов удобно представить как символ с ASCII-кодом, равным числу повторов.

3. Symbol — сам повторяющийся символ.

Вспомним, что в таблице ASCII-кодам 65, 69 и 38 соответствуют английские буквы A и E и амперсанд &. Таким образом, рассматриваемую строку можно представить в виде

$$\texttt{Priv}\underline{\#\texttt{Aet}}\texttt{, I wait you here t}\underline{\#\texttt{Eo}}\,\texttt{l}\underline{\#\&\texttt{ong}}$$

Трехбайтовые последовательности <Escape_symbol, Counter, Symbol> мы выделили подчеркиванием.

Сразу отметим, что среди символов ASCII есть непечатные, такие как символы с кодами от 0 до 31, означающие конец файла, возврат каретки, вертикальную табуляцию и т. п. Чтобы изобразить их на страницах нашего пособия, мы применяем условную запись — код в рамке.

Приведем второй пример. Строку

$$\texttt{1000000000 + 2000000000 = 3000000000}$$

можно представить в сжатом виде[1]

$$1\#\boxed{9}0 + 2\#\boxed{9}0 = 3\#\boxed{9}0$$

Таким образом, вместо 36 символов мы использовали лишь 18, сжали строку в два раза.

Как происходит декодирование? Просматривая сжатую строку, декодировщик ищет спецсимвол #. Встретив #, декодировщик понимает, что следующие два символа надо обработать по специальной схеме: повторить нужное число раз соответствующий символ. Например, встретив #$\boxed{9}$0, декодировщик развернет ее в строку 00000000.

Теперь сформулируем правила кодирования.

1. Цепочки длиной менее трех символов не выгодно кодировать, так как в результате получается три байта. Таким образом кодируем цепочки, длина которых от четырех байт включительно.

2. Если в сжимаемой строке имеются цепочки длиной более 255 символов, то их длину нельзя представить в виде одного байта. Выход такой: от длинной цепочки отрезать куски по 255 байт пока это возможно и кодировать эти куски обычным способом, т. е. писать #$\boxed{255}$ и повторяемый символ. В результате от длинной цепочки останется хвост, длина которого не более 255, его можно закодировать обычным способом.

3. Если в сжимаемой строке имеется символ, совпадающий со спецсимволом #, то просто так его переписать в сжатую строку нельзя, иначе это вызовет ошибку при декодировании. Спецсимволы надо кодировать всегда.

Проиллюстрируем все эти правила на примере

$$\texttt{a}\underbrace{\texttt{bb}\ldots\texttt{b}}_{823\text{раз}}\texttt{\#cc}\underbrace{\texttt{ee}\ldots\texttt{e}}_{513\text{раз}}\texttt{f\#\#\#\#}$$

Заметим, что 823 = 3*255 + 55, 513 = 2*255 + 3, ASCII-коду 55 соответствует цифра 7. В результате получим

$$\texttt{a\#}\boxed{255}\texttt{b\#}\boxed{255}\texttt{b\#}\boxed{255}\texttt{b\#7b\#}\boxed{1}\texttt{\#cc\#}\boxed{255}\texttt{e\#}\boxed{255}\texttt{eeeef\#}\boxed{4}\texttt{\#}$$

Еще раз обратим внимание на то, что запись $\boxed{4}$ означает символ с ASCII-кодом 4; этот символ означает конец передачи (a в UNIX-системах интерпретируется как конец вводимых данных) и относится к числу непечатных символов.

Приведем еще пример. Строка $\underbrace{\#\#\ldots\#}_{36\text{раз}}$ будет закодирована как ###, потому что ASCII-коду 36 соответствует символ #. В итоге первый символ # — это спецсимвол, второй # — 36 повторов, третий # — символ, который надо повторить 36 раз.

Изложенный алгоритм сжатия допускает одну оптимизацию. Заметим, что цепочки, состоящие не из #, кодируются лишь в том случае, если их длина не менее 4-х символов; следовательно, символы с кодами 0, 1, 2, 3 не используются в качестве Counter. Можно сделать своего рода сдвиг на 4 и кодировать цепочку aaaa не как #$\boxed{4}$a, но как #$\boxed{0}$a.

[1]Напомним, что ASCII символ с кодом 9 — это табуляция.

Аналогично для цепочки aaaaa вместо #5a получим #1a и т. д. Таким образом удастся цепочку длиной 259 символов упаковать в три байта: #255a; без оптимизации та же цепока из 259 символов потребовала бы 6 байт: #255a#4a.

Эффективность алгоритма сжатия оценивают по коэффициенту сжатия. Коэффициентом сжатия назовем отношение длины исходного файла к длине сжатого файла.

## 1.3. Постановка задачи

В текстовом фале `input.txt` записана строка. Необходимо закодировать ее, используя алгоритм RLE, закодированную строку сохранить в файле `code.txt`. После этого раскодировать сроку из файла `code.txt`, результат сохранить в `decode.txt`. Очевидно, что строки в файлах `input.txt` и `decode.txt` должны совпадать.

Режим работы программы (т. е. кодирование или декодирование), имя файла с входной строкой и имя файла для записи результата задаются как аргументы командной строки. Например:

```
C:/>CScript rle.js code input.txt code.txt
C:/>CScript rle.js decode code.txt decode.txt
```

Вычислить коэффициент сжатия.

## 1.4. Указания к решению

Для доступа к параметрам командной строки используется коллекция `WSH.Arguments`, которая содержит параметры командной строки для исполняемого сценария. Пусть скрипт `first.js` вызывался с параметрами 48 и 56

```
C:/>CSscript first.js 48 56:
```

Выведем на экран эти параметры:

```
WSH.echo(WSH.Arguments(0),' ',WSH.Arguments(1));
```

Объект `FileSystemObject` обеспечивает доступ к файловой системе Windows. Его конструктор имеет вид: `new ActiveXObject('Scripting.FileSystemObject')`. Сценарий может создать только один экземпляр данного объекта, сколько бы раз в нем не вызывался данный конструктор.

Для чтения из текстового файла или записи в текстовый файл необходимо этот файл сначала открыть (на чтение или запись соответственно). Приведем два образца кода.

Откроем файл на запись и запишем строку:

```
fso = new ActiveXObject("Scripting.FileSystemObject");
fh = fso.OpenTextFile("c:\\file.txt", 2, true);
fh.WriteLine("Тестовая строка.");
fh.Close();
```

Откроем файл на чтение и прочитаем все его содержимое и выведем на экран:

```
fso = new ActiveXObject("Scripting.FileSystemObject");
fh = fso.OpenTextFile("c:\\test.txt");
s = fh.ReadAll();
fh.Close();
WSH.echo(s);
```

## 1.5. Задания для самостоятельной работы

1. Прочитайте, как алгоритм RLE успешно применяется для сжатия картинок BMP.

2. В строке `abbcabbcabbcabbcabbcabbc` один и тот же фрагмент `abbc` повторяется шесть раз. Можно ли алгоритмом RLE успешно сжать такую строку? Предложите свой вариант сжатия для подобных строк.

# 2. Энтропия Шеннона

## 2.1. История вопроса

Фундаментальным понятием компьютерных наук является «информация». Наивные определения типа «информация — это сведения (сообщения, данные) независимо от формы их представления» не могут быть использованы для развития информатики как науки. Необходимость в формальном, строгом определении продиктована запросами инженеров, связистов, криптографов и других практикующих специалистов. Yабор основополагающих определений и аксиом позволит построить теорию, хорошо согласующуюся с практикой. Эта теория послужит базой для математического моделирования, а численное исследование моделей гораздо дешевле, чем проведение натурных экспериментов (действительно, аренда магистрального канала у Ростелекома для исследования эффективности нового алгоритма кодирования сигналов может стоить очень дорого). Первые попытки дать математическое определение информации были предприняты Хартли[1]. На сегодняшний день общепринятой является формализация, предложенная Шенноном[2].

Чтобы подчеркнуть прикладную значимость работ Шеннона, отметим, что задача, стоявшая перед ним, заключалась в оптимальном кодировании сообщений, передаваемых через каналы с помехами. Шеннону удалось показать, что при заданном соотношении между мощностью полезного сигнала и мощностью помехи можно передавать по каналу связи сообщения со сколь угодно малой вероятностью ошибок. Развитая Шенноном теория информации помогла решить главные проблемы, связанные с передачей сообщений, а именно: устранить избыточность передаваемых сообщений, произвести кодирование и передачу сообщений по каналам связи с шумами. Решение проблемы избыточности подлежащего передаче сообщения позволяет максимально эффективно использовать канал связи. К примеру, повсеместно используемые в настоящее время методы снижения избыточности в системах телевизионного вещания позволяют передавать до шести цифровых программ коммерческого телевидения в полосе частот, которую занимает обычный сигнал аналогового телевидения.

## 2.2. Формальное определение информации и энтропии по Шеннону

Для дальнейшего изложения нам потребуется понятие дискретной случайной величины. Будем говорить, что величина является случайной, если в результате проведенного испытания она может принять то или иное заранее неизвестное значение. Различные значения появляются с различными вероятностями. Чтобы описать дискретную случайную величину, которая может принимать одно из $n$ различных значений, составим таблицу:

[1]Ральф Винтон Лайон Хартли (англ. Ralph Vinton Lyon Hartley, 1888—1970) — американский ученый-электронщик. В статье Transmission of Information (Bell System Technical Journal. 1928. P. 535–563) ввел понятие «логарифмическая мера информации» $H = Kln(M)$.

[2]Клод Элвуд Шеннон (англ. Claude Elwood Shannon, 1916—2001) — американский инженер и математик. В статье A Mathematical Theory of Communication (Bell System Technical Journal/.1948. P. 379–423, 623–656) ввел понятие «информационная энтропия».

| **X** | $x_1$ | $x_2$ | ... | $x_n$ |
|---|---|---|---|---|
| **P** | $p_1$ | $p_2$ | ... | $p_n$ |

Здесь $x_i,\ i = \overline{1,n}$, — значения случайной величины, а $p_i,\ i = \overline{1,n}$, — соответствующие вероятности. Ясно, что $\sum_{i=1}^{n} p_i = 1$.

Формализуем процесс передачи сообщений по сети в терминах случайных величин, параллельно иллюстрируя приводимые построения на простом примере. После этого легко будет дать определение в общем случае.

Есть источник, который через определенные моменты времени посылает в канал сообщение, например, каждые 10 минут посылает число 0, 1, 2 или 3. Заранее нельзя достоверно сказать, что он пошлет в следующий момент, однако из статистических наблюдений известно частотное распределение возможных вариантов. Например, «0» появляется в три раза чаще, чем «3»; «1» появляется в четыре раза чаще, чем «3»; «2» появляется в раза два чаще, чем «3». Таким образом, можно составить таблицу частот. В данном случае она выглядит так (сумма частот равна единице):

| **Сигнал источника** | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| **Частота появления** | 0.3 | 0.4 | 0.2 | 0.1 |

табл. частот источника

Канал не защищен от помех, поэтому принятое сообщение может не совпадать с отправленным. Возникает вопрос: можно ли по полученному сообщению судить о том, что было отправлено? Если «да», то насколько наши суждения достоверны; много ли в принятом сообщении информации о том, что посылал источник? Ясно пока лишь то, что при прочих равных условиях с ростом интенсивности помех количество информации, содержащейся в полученном сообщении о посланном уменьшается.

Источник можно рассматривать как случайную величину **X**, получатель — **Y**; отправленное число $x$ — это реализация случайной величины **X**, полученное число $y$ — это реализация случайной величины **Y**. Если помех в канале нет то, очевидно, $x = y$, и имеет место *закон совместного распределения:*

| **X** \ **Y** | **0** | **1** | **2** | **3** |
|---|---|---|---|---|
| **0** | 0.3 | 0 | 0 | 0 |
| **1** | 0 | 0.4 | 0 | 0 |
| **2** | 0 | 0 | 0.2 | 0 |
| **3** | 0 | 0 | 0 | 0.1 |

табл. для канала без помех

Числа, стоящие в данной таблице, надо понимать так: из ста посылаемых сообщений в среднем тридцать содержат «0», получатель принимает их без искажения, т. е. все они содержат «0»; сорок посылаемых сообщений содержат «1», получатель принимает их без искажения, т. е. все они содержат «1», и т. д. Ясно, что ненулевые элементы стоят только на главной диагонали.

Ситуация меняется, если интенсивность помехи не равна нулю. Здесь уже возможно, что источник пошлет «0», а получатель примет «1». Пусть наблюдения показали, что таблица совместного распределения имеет вид:

| **X** \ **Y** | **0** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|---|
| **0** | 0.23 | 0.06 | 0.01 | 0 | 0 |
| **1** | 0.04 | 0.31 | 0.05 | 0 | 0 |
| **2** | 0 | 0.06 | 0.12 | 0.02 | 0 |
| **3** | 0 | 0 | 0.01 | 0.07 | 0.02 |

табл. для канала с помехами

Числа, стоящие в данной таблице, надо понимать так: из ста посылаемых сообщений в среднем тридцать содержат «0», из них лишь двадцать три дойдут до получателя без искажения, т.е. будут содержать «0», шесть сообщений исказятся и принесут «1», ещё одно ошибочно принесет «2». Запись $\mathbf{P}(\mathbf{X}=0;\mathbf{Y}=1)=0.06$ надо читать так «вероятность того, что источник послал «0», а получатель принял «1», равна 0.06».

Пусть задан закон совместного распределения двух случайных величин $\mathbf{X}$ и $\mathbf{Y}$.

| **X** \ **Y** | $y_1$ | $y_1$ | ... | $y_m$ |
|---|---|---|---|---|
| $x_1$ | $p_{11}$ | $p_{12}$ | ... | $p_{1m}$ |
| $x_1$ | $p_{21}$ | $p_{22}$ | ... | $p_{1m}$ |
| ... | ... | ... | ... | ... |
| $x_n$ | $p_{n1}$ | $p_{n2}$ | ... | $p_{nm}$ |

Здесь $\mathbf{P}(\mathbf{X}=x_i;\mathbf{Y}=y_j)=p_{ij},\ i=\overline{1,n},\ j=\overline{1,m}$. Обозначим $p_i=\sum\limits_{j=1}^{m}p_{ij},\ q_j=\sum\limits_{i=1}^{n}p_{ij}$.

**Количество информации**, содержащейся в случайной величине $\mathbf{Y}$ относительно $\mathbf{X}$ — числовая характеристика пары случайных величин, вычисляемая по формуле:

$$I(\mathbf{X},\mathbf{Y})=\sum_{i=1}^{n}\sum_{j=1}^{m}p_{ij}\log_2\frac{p_{ij}}{p_i\,q_j},\tag{1}$$

где суммирование ведется только для ненулевых частот.

Просуммировав по столбцам частоты в таблице совместного распределения для случая, когда канал подвержен помехам, получим таблицу частот для получателя; мы обозначаем эти частоты $q_j,\ j=\overline{1,m}$.

| **Сигнал получателя** | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **Частота появления** | 0.27 | 0.43 | 0.19 | 0.09 | 0.02 |

табл. частот получателя

$$I(\mathbf{X},\mathbf{Y})=\sum_{i=1}^{4}\sum_{j=1}^{5}p_{ij}\log_2\frac{p_{ij}}{p_i\,q_j}=$$

$$=0.23\cdot\log_2\frac{0.23}{0.27\ \cdot 0.3}+0.06\cdot\log_2\frac{0.06}{0.43\ \cdot 0.3}+0.01\cdot\log_2\frac{0.01}{0.19\ \cdot 0.3}+$$

$$+0.04\cdot\log_2\frac{0.04}{0.27\ \cdot 0.4}+0.31\cdot\log_2\frac{0.31}{0.43\ \cdot 0.4}+0.05\cdot\log_2\frac{0.05}{0.19\ \cdot 0.4}+$$

$$+0.06\cdot\log_2\frac{0.06}{0.43\ \cdot 0.2}+0.12\cdot\log_2\frac{0.12}{0.19\ \cdot 0.2}+0.02\cdot\log_2\frac{0.02}{0.09\ \cdot 0.2}+$$

$$+0.01 \cdot \log_2 \frac{0.01}{0.19 \cdot 0.1} + 0.07 \cdot \log_2 \frac{0.07}{0.09 \cdot 0.1} + 0.02 \cdot \log_2 \frac{0.02}{0.02 \cdot 0.1} \approx 0,86614875.$$

Вновь рассмотрим случай, когда интенсивность помехи равна нулю; при этом, очевидно, $\mathbf{X} = \mathbf{Y}$. В этом случае

$$\mathbf{P}(\mathbf{X} = x_i; \mathbf{Y} = y_j) = \mathbf{P}(\mathbf{X} = x_i; \mathbf{X} = x_j) = \begin{cases} 0, & i \neq j; \\ p_i, & i = j, \end{cases}$$

и тогда

$$I(\mathbf{X}, \mathbf{X}) = \sum_{i=1}^{n} p_i \log_2 \frac{p_i}{p_i\, p_i} = -\sum_{i=1}^{n} p_i \log_2 p_i.$$

Полученная числовая характеристика случайной величины $\mathbf{X}$ называется **энтропией**[1] случайной величины $\mathbf{X}$ и обозначается $H(\mathbf{X})$. Таким образом

$$H(\mathbf{X}) = -\sum_{i=1}^{n} p_i \log_2 p_i. \tag{2}$$

Каков же содержательный смысл энтропии? Ожидая очередное сообщение от источника, получатель находится в состоянии неопределенности. После получения этого очередного (одного) символа, данная неопределенность снимается. Энтропия — это 1) количество неопределенности относительно того, что послал источник, которое снялось после получения этого одного символа, или, что то же самое, 2) количество информации относительно того, что послал источник, которое мы получили с этим одним символом.

Читатель может удивиться первой трактовке — что значит «сколько неопределенности снялось»? Мы знаем достоверно, что послал источник (в канале же нет помех), т.е. вся неопределенность снялась! Однако, внимательное прочтение первой трактовки позволит уловить её суть. Рассмотрим пример.

Пусть источник может равновероятно посылать одно из двух значений, т.е. таблица частот имеет вид:

| $X_1$ | 0 | 1 |
|---|---|---|
| **P** | 0.5 | 0.5 |

Согласно (2) имеем $H(\mathbf{X}_1) = -(0.5 \cdot \log_2 0.5 + 0.5 \cdot \log_2 0.5) = 1$, т.е. был получен 1 бит информации.

Пусть теперь источник может равновероятно посылать одно из тридцати двух значений, т.е. таблица частот имеет вид:

| $X_2$ | 0 | 1 | ... | 31 |
|---|---|---|---|---|
| **P** | 0.03125 | 0.03125 | ... | 0.03125 |

Согласно (2) имеем $H(\mathbf{X}_2) = -\left(\frac{1}{32} \cdot \log_2 \frac{1}{32} + ... + \frac{1}{32} \cdot \log_2 \frac{1}{32}\right) = -\log_2 \frac{1}{32} = 5$, т.е. было получено 5 бит информации.

[1]Понятие энтропии было заимствовано из термодинамики, где оно, если говорить очень приблизительно, является мерой беспорядка в системе. Информационная энтропия является мерой хаотичности информации, мерой неопределенности.

Обсудим полученные результаты. И в первом случае, и во втором, получатель, приняв сигнал от источника, точно знает, что тот посылал. Однако, во втором случае информации доставляется в пять раз больше (и, соответственно, в пять раз больше снимается неопределенности). Так происходит потому, что предугадать значение, которое пошлет источник во втором случае труднее: $2^5$ вариантов во втором случае против $2^1$ в первом.

Для полноты сравнения рассмотрим случай с неравномерным распределение частот.

| $X_3$ | 0 | 1 |
|---|---|---|
| **P** | 0.9 | 0.1 |

Очевидно, что гораздо проще предугадывать значение случайной величины $\mathbf{X}_3$ (т.к. она почти всегда равна нулю), чем $\mathbf{X}_1$ . Значит неопределенности при получении очередного символа снимается меньше, и информационная ценность получаемого символа меньше. Действительно, согласно (2) имеем $H(\mathbf{X}_3) = -(0.9 \cdot \log_2 0.9 + 0.1 \cdot \log_2 0.1) \approx 0.4689956$, т.е. было получено 0.4689956 бит информации.

Приведенные примеры показывают, что возможна и третья трактовка энтропии. Энтропия — это минимум среднего количества бит, которое нужно передавать по каналу связи о текущем состоянии источника.

В приведенных определениях количества информации (1) и энтропии (2) логарифмы вычислялись по основанию 2. Однако ничто не мешает определить эти величины, вычисляя логарифмы по другому основанию, например, $e$ или 10; все математические свойства при этом останутся. Действительно, согласно формуле перехода к другому основанию $\log_b a = \dfrac{\log_c a}{\log_c b}$, имеем

$$-\sum_{i=1}^{n} p_i \ln p_i = -\sum_{i=1}^{n} p_i \frac{\log_2 p_i}{\log_2 e} = -\frac{1}{\log_2 e} \sum_{i=1}^{n} p_i \log_2 p_i = \frac{1}{\log_2 e} H(\mathbf{X}).$$

Таким образом, переход к другому основанию эквивалентен домножению на нормировочный коэффициент.

Некоторые авторы предлагают в качестве основания логарифма брать $n$ — число возможных исходов случайной величины; или, в терминах источник-получатель, количество различных сигналов, которые может послать источник:

$$H(\mathbf{X}) = -\sum_{i=1}^{n} p_i \log_n p_i. \tag{3}$$

Такое разнообразие подходов к измерению энтропии не должно смущать читателя — вспомните, сколько существует единиц измерения длины: метр, фут, морская миля[1], световой год[2] и т.д. Все эти меры приводимы друг к другу. Точно так же формулы (2) и (3) в известном смысле эквивалентны.

[1]Первоначально морская миля определялась как длина дуги большого круга на поверхности земного шара размером в одну угловую минуту. Таким образом, перемещение на одну морскую милю вдоль меридиана примерно соответствует изменению географических координат на одну минуту широты. По современному определению, международная морская миля равна 1852 метрам.

[2]По определению Международного Астрономического Союза световой год равен расстоянию, которое свет проходит в вакууме, не испытывая влияния гравитационных полей, за один юлианский год.

## 2.3. Постановка задачи

На вход подается строка — последовательность букв. Требуется

1. построить алфавит (т.е. множество всех различных символов исходной строки) и найти частоту для всех символов алфавита;
2. для полученной таблицы рассчитать энтропию согласно (3).

Пример 1. На вход поступает строка `abcd`.

| **Алфавит** | a | b | c | d |
|---|---|---|---|---|
| **Частоты** | 0.25 | 0.25 | 0.25 | 0.25 |

энтропия равна `1`.

Пример 2. На вход поступает строка `abrakadabra`.

| **Алфавит** | a | b | r | k | d |
|---|---|---|---|---|---|
| **Частоты** | 5/11 | 2/11 | 2/11 | 1/11 | 1/11 |

энтропия равна энтропия приблизительно равна `0.87874099`.

## 2.4. Указания к решению

Проведем необходимую формализацию. Именно, введем в рассмотрение случайную величину (источник информации) **X**, о статистических свойствах которой будем судить по имеющейся строке. Множество значений **X** — множество всех различных букв строки, частоты этих значений могут найдены непосредственным подсчетом.

Для построения алфавита и таблицы частот рекомендуем воспользоваться следующим фрагментом кода

```
str=WScript.StdIn.ReadLine()
alph=new Array();
for(i=0;i<str.length;i++)
    alph[str.charAt(i)]=0
for(i=0;i<str.length;i++)
    alph[str.charAt(i)]++
```

## 2.5. Задания для самостоятельной работы

1. Одной из важных задач, рассматриваемых в курсе информатики, является задача сжатия длинных строк. Коэффициентом сжатия называется отношение длины исходной строки к длине сжатой строки.

Пусть дано две строки `0111111111` и `0112234567`. Какую строку можно эффективнее сжать? У какой строки энтропия больше? Есть ли связь между величиной энтропией и коэффициентом сжатия?

2. Прочитайте статью Shannon Claude E., A Mathematical Theory of Communication // Bell System Technical Journal, 1948. pp. 379–423, 623–656.

# 3. Виртуальная машина

## 3.1. Об архитектуре машины фон Неймана и машинных языках

Компьютер — это набор микросхем; если не оснастить его программами, он будет абсолютно бесполезен. «Сердцем» компьютера является центральный процессор, который умеет выполнять лишь элементарные операции, такие как арифметические (например, сложение двух чисел), логические (например, исключающее ИЛИ), копирование содержимого из одной области памяти в другую и т. п. Последовательность этих элементарных действий и образует программу. Таким образом, решение любой задачи, например обработка изображения в PhotoShop или решение уравнений в MATLAB, сводится к выполнению длинной последовательности арифметических, логических и других элементарных операций. Эти элементарные операции мы будем в дальнейшем называть *машинными командами*. Множество команд, понимаемых данным процессором, называется *машинным языком*.

Процессор умеет работать только с данными, хранящимися в оперативной памяти (RAM — Random Access Memory) или на регистрах. Напомним, что процессор имеет небольшой набор ячеек собственной памяти, называемых регистрами; регистры нужны процессору для непосредственного выполнения арифметических и логических операций, адресации памяти и т. д.

Процессор выполняет программу (собственно, это он только и умеет делать и все время делает). И программы, и данные, с которыми эти программы работают, хранятся в оперативной памяти, ибо, как было отмечено выше, процессор умеет работать только с тем, что загружено в память. Хранятся они там в двоичном виде (в виде пресловутых нулей и единиц). Программа, загруженная в память, состоит из большого числа машинных команд. Как же процессор понимает, какую команду ему сейчас выполнять? Особое место среди регистров процессора занимает регистр IP (Instruction Pointer). Именно в нем хранится адрес команды, которую нужно выполнять в данный момент.

Примитивизм языка машинных команд приводит к тому, что решение даже самых, казалось бы, простых задач, например вывод содержимого файла на экран, становится очень трудоемким для программиста — требует многих десятков и сотен команд.

Язык — это средство и одновременно способ общения. Язык машинных команд — средство общения программиста с машиной. В далеких 50-х программистам, пишущим на машинных языках, хотелось иметь более удобное средство общения с машиной; им нужен был язык, более близкий к человеческому языку, но в то же время столь же формальный, как и машинный, позволяющий машине однозначно толковать написанное. Примерами таких языков стали FORTRAN (1957), ALGOL (1958), COBOL (1959), позже появился C (1969—1973 ). Программы, написанные на этих языках, выглядели уже совсем «почеловечески».

```
#include <stdio.h>
int main(void){
    printf("Hello, World!\n");
    return 0;
}
```

## 3.2. О трансляторах, компиляторах, интерпретаторах и виртуальных машинах

Как бы хороша ни была программа на C++, процессору она не понятна, так как процессор понимает свой и только свой язык. Нужна программа, которая переводит текст, написанный на языке программирования высокого уровня, в язык машинных команд. Как же осуществляется этот перевод? Существует несколько подходов к этой весьма непростой проблеме.

**Трансляция программы** — преобразование программы, представленной на одном из языков программирования, в программу на другом языке, в определенном смысле равносильную первой.

Язык, на котором представлена входная программа, называется *исходным языком*, а сама программа — *исходным кодом*. Выходной язык называется *целевым языком*.

**Транслятор** — 1) программа или техническое средство, выполняющее трансляцию программы;
2) машинная программа, которая транслирует с одного языка на другой, в частности с одного языка программирования на другой.

Итак, цель трансляции — преобразовать текст с одного языка на другой, который понятен адресату. В случае программ-трансляторов адресатом является либо техническое устройство (процессор), либо программа-интерпретатор. Обсудим подробно каждый из этих вариантов.

Транслятор, который преобразует программы в машинный язык, принимаемый и исполняемый непосредственно процессором, называется **компилятором**. Словарь по программированию и информатике А. Б. Борковского дает такое определение компилятора: это программа, переводящая текст программы, написанной на языке высокого уровня, в эквивалентную программу на машинном языке[2].

Программа, подвергаемая компиляции, как правило, зависит от сервисов, предоставляемых операционной системой и сторонними библиотеками (например, файловый ввод–вывод или графический интерфейс). Машинный код, получаемый при компиляции программы, так же связан с этими сервисами, следовательно, ориентирован на исполнение в среде конкретной операционной системы. Архитектура (набор программно-аппаратных средств), для которой производится компиляция, называется *целевой машиной*. Для каждой целевой машины (IBM, Apple и т. д.) и каждой операционной системы или семейства операционных систем, работающих на целевой машине, требуется свой компилятор. (Кроме того, компиляторы могут быть оптимизированы под разные типы процессоров из одного семейства путем использования специфичных для этих процессоров инструкций. Например, код, скомпилированный под процессоры семейства Pentium, может использовать специфичные для этих процессоров наборы инструкций — MMX, SSE, SSE2.) Платформозависимость получаемого кода является недостатком этого подхода.

Компилятор полностью (от начала и до конца) просматривает исходный файл и создает исполняемый файл. Компиляция и исполнение — два разных процесса, разделенных

[2]Существуют компиляторы, которые переводят программу не в машинный код, а в программу на некотором специально созданном низкоуровневом языке (например, байт-коде), — о них речь пойдет позже.

во времени. Процессор использует исполняемый файл, ни к исходному файлу, ни к компилятору не обращаясь. Отсюда одно из достоинств копилятора — не требуется наличие компилятора на целевой машине, для которой программа была скомпилирована.

Другой метод состоит в том, что программа транслируется и параллельно с этим исполняется с помощью интерпретатора.

**Чистый интерпретатор** — 1) вид транслятора, осуществляющего пооператорную (покомандную) обработку и выполнение исходной программы или запроса;
2) программа (иногда аппаратное средство), анализирующая команды или операторы программы и тут же выполняющая их;
3) Языковый процессор, который построчно анализирует исходную программу и одновременно выполняет предписанные действия, а не формирует на машинном языке скомпилированную программу, которая выполняется впоследствии.

Чистая интерпретация применяется, как правило, для языков с простой структурой (например, АПЛ или Лисп). Интерпретаторы командной строки обрабатывают команды в скриптах в UNIX или в пакетных файлах (.bat) в MS-DOS также, как правило, в режиме чистой интерпретации.

Алгоритм работы интерпретатора следующий:
1) прочитать инструкцию;
2) проанализировать инструкцию и определить соответствующие действия;
3) выполнить соответствующие действия;
4) если не достигнуто условие завершения программы, прочитать следующую инструкцию и перейти к пункту 2.
Недостаток такого подхода состоит в том, что интерпретируемая программа не может выполняться отдельно без программы-интерпретатора — интерпретатор должен быть в наличии на целевой машине, где должна исполняться программа (ср. с компиляторами). С другой стороны, достигается платформонезависимость на уровне исходных кодов: программа будет работать на любой платформе, где установлен соответствующий интерпретатор. Также к недостаткам интерпретатора относят и то, что интерпретируемая программа выполняется медленнее, поскольку промежуточный анализ исходного кода и планирование его выполнения требуют дополнительного времени в сравнении с непосредственным исполнением машинного кода, в который мог бы быть скомпилирован исходный код. Кроме того, практически отсутствует оптимизация кода, что приводит к дополнительным потерям в скорости работы интерпретируемых программ.

Существуют промежуточные между компиляцией и чистой интерпретацией решения — система из компилятора, переводящего исходный код программы в промежуточное представление, например в байт-код, и собственно интерпретатора, который выполняет полученный промежуточный код. **Байт-код** — машинно-независимый код низкого уровня, генерируемый транслятором и исполняемый интерпретатором. Программа на байт-коде обычно выполняется интерпретатором байт-кода и, как следствие, является портируемой, т. е. может исполняться на разных платформах и архитектурах (это родовое преимущество интерпретируемых языков над компилируемыми). Преимущество таких систем над чистой интерпретацией состоит в том, что за счет выноса анализа исходного кода в отдельный, разовый проход и минимизации этого анализа в интерпретаторе достигается более высокая скорость выполнения программ. По этой причине многие современные интерпретируемые языки транслируют исходные тексты программ в байт-код и запускают

интерпретатор байт-кода. К таким языкам относятся Perl, PHP и Python.

Дальнейшее рассмотрение вопросов компиляции и интерпретации, таких как, например, Just-in-time компиляция, выходит за рамки данного учебного пособия. Заинтересовавшийся читатель должен обратиться к специальной литературе.

Интерпретатор программно моделирует машину, цикл выборки–исполнения которой работает не с машинными командами, а с командами языков высокого уровня или инструкциями байт-кода. Такое программное моделирование являет собой пример работы виртуальной машины. Рассмотрим здесь лишь один класс виртуальных машин — виртуальные машины, моделирующие работу микропроцессора. **Виртуальная машина** — программная и/или аппаратная система, эмулирующая[1] аппаратное обеспечение некоторой платформы и исполняющая программы, написанные для этой платформы (целевая платформа) на другой платформе (платформа-хозяин). Виртуальная машина исполняет некоторый машинно-независимый код, например, программный код на высокоуровневом языке (чистый интерпретатор), байт-код (интерпретатор смешанного типа) или машинный код реального процессора (виртуальная машина, исполняющая машинный код Intel на компьютере Apple). Помимо процессора, виртуальная машина может эмулировать работу как отдельных компонент аппаратного обеспечения, так и целого реального компьютера (включая BIOS, оперативную память, жесткий диск и другие периферийные устройства).

## 3.3. Постановка задачи

Требуется:
1) придумать собственный низкоуровневый язык, выразительные способности которого будут достаточны для решения простых вычислительных задач;
2) написать на этом языке программу вычисления факториала натурального числа и программу нахождения наибольшего общего делителя двух натуральных чисел;
3) написать на JScript виртуальную машину, исполняющую программы, написанные на разработанном языке; иными словами, написать на JScript интерпретатор, переводящий команды разработанного языка в команды JScript.

## 3.4. Указания к решению

Обсудим, что значит «низкоуровневый язык», о котором идет речь в постановке задачи. **Низкоуровневый язык программирования** (язык программирования низкого уровня) — язык программирования, близкий к языку машинных команд реального или виртуального процессора. Для обозначения команд в низкоуровневом языке программирования обычно применяется мнемоническое обозначение. Это позволяет запоминать команды не в виде последовательности двоичных нулей и единиц, а в виде осмысленных сокращений слов человеческого языка (обычно английских). Иногда одно мнемоническое обозначение соответствует целой группе машинных команд.

[1]Эмуляция — воспроизведение программными или аппаратными средствами либо их комбинацией работы других программ или устройств. Эмуляция позволяет выполнять компьютерную программу на платформе (компьютерной архитектуре и/или операционной системе), отличной от той, для которой она была написана в оригинале.

Среди синтаксических конструкций разрабатываемого низкоуровнего языка не будет, к примеру, «цикла `for`». Вместо этого будут команды, позволяющие непосредственно работать с памятью и регистрами (не настоящими, конечно, а эмулированными). Примерами таких команд могут быть «разместить в памяти по адресу такому-то значение такое-то» или «присвоить регистру IP значение такое-то».

Язык программирования характеризуется прежде всего *синтаксисом* и *семантикой*. Синтаксис описывает структуру программы как набора символов (безотносительно к содержанию), семантика определяет смысловое значение предложений алгоритмического языка.

Какие же средства должен предоставлять наш будущий язык? Поскольку речь идет о вычислении факториала натурального числа, то в языке должна быть команда ввода этого самого числа с клавиатуры; иначе как объяснить компьютеру, факториал чего считать. В языке C++ для ввода используется команда `cin>>a`, в Pascal — команда `read(a)`. Как видно из этих примеров, синтаксис команды ввода разный, а смысл однаковый. Договоримся, что наша команда ввода будет называться `input`. Обратим теперь внимание на то, что команды ввода данных с клавиатуры имели аргумент — куда считать, т. е. куда сохранить считанные данные. В высокоуровневых языках аргументом служит имя переменной, в низкоуровневом языке само понятие переменной может отсутствовать. Договоримся, что аргументом команды `input` будет адрес ячейки памяти, куда нужно сохранить введенное значение. Например, `input 13` означает, что введенное с клавиатуры число нужно сохранить в 13-й ячейке эмулированной памяти.

Ясно, что после того, как факториал будет посчитан, значение (хранящееся в какой-то ячейке эмулированной памяти) надо вывести на экран. Язык должен иметь необходимые для этого средства. Пусть соответствующая команда называется `output`, единственным аргументом которой будет адрес ячейки памяти, откуда надо вывести содержимое на экран. Например, `output 7` — вывести на экран содержимое 7-й ячейки эмулированной памяти.

Потребуется также команда сложения двух чисел, определим ее. Команда `add arg1 arg2 arg3` складывает числа, хранящиеся в памяти по адресам `arg1` и `arg2`, и записывает сумму в память по адресу `arg3`. По аналогии могут быть введены команды «переход по условию», «присвоение значения» и т. д.

Приведем пример программы сложения двух чисел, написанной на разрабатываемом языке.

```
input 11
input 12
add 11 12 12
output 12
```

Сразу возникают вопросы: куда этот текст писать? Где и в каком формате его сохранять? Исходный текст можно сохранить в простом текстовом формате, например, в файле `first_prog.txt`. Только что созданный файл `first_prog.txt` очень напоминает `.bat` файл MS-DOS[1].

[1]Пакетный файл (англ. batch file) — текстовый файл в MS-DOS, OS/2 или Windows, содержащий последовательность команд, предназначенных для исполнения командным интерпретатором. После запуска

Необхоимо отдавать себе отчет в том, что придумать язык, описать его синтаксис и семантику — это только половина дела. Программа `first_prog.txt` не может быть исполнена, так как еще не создан интерпретатор, понимающий команды языка, на котором написана эта программа. К его созданию мы переходим, при этом будем пользоваться термином «виртуальная машина».

Согласно приведенному выше определению, виртуальная машина — программная система, эмулирующая аппаратное обеспечение некоторой платформы. Как же виртуальная машина эмулирует оперативную память? Самый простой способ сымитировать память — объявить массив, куда можно сохранять и откуда можно считывать данные и команды. Чтобы эмулировать регистр IP, можно объявить переменную, в которой будет храниться адрес (номер ячейки массива) команды, исполняемой в данный момент. Известно, что прежде, чем исполнять программу, хранящуюся на жестком диске, ее необходимо загрузить в оперативную память и в регистре IP установить адрес первой команды этой программы. Для реализации нашей виртуальной машины считаем содержимое файла `first_prog.txt` и разместим команды и данные в ячейках массива, эмулирующего оперативную память. После этого переменной, эмулирующей регистр IP, присвоим соответствующее значение.

Программу, написанную на JScript, и реализующую виртуальную машину назовем, например, `vm.js`.

```
var mem = new Array()
var fso = new ActiveXObject('Scripting.FileSystemObject')
var text_prog= fso.OpenTextFile('first_prog.txt')
var s=''
while(!text_prog.AtEndOfStream)
    s+=text_prog.ReadLine()+' '
s+='exit'
mem=s.split(' ')
var ip=0
```

Теперь в отладочных целях выполним следующий фрагмент кода:

```
for(var count=0;count<mem.length;count++)
    WScript.echo('В ячейке ',count,' хранится ',mem[count])
```

На экране должен появиться так называемый дамп памяти:

```
В ячейке 0 хранится input
В ячейке 1 хранится 11
В ячейке 2 хранится input
В ячейке 3 хранится 12
```

пакетного файла, программа-интерпретатор (как правило, COMMAND.COM или CMD.EXE) читает его строка за строкой и последовательно исполняет команды. Основная область применения — автоматизация наиболее рутинных операций, таких как обработка текстовых файлов, копирование, перемещение, переименование, удаление файлов, работа с папками, архивация, создание резервных копий баз данных и т. п.

```
...
В ячейке 9 хранится 12
В ячейке 10 хранится exit
```

Программа `vm.js` должна моделировать цикл выборки-исполнения. Реализуется это следующим образом:

```
while(mem[ip]!='exit')
    switch(mem[ip]){
        case 'input':
            WScript.Echo('Введи значение')
            mem[mem[ip+1]]=parseFloat(WScript.StdIn.ReadLine())
            ip+=2
            break
        case 'output':
            WScript.Echo(mem[mem[ip+1]])
            ip+=2
            break
        case 'add':
            mem[mem[ip+3]]=mem[mem[ip+1]]+mem[mem[ip+2]]
            ip+=4
            break
        case 'exit':
            WScript.Quit()
    }
```

# 4. Представление чисел с плавающей запятой. Стандарт IEEE 754

## 4.1. Системы счисления

Общепринятой является десятичная система счисления — позиционная система счисления, в основании которой лежит число 10. Числа записываются с помощью десяти цифр: 0, 1, 2,..., 8, 9. В десятичной записи числа каждая цифра имеет свой «вес», зависящий от позиции, в которой эта цифра стоит. Так, число 12502 можно представить в виде $1 \cdot 10^4 + 2 \cdot 10^3 + 5 \cdot 10^2 + 0 \cdot 10^1 + 2 \cdot 10^0$, видно, что цифры, стоящие в разных разрядах, имеют разный «вес».

Число можно представлять не только в десятичной, но и в иных системах счисления, например, двоичной или восьмеричной, разложив это число по степеням числа два или, соответственно, числа восемь. Рассмотрим пример: представить число $183_{(10)}$ в двоичной, восьмеричной и троичной системах счисления (нижний индекс 10 указывает на то, в какой системе счисления записано число).

- $183 = 128 + 32 + 16 + 4 + 2 + 1 = 2^7 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0 = 10110111_{(2)}$,
- $183 = 2 \cdot 64 + 6 \cdot 8 + 7 = 2 \cdot 8^2 + 6 \cdot 8^1 + 7 \cdot 8^0 = 267_{(8)}$,
- $183 = 2 \cdot 81 + 2 \cdot 9 + 3 = 2 \cdot 3^4 + 2 \cdot 3^2 + 1 \cdot 3^1 = 20210_{(3)}$.

## 4.2. Представление целых чисел в компьютере

Известно, что целых чисел бесконечно много, строго говоря, счетно много. Понятно, что возможностей компьютера хватит лишь на то, чтобы хранить конечное число чисел из ограниченного диапазона.

Рассмотрим пример, когда для хранения целого неотрицательного числа отводится один байт. Принято, что 0 — это наименьшее из возможных чисел, которое можно сохранить в отведенном байте; при этом все биты устанавливаются равными 0. Наибольшее из возможных целых чисел, которые удается сохранить, располагая всего одним байтом, — это 255. При этом все восемь бит устанавливаются равными 1. Имеют место следующие простые соотношения: $2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0 = 128+64+32+16+8+4+2+1 = 255$. Число 1 представляется в виде 00000001, число 2 — в виде 00000010, число 3 — в виде 00000011, число 4 — в виде 00000100 и т. д.

Ясно, что ограничение сверху числом 255 является недопустимым для многих прикладных задач: так, численность деревьев в большом лесу превышает указанный порог. В языках C и Pascal для хранения целых беззнаковых чисел из диапазона $0 \div 65535$ отводится два байта (типы unsigned short и word соответственно). Принцип представления чисел остается тот же — число переводится в двоичную систему счисления, при этом количество (двоичных) разрядов на этот раз не должно превышать 16.

Для хранения знаковых целых чисел можно предложить различные подходы. Дальнейшие рассуждения проведем для случая, когда отводится один байт для хранения числа.

**Первый подход** состоит в том, что некоторый бит, например, старший, отводится для хранения знака числа (0 означает «+», 1 означает «–»), а оставшиеся биты используются для хранения абсолютной величины числа. Ясно, что при этом диапазон хранимых чисел должен, по сути, уменьшиться вдвое: от 0 до ±127. Например, число 35 (35 = 32 + 2 + 1) представляется в виде **0**0100011 (бит знака выделен жирным), а число –100 (100 = 64 + 32 + 4) — в виде **1**1100100. Наибольшее положительное **0**1111111 соответствует +127, наибольшее по модулю отрицательное **1**1111111 соответствует –127.

Данный подход имеет недостатки. Во-первых, для хранения числа 0 есть два варианта: **0**0000000 и **1**0000000, так называемый «плюс ноль» и «минус ноль». Во-вторых, сравнение и выполнение арифметических операций с числами, представленными подобным образом, требует написания нетривиальной логики. Приведем пример сравнения:

```
ЕСЛИ(биты знака разные)
    {
    ЕСЛИ(все остальные биты нулевые)
        {числа равные}/*<<плюс ноль>> равен <<минус ноль>>*/
    ИНАЧЕ
        {больше то, где бит знака нулевой}
    }
ИНАЧЕ
    {
    ЕСЛИ(бит знака нулевой)
        {больше то, где единичный бит встретиться раньше,
        в противном случае числа равны}
        /*т.е. больше то, которое больше по модулю*/
    ИНАЧЕ
        {больше то, где единичный бит встретиться позже,
        в противном случае числа равны}
        /*т.е. больше то, которое меньше по модулю*/
    }
```

Процессору выполнять такого рода сравнения сложно.

**Второй подход** состоит в том, что применяется так называемый *сдвиг*. Прежде чем излагать суть этого подхода, сделаем одно замечание, которое позволит понять природу недостатков только что рассмотренного первого подхода и наметить пути их преодоления.

Все 256 наборов из восьми бит естественным образом упорядочиваются от 00000000 до 11111111. Задача представления и хранения целых беззнаковых чисел может быть сформулирована следующим образом: «Построить взаимооднозначное соответствие целых чисел из промежутка $0 \div 255$ и наборов из 8 бит». Такое взаимооднозначное соответствие строится легко; оно представлено в таблице — большему (в лексикографическом порядке) байту соответствует большее число.

| байт | число |
|---|---|
| 11111111 | 255 |
| 11111110 | 254 |
| 11111101 | 253 |
| ... | ... |
| 10000000 | 128 |
| 01111111 | 127 |
| ... | ... |
| 00000010 | 2 |
| 00000001 | 1 |
| 00000000 | 0 |

Рассмотрим первый способ хранения целых знаковых чисел с только что изложенных позиций (построения соответствия числовых диапазонов и наборов бит).

| байт | число |
|---|---|
| 11111111 | -127 |
| 11111110 | -126 |
| ... | ... |
| 10000001 | -1 |
| 10000000 | -0 |
| 01111111 | 127 |
| 01111110 | 126 |
| ... | ... |
| 00000010 | 2 |
| 00000001 | 1 |
| 00000000 | +0 |

Здесь нарушен принцип взаимооднозначности, так как существует $+0$ и $-0$. Кроме того, нарушен естественный порядок, так как большему (в лексикографическом порядке) байту не обязательно соответствует большее число.

Набор из 256 различных байт позволяет сохранить числа в диапазоне $-128 \div 127$, который «почти симметричен» относительно нуля[1]. Числа в этом промежутке упорядочены. Составим таблицу, которая задает взаимооднозначное соответствие, сохраняющее естественный порядок.

[1]Читатель может спросить: «А почему бы не выбрать промежуток $-127 \div 128$?» Это лишь вопрос договоренности. Стандарт языков C и Pascal предписывает использовать именно $-128 \div 127$. Забегая вперед, отметим, что для хранения порядка чисел с плавающей запятой используется диапазон, где положительных чисел на одно больше, чем отрицательных.

| байт | число |
|---|---|
| 11111111 | 127 |
| 11111110 | 126 |
| 11111101 | 125 |
| ... | ... |
| 10000001 | 1 |
| 10000000 | 0 |
| 01111111 | -1 |
| ... | ... |
| 00000010 | -126 |
| 00000001 | -127 |
| 00000000 | -128 |

Сформулируем правило, позволяющее делать перевод числа в компьютерное (внутреннее) представление. *Если к числу $X$ прибавить 128, перевести сумму в двоичную систему счисления, то полученная двоичная запись определит внутреннее представление числа $X$ (т. е. значения восьми бит).*

П р и м е р. Пусть $X = 10$.
Делаем сдвиг на 128: $10 + 128 = 138$.
Переводим в двоичную систему счисления: $138_{(10)} = 128 + 8 + 2 = 10001010_{(2)}$.
Внутреннее представление числа 10: 10001010.

П р и м е р. Пусть $X = -56$.
Делаем сдвиг на 128: $-56 + 128 = 72$.
Переводим в двоичную систему счисления: $72_{(10)} = 64 + 8 = 01001000_{(2)}$.
Внутреннее представление числа –56: 01001000.

Сформулируем обратное правило, позволяющее интерпретировать внутреннее представление целых знаковых чисел. *Для того чтобы определить, какое целое знаковое число сохранено в байте, надо на этот байт посмотреть как на двоичную запись числа, перевести его в десятичную систему счисления и из полученного числа вычесть 128.*

П р и м е р. В байте записано 01111111.
Переводим в десятичную систему счисления: $01111111_{(2)} = 127_{(10)}$.
Делаем сдвиг на 128: $127 - 128 = -1$.
Внутреннему представлению 01111111 соответствует число –1.

П р и м е р. В байте записано 10001101.
Переводим в десятичную систему счисления: $10001101_{(2)} = 141_{(10)}$.
Делаем сдвиг на 128: $141 - 128 = 13$.
Внутреннему представлению 10001101 соответствует число 13.

## 4.3. Числа с фиксированной запятой

Одним из вариантов представления вещественных чисел в компьютере является использование формата с фиксированной запятой. Рассмотрим вариант, когда для хранения вещественного числа отведено два байта. Естественно один отвести для хранения целой части, а другой — дробной.

## 4.4. Научная нотация чисел

Все числа кроме нуля могут быть записаны в так называемой *научной нотации*: например, число $5243.17 = 5.24317 \cdot 10^3$ или $-0.00021 = -2.1 \cdot 10^{-4}$, или $1 = 1.0 \cdot 10^0$. Эта запись включает в себя:

*знак* +/-;

*мантиссу*, состоящую из ровно одной ведущей цифры, не равной нулю, и дробной части, которая может быть нулевой;

степень, в основании которой стоит база системы счисления, показатель степени называют *порядком*.

## 4.5. Внутреннее представление чисел с плавающей запятой. Нормализованные числа

Согласно стандарту IEEE 754, для хранения вещественных чисел отводится 4 байта: из них 1 бит для знака, 8 бит для порядка (который может быть как положительным, так и отрицательным) и 23 бита для «усеченной» мантиссы. Формат предполагает хранение нормализованных чисел, двух нулей (положительного и отрицательного), денормализованных чисел и исключений (плюс и минус бесконечность и не числа[1]).

Начнем изучение стандарта с нормализованных чисел[2]. Если число положительное, бит знака хранит 0, иначе — 1. Формат должен обеспечивать хранение как «больших» (больше 1), так и «маленьких» (меньше 1) чисел, соответственно порядок может быть как положительный, так и отрицательный.

Рассмотрим несколько примеров.

1. Число $13_{(10)} = 8 + 4 + 1 = 1101_{(2)} = 1.101_{(2)} \cdot 2^{11_{(2)}}$. Хотя показатель степени было бы правильно записывать в двоичной системе счисления, мы запишем его в привычной десятичной (а мантиссу оставим все-таки в двоичной): $13 = 1.101 \cdot 2^3$.
2. Число $-6.75_{(10)} = -(4 + 2 + 1/2 + 1/4) = -110.11_{(2)} = -1.1011 \cdot 2^2$.
3. Число $0.15625_{(10)} = 1/8 + 1/32 = 0.00101_{(2)} = 1.01 \cdot 2^{-3}$.
4. Число $1_{(10)} = 1_{(2)} = 1.0 \cdot 2^0$.
5. Число $0.2_{(10)} = 1/5 = 0.001100110011..._{(2)} = 1.1001100... \cdot 2^{-3}$.

[1]Не числа возникают, например, при попытке извлечь квадратный корень из отрицательного числа или при делении нуля на нуль.

[2]Если очень грубо, то нормализованные числа — это не очень большие и одновременно не слишком маленькие числа.

Вернемся к вопросу о представлении порядка в 8 битах, отведенных для него. Из приведенных примеров видно, что порядок есть целое число; в первом и втором примерах порядок положительный, в третьем и пятом — отрицательный, а в четвертом — равен нулю. Таким образом, задача хранения порядка вещественного числа в компьютере свелась к ранее решенной задаче хранения целого знакового числа.

Под порядок отведено 8 бит, следовательно, можно было бы хранить порядки от –128 до +127, используя сдвиг на 128. Однако, согласно стандарту IEEE 754, выбран диапазон[3] от –127 до +128 со смещением 127; при этом нулевой байт и байт, состоящий из единиц, отведены для специальных целей (хранения денормализованный чисел и бесконечностей соответственно, но об этом в дальнейшем). Таким образом, 8 бит порядка позволяют хранить порядки от –126 (00000001) до +127 (11111110) с использованием сдвига 127.

Обсудим теперь особенности хранения мантиссы. Для рассмотренных пяти примеров мантиссу можно было бы хранить в виде:

1. $1101\,\underbrace{00\ldots0}_{19}$. Здесь предполагается, что десятичный разделитель стоит после первой цифры;
2. $11011\,\underbrace{00\ldots0}_{18}$;
3. $101\,\underbrace{00\ldots0}_{20}$;
4. $1\,\underbrace{00\ldots0}_{22}$;
5. 11001100110011001100110. Здесь произошла потеря цифр, вышедших за разрядную сетку. Ничего страшного в этом нет — бесконечную дробь в 23 разряда не записать.

Видно, что ведущая цифра всегда 1. Согласно определению научной нотации числа, ведущая цифра не равна нулю, но в двоичной системе счисления существует единственная не равная нулю — это цифра 1. Таким образом, хранить ведущую единицу нет необходимости! Это красивое наблюдение позволяет в 23 битах, отведенных для мантиссы, хранить 24 цифры (ведущая единица предполагается и 23 цифры после запятой храним).

Соберем полученные ранее результаты и запишем, как хранятся разобранные пять чисел в компьютере. Для удобства восприятия бит знака, порядок и мантисса будем отделять пробелами.

1. Число $13 = 1.101_{(2)} \cdot 2^3$. Бит знака равен 0. К порядку добавляем 127: $3+127 = 130 = 10000010_{(2)}$. В мантиссе убираем ведущую единицу, получаем $101\,\underbrace{00\ldots0}_{20}$.

Числу 13 соответствует 0 10000010 101 0000000000 0000000000.

2. Число $-6.75 = -1.1011_{(2)} \cdot 2^2$. Бит знака равен 1. К порядку добавляем 127: $2+127 = 129 = 10000001_{(2)}$. В мантиссе убираем ведущую единицу, получаем $1011\,\underbrace{00\ldots0}_{19}$.

Числу –6.75 соответствует 1 10000001 101 1000000000 0000000000.

3. Число $0.15625 = 1.01_{(2)} \cdot 2^{-3}$. Бит знака равен 0. К порядку добавляем 127: $-3+127 = 124 = 01111100_{(2)}$. В мантиссе убираем ведущую единицу, получаем $01\,\underbrace{00\ldots0}_{21}$.

Числу 0.15625 соответствует 0 01111100 010 0000000000 0000000000.

4. Число $1 = 1.0_{(2)} \cdot 2^0$. Бит знака равен 0. К порядку добавляем 127: $0 + 127 = 127 = 01111111_{(2)}$. В мантиссе убираем ведущую единицу, получаем $\underbrace{00\ldots0}_{23}$.

[3] См. сноску в разд. 4.2.

Числу 1 соответствует 0 01111111 000 00000000000 0000000000.

5 Число $0.2 = 1.100110011..._{(2)} \cdot 2^{-3}$. Бит знака равен 0. К порядку добавляем 127: $-3 + 127 = 124 = 01111100_{(2)}$. В мантиссе убираем ведущую единицу, получаем 100 1100110011 0011001100.
Числу 0.2 соответствует 0 01111100 100 1100110011 0011001100.

В качестве упражнения предлагаем читателю перевести во внутреннее представление числа 2, –20.09375, 1115139.

Формат float предусматривает возможность хранения +0 и –0, внутреннее представление 0 00000000 000 00000000000 0000000000 и 1 00000000 000 00000000000 0000000000 соответственно. Отметим, что если для целых наличие +0 и –0 — существенный недостаток формата, то для вещественных вовсе нет. Причина в том, что особо малые положительные числа, которые уже не удается сохранить, сбрасываются в +0, а отрицательные — в –0. Эти две величины математик-программист может и должен интерпретировать поразному. Здесь уместно вспомнить математический анализ, который учит, что стремление к +0 и к –0 — это не одно и то же!

## 4.6. Преимущества формата float над форматом fixed

Для того чтобы лучше понять причины использования чисел с плавающей запятой, и в чем преимущество формата float, разберем понятие *относительной погрешности.*

Рассмотрим содержательный пример. Во время измерения длины допускают погрешность ±1 мм. Много это или мало? Ответить на этот вопрос нельзя, ибо все познается в сравнении. Если измеряют размеры зубного протеза, то это недопустимо большая погрешность, если определяют размер строительной балки — вполне допустимая, а если измеряют глубину угольной шахты, то 1 мм — это излишняя точность, которая будет только мешать в расчетах.

Погрешность $A = 1$ мм — это абсолютная погрешность. Во всех трех случаях она одинаковая. Важнейшей характеристикой точности является относительная погрешность, которая определяется как отношение абсолютной погрешности к измеренной величине. Так, если в результате измерений получились следующие значения: размер зуба — 1 см, балки — 10 м, шахты — 100 м, то $\Delta_1 = \frac{A}{1\text{ см}} = 10\% > \Delta_2 = \frac{A}{10\text{ м}} > \Delta_3 = \frac{A}{100\text{ м}} = 0.001\%$.

Курс «Методы приближенных вычислений» учит, что абсолютная погрешность отбрасывания (не округления) цифры не превосходит единицы разряда последней оставленной цифры (и эта оценка неулучшаема). Например, для числа $\pi \approx 3.141592654$, хранимого с двумя знаками после запятой, 3.14, $A \le 10^{-2}$, а если хранить с четырьмя знаками после запятой, то 3.1415, $A \le 10^{-4}$. В числах с фиксированной запятой абсолютная погрешность, связанная с конечностью разрядной сетки, составляет $2^{-(k+1)}$, где $k$ — это количество разрядов после запятой, и эта величина постоянная (в том смысле, что не зависит от хранимого числа). Храним ли мы число $1\underbrace{00\ldots0}_{k-1}.\underbrace{00\ldots0}_{k}$, или $1.\underbrace{00\ldots0}_{k}$, или $0.\underbrace{00\ldots0}_{k-1}1$, абсолютная погрешность $A$ не превосходит $2^{-k}$. При этом относительная погрешность возрастает и составляет $2^{-2k}$, $2^{-k}$ и $2^0 = 100\%$ соответственно.

Числа с плавающей запятой реализуют другой подход — постоянной является относи-

тельная погрешность, а абсолютная растет с ростом хранимого числа[3]. Для того чтобы понять это, определим сначала роли, которые играют порядок и мантисса.

Порядок отвечает за «великость»/«малость» числа. Действительно, меняя порядок, можно получить «большие» числа, например $2^{127}$, «средние» — 1 или 10, а также «малые», например $2^{-126}$; изменения мантиссы такого разброса дать не можгут. В самом деле, числа с одним порядком отличаются не более чем в `база_системы_счисления` раз (т. е. в рассматриваемом случае не более чем в 2 раза[1]). Читатель может вспомнить фразы «это величины одного порядка», «на порядок больше».

Мантисса отвечает за точность представления числа а именно она обеспечивает хранение 23 двоичных знаков после запятой. Число 1 хранится в виде 0 01111111 000 00000000000 0000000000. Число $1 + 2^{-24}$ будет иметь точно такое же внутреннее представление! Оно может быть представлено в виде $1.\underbrace{00\ldots0}_{23}1$; последняя единица выходит за разрядную сетку и теряется. Чему равна абсолютная погрешность представления числа $1 + 2^{-24}$ во float? Очевидно, она в точности равна $2^{-24}$. Вообще, все числа из промежутка $[1; 1 + 2^{-23})$ имеют то же внутреннее представление, что и 1. Для дальнейшего будет удобно представить данный промежуток в виде $[1; 1 + 2^{-23})\, 2^0$. Если дать оценку абсолютной погрешности представления любого числа из промежутка $[1; 1 + 2^{-23})\, 2^0$, то получим $A_1 = 2^{-23}$.

Число 2 хранится в виде 0 10000000 000 00000000000 0000000000. Число $2 + 2^{-23}$ будет иметь точно такое же внутреннее представление. Оно может быть представлено в виде $10.\underbrace{00\ldots0}_{22}1$, или, что то же самое, $1.\underbrace{00\ldots0}_{23}1{\cdot}2$; последняя единица выходит за разрядную сетку и теряется. Абсолютная погрешность представления числа $2 + 2^{-23}$ во float равна $2^{-23}$. Все числа из промежутка $[2; 2 + 2^{-22})$ имеют то же внутреннее представление, что и 2. Представим данный промежуток в виде $[1; 1 + 2^{-23})\, 2^1$. Если дать оценку абсолютной погрешности представления любого числа из промежутка $[1; 1 + 2^{-23})\, 2^1$, то получим $A_2 = 2^{-22}$.

Число 1024 хранится в виде 0 10001001 000 00000000000 0000000000. Число $1024 + 2^{-14}$ будет иметь точно такое же внутреннее представление. Оно может быть представлено в виде $1\,\underbrace{00\ldots0}_{10}.\underbrace{00\ldots0}_{13}1$, или, что то же самое, $1.\underbrace{00\ldots0}_{23}1 \cdot 2^{10}$; последняя единица снова выходит за разрядную сетку и теряется. Абсолютная погрешность представления числа $1024 + 2^{-14}$ во float равна $2^{-14}$. Все числа из промежутка $[1024; 1024 + 2^{-13})$ имеют то же внутреннее представление, что и 1024. Представим данный промежуток в виде $[1; 1 + 2^{-23})\, 2^{10}$. Если дать оценку абсолютной погрешности представления любого числа из промежутка $[1; 1 + 2^{-23})\, 2^{10}$, получим $A_{1024} = 2^{-13}$.

Какова относительная погрешность представления чисел в этих трех примерах? Из определения следует, что одинаковая:

$$\Delta_1 = \frac{A_1}{1} = \frac{2^{-23}}{1} = 2^{-23},$$

[3]В этом разделе речь будет идти о нормализованных числах, для денормализованных эти рассуждения неверны. Однако денормализованные числа — это особый случай, и общие принципы он не отменяет.

[1]Если давать неулучшаемую оценку, то, в связи с конечностью разрядной сетки, — не более чем в $(2 - 2^{-23})$ раза.

$$\Delta_2 = \frac{A_2}{2} = \frac{2^{-22}}{2} = 2^{-23},$$
$$\Delta_{1024} = \frac{A_{1024}}{1024} = \frac{2^{-13}}{2^{10}} = 2^{-23}.$$

Еще раз подчеркнем, что в практически важных задачах (расчет строительных конструкций, обработка результатов геологоразведки, навигация по геофизическим полям, распознавание лиц на изображениях и т. д.) необходимо обеспечить малость именно относительной погрешности. Формат float обеспечивает относительную погрешность $2^{-23}$ как для больших, так и для малых чисел.

## 4.7. Внутреннее представление чисел с плавающей запятой. Денормализованные числа

Выясним, насколько большие/малые числа можно хранить, используя выше описанную схему; речь идет только о положительных числах, так как для отрицательных все то же самое, отличие только в бите знака. Самое большое число имеет вид 0 11111110 111 11111111111 1111111111. Действительно, все возможности порядка исчерпаны (порядок из восьми единиц 11111111 — это исключение, соответствующее $\pm$ бесконечности или не числам), и мантисса самая большая — все единицы. Это соответствует числу $(2 - 2^{-23}) \cdot 2^{127}$. Самое маленькое (нормализованное) число имеет вид 0 00000001 000 00000000000 0000000000. Действительно, все возможности порядка исчерпаны (порядок из восьми нулей 00000000 используется для хранения денормализованных чисел), и мантисса самая маленькая — все нули. Это соответствует числу $1 \cdot 2^{-126}$.

Необходимость хранить особо маленькие числа (меньше, чем $2^{-126}$) приводит к понятию денормализованного числа. Формат представления денормализованного числа предполагает:

1) все биты порядка равны 0;

2) мантисса сордержит хотя бы один ненулевой бит.

При этом интерпретировать числа надо следующим образом:

1) порядок равен –126;

2) ведущая цифра мантиссы равна 0.

Казалось бы, надо интерпретировать порядок 00000000 как число –127. Почему же стандарт устанавливает иное? Формальное объяснение этого явления состоит в том, что у нормализованных чисел ведущая цифра равна 1, а у денормализованных — 0. Разберем этот вопрос подробно. Самое большое денормализованное число, которое можно представить во float, — это 0 00000000 111 11111111111 1111111111. Ему соответствует $2^{-126}\, 0.\underbrace{11\ldots1}_{23} = 2^{-126}(1 - 2^{-23})$, т. е. оно почти вплотную подходит к самому маленькому нормализованному числу $2^{-126}$. Это очень хорошо — между двумя диапазонами нет большого разрыва. А что бы случилось, если бы порядок 00000000 интерпретировался как число –127? Тогда самое большое денормализованное число было бы равно $2^{-127}(1-2^{-23})$, т. е. оно не то что не подходит вплотную к самому маленькому нормализованному числу $2^{-126}$, а в 2 раза его меньше! Имеет место большой разрыв.

Возможно, читателю будет проще понять это рассуждение на примере с десятичными числами. Есть последовательность 1, 2, 3, ..., 9, 10. К ней спереди надо приписать либо

0.1, 0.2, 0.3, ..., 0.8, 0.9, либо 0.01, 0.02, 0.02, ..., 0.08, 0.09. В первом случае сочленение более «гладкое». Здесь прослеживается прямая аналогия с сочленением диапазонов нормализованных и денормализованных чисел.

Чего добиваются, вводя денормализованные числа, фактически договариваясь о ведущем нуле? Мантисса отчасти берет на себя «функцию порядка», позволяет уменьшать числа во много раз. Например, 0 00000000 100 0000000000 0000000000 $= 2^{-127}$, 0 00000000 000 0100000000 0000000000 $= 2^{-131}$. Заметим, что при сохранении все более малых чисел происходит деградация точности. Так, если в мантиссе после запятой стоит $k$ нулей, то только $23-k$ цифр отвечают за точность. Самое маленькое денормализованное число 0 00000000 000 0000000000 0000000001 $= 2^{-126} \cdot 2^{-23} = 2^{-149}$; при этом относительная погрешность представления составляет $2^0 = 100\%$, а не $2^{-23}$, как у нормализованных чисел.

## 4.8. Исключения: бесконечность и не число

Значения, большие чем $(2 - 2^{-23}) \cdot 2^{127}$, интерпретируются как плюс бесконечность. Такие значения представлены в виде 0 11111111 000 0000000000 0000000000, т. е. бит знака равен 0, все биты порядка равны 1, все биты мантиссы равны 0. Аналогично хранится минус бесконечность, 1 11111111 000 0000000000 0000000000.

В результате действий с числами могут возникать не числа (англ. not a number). Причины могут быть разные: недопустимая арифметическая операция (например, извлечение квадратного корня из отрицательного числа), умножение бесконечности на ноль, использование неопределенной переменной и т. д. Не числа бывают двух видов: сигнальные и несигнальные (тихие) (англ. Signalling NaN и Quite NaN соответственно). Сигнальные не числа хранятся в виде
0 11111111 $xxx\,xxxxxxxxx\,xxxxxxxxx$, где хотя бы один бит мантиссы равен 1 (иначе получили бы плюс бесконечность). Несигнальные не числа хранятся в виде
1 11111111 $xxx\,xxxxxxxxx\,xxxxxxxxx$, где хотя бы один бит мантиссы равен 1 (иначе получили бы минус бесконечность). Как видно, существует много представлений не чисел.

## 4.9. Сложение чисел с плавающей запятой

Рассмотрим алгоритм сложения чисел с плавающей запятой сначала для нормализованных чисел одного порядка, потом для чисел разного порядка. Как производить сложение, если хотя бы одно из слагаемых денормализовано, будет понятно из соображений аналогии.

Рассмотрим пример: требуется найти сумму $17 + 22$.

Имеем $17_{(10)} + 22_{(10)} = 1.0001 \cdot 2^4 + 1.0110 \cdot 2^4 = 10.0111 \cdot 2^4 = 1.00111 \cdot 2^5$. При сложении чисел одного порядка (в силу того, что ведущая цифра равна 1) сумма будет иметь порядок на 1 больший, чем у слагаемых. Таким образом, надо 1) вспомнить, что у мантисс ведущая цифра равна 1; 2) сложить мантиссы; 3) запятую в сумме сдвинуть на одну позицию влево

(при этом 23-я цифра мантиссы выйдет за разрядную сетку) и порядок увеличить на 1; 4) ведущую единицу суммы мысленно выбросить. Ведущие единицы мантиссы выделены курсивом.

$$\begin{array}{rl} 17 = & 0\ \ 10000011\ \ \textit{1}\,000\,1000000000\,0000000000 \\ 22 = & 0\ \ 10000011\ \ \textit{1}\,011\,0000000000\,0000000000 \end{array} .$$

Складываем мантиссы:

$$39 = \quad 0\ \ 10000011\ \ \textit{10}\,011\,1000000000\,0000000000\ .$$

Сдвигаем запятую в сумме на одну позицию влево и увеличиваем порядок на 1:

$$39 = \quad 0\ \ 10000100\ \ \textit{1}\,001\,1100000000\,0000000000\ .$$

Мысленно выбрасываем ведущую единицу суммы:

$$39 = \quad 0\ \ 10000100\ \ 001\,1100000000\,0000000000\ .$$

Правило сложения чисел с разными порядками гласит: *порядок меньшего слагаемого должен быть выравнен до порядка большего слагаемого за счет приписывания ведущих нулей к меньшему.*

Рассмотрим пример: требуется найти сумму $17 + 3$.

Имеем $17_{(10)} + 3_{(10)} = 1.0001 \cdot 2^4 + 1.1 \cdot 2^1 = 1.0001 \cdot 2^4 + 0.0011 \cdot 2^4 = 1.01 \cdot 2^4$. Ведущие единицы мантиссы выделены курсивом.

$$\begin{array}{rl} 17 = & 0\ \ 10000011\ \ \textit{1}\,000\,1000000000\,0000000000 \\ 3 = & 0\ \ 10000000\ \ \textit{1}\,100\,0000000000\,0000000000 \end{array} .$$

Выравниваем порядки и пишем ведущие нули у меньшего слагаемого:

$$\begin{array}{rl} 17 = & 0\ \ 10000011\ \ \textit{1}\,000\,1000000000\,0000000000 \\ 3 = & 0\ \ 10000011\ \ \textit{0}\,001\,1000000000\,0000000000 \end{array} .$$

Складываем мантиссы:

$$20 = \quad 0\ \ 10000011\ \ \textit{1}\,010\,0000000000\,0000000000\ .$$

Мысленно выбрасываем ведущую единицу суммы:

$$20 = \quad 0\ \ 10000011\ \ 010\,0000000000\,0000000000\ .$$

## 4.10. Постановка задачи

Написать программу, которая работает в одном из двух режимов: перевод или выполнение арифметических операций.

В первом режиме программа считывает из файла `in.txt` строку: 1) если строку можно интерпретировать как число, то в файл `out.txt` записывается внутреннее представление этого числа, в частности, если число по модули превосходит $(2 - 2^{-23}) \cdot 2^{127}$, то в файл

`out.txt` записывается внутреннее представление бесконечности; 2) если строку нельзя интерпретировать как число, то в файл `out.txt` записывается внутреннее представление не числа.

Во втором режиме программа считывает из файла `in.txt` строку, содержащую два числа, разделенных операцией + или −. Программа переводит операнды во внутреннее представление и выполняет сложение/вычитание над операндами в их внутреннем представлении согласно правилам стандарта IEEE 754. В файл `out.txt` записывается результат выполнения арифметической операции во внутреннем и в десятичном виде.

Режим работы указывается в качество параметра командной строки.

П р и м е р. `CScript float.js conv`

| `input.txt` | 340282340 0000000000 0000000000 0000000000 |
|---|---|
| `out.txt` | 0 11111110 111 1111111111 1111111111 |

П р и м е р. `CScript float.js conv`

| `input.txt` | 340282360 0000000000 0000000000 0000000000 |
|---|---|
| `out.txt` | 0 11111111 000 0000000000 0000000000 |

П р и м е р. `CScript float.js conv`

| `input.txt` | 10 негритят |
|---|---|
| `out.txt` | 0 11111111 100 0000000000 0000000000 |

П р и м е р. `CScript float.js calc`

| `input.txt` | 3.1 + 16.07 |
|---|---|
| `out.txt` | 0 10000011 001 1001010111 0000101000 ∼ 19.17 |

## 4.11. Задания для самостоятельной работы

1. Для какого максимального $x$ верно (в смысле машинной арифметики, формат float) равенство $1 + x = 1$? А равенство $100 + x = 100$?

2. Верно ли утверждение, что суммы $1 + 2 + 3 + 4 + \ldots + 999 + 1000$, полученные при сложении слева направо и при сложении справа налево (вычисления проводятся в формате float), совпадают? Верно ли то же утверждение для суммы $1 + 2 + 3 + 4 + \ldots + 10^{12}$?

3. Всегда ли верно равенство $x/2 + x/2 = x$? Вычисления проводятся в формате float.

4. Всегда ли верно равенство $(x + 2^{20}) - x = 2^{20}$? Вычисления проводятся в формате float, скобки задают порядок выполнения действий.

5. Придумайте свой пример, показывающий, что вычисления во float могут давать результат, существенно отличный от истинного.

# 5. Поиск подстроки в строке. Алгоритм грубой силы и использование хэшей

## 5.1. Постановка задачи

Дана строка $S$ длины $n$ и строка $T$ длины $m$. В дальнейшем, не оговаривая каждый раз отдельно, будем предполагать, что $n \geq m$. Будем нумеровать символы строк начиная с 1. Обозначим через $S[i\,..\,j],\ 1 \leq i \leq j \leq n$, подстроку строки $S$, начинающуюся в $i$-й позиции и заканчивающуюся в $j$-й позиции. Требуется найти все вхождения строки $T$ в строку $S$, т. е. указать все позиции строки $S$, начиная с которых читается строка $T$. Множество вхождений, которое надо построить, формально описывается так: $M = \{i | 1 \leq i \leq n - m + 1,\ S[i\,..\,i + m] = T\}$.

Например, пусть $S = abcabdecab$, $T = cab$, тогда $M = \{3; 8\}$. Или такой пример: пусть $S = abababacaba$, $T = aba$, тогда $M = \{1; 3; 5; 9\}$. Таким образом, мы допускаем «перекрытие» — ситуацию, когда расстояние между двумя соседними вхождениями меньше длины строки $T$.

## 5.2. Метод решения: алгоритм грубой силы

Наиболее очевидным и простым является алгоритм грубой силы (англ. brute force). Он состоит в последовательном посимвольном сравнении строк.

Начало шаблона $T$ совмещается с началом строки $S$. Проверка начинается с первого символа шаблона $T$. Если символы совпали, то производится сравнение следующего символа и т. д. Возможна одна и только одна из двух ситуаций:

I. Если все символы шаблона совпали с соответствующими символами подстроки, то подстрока найдена; надо 1) сделать соответствующую запись об этом вхождении и 2) сдвинуть шаблон на 1 символ вправо и начать новый поиск.

II. Если какой-то символ шаблона не совпал с соответствующим символом строки, то надо сдвинуть шаблон на 1 символ вправо и начать новый поиск.

Проверку осуществлять до тех пор, пока есть куда сдвигать шаблон, т. е. пока правый конец шаблона не выйдет за правый край строки $S$ в результате очередного сдвига.

Внимательное рассмотрение этого описания позволяет увидеть, что сдвиг шаблона на 1 символ вправо производится независимо от того, все символы шаблона совпали с соответствующими символами строки $S$ или нет.

```
i=1
Пока (i<=n-m+1)
Начало_цикла
   j=1
   Пока (S[i+j-1]==T[j])
   Начало_цикла
       j++
       Если (j==m+1)
```

```
        Начало_если
            Вывести (i)
            Выйти_из_цикла
        Конец_если
    Конец_цикла
    i++
Конец_цикла
```

Здесь оператор `Выйти_из_цикла` означает выход из внутреннего цикла.

## 5.3. Метод решения: использование хэшей

Каждому символу, входящему в строку, можно поставить в соответствие число, например, это может быть код символа в таблице ASCII (разумеется, если символ принадлежит этой таблице). Так, например, для английских букв Код(A)=65, Код(Z)=90, Код(a)=97, Код(z)=122, для русских букв Код(А)=192, Код(Я)=223, Код(а)=224, Код(я)=255.

**Утверждение 1.** Если две строки $S_1$ и $S_2$ совпадают, то сумма кодов символов строки $S_1$ совпадает с суммой кодов символов строки $S_2$.

**Утверждение 2.** Если сумма кодов символов строки $S_1$ не совпадает с суммой кодов символов строки $S_2$, то строки $S_1$ и $S_2$ не совпадают.

Если использовать это простое наблюдение при поиске подстроки в строке, то можно избежать целого ряда проверок, которые заведомо покажут несоответствие шаблона и фрагмента строки.

Пусть, например, $S = aaaaabc$, $T = aaab$.

$\sum_{k=1}^{4} \text{Код}(T[k]) = 97 + 97 + 97 + 98 = 389.$

$\sum_{k=1}^{4} \text{Код}(S[k]) = 97 + 97 + 97 + 97 = 388 \neq 389$, следовательно, $S[1\,..\,4] \neq T$.

$\sum_{k=2}^{5} \text{Код}(S[k]) = 388 \neq 389$, следовательно, $S[2\,..\,5] \neq T$.

$\sum_{k=3}^{6} \text{Код}(S[k]) = 389 = 389$, следовательно, возможно совпадение фрагмента $S[3\,..\,6]$ и шаблона $T$; необходимо сравнивать строки посимвольно. Непосредственная проверка показывает совпадение.

$\sum_{k=4}^{7} \text{Код}(S[k]) = 391 \neq 389$, следовательно, $S[4\,..\,7] \neq T$.

Разумеется, возможна ситуация, когда суммы кодов символов совпадут, а сами строки будут отличаться. Такая ситуация называется *коллизией*.

Пусть, например, $S = bdfh$, $T = aegg$.

$\sum_{k=1}^{4} \text{Код}(T[k]) = 97 + 101 + 103 + 103 = 403$, $\sum_{k=1}^{4} \text{Код}(S[k]) = 98 + 99 + 102 + 104 = 403$, т. е. имеет место коллизия.

Пусть, например, $S = abcabc$, $T = cab$.

$\sum_{k=1}^{3} \text{Код}(T[k]) = 99 + 98 + 97 = 294.$

$S[1\,..\,3] = abc$, $\sum_{k=1}^{3} \text{Код}(S[k]) = 97 + 98 + 99 = 294$, коллизия.

$S[2\,..\,4] = bca$, $\sum_{k=2}^{4} \text{Код}(S[k]) = 98 + 99 + 97 = 294$, коллизия.

$S[3\,..\,5] = cab$, $\sum_{k=3}^{5} \text{Код}(S[k]) = 99 + 98 + 97 = 294$, посимвольная проверка показывает совпадение $S[3\,..\,5]$ и шаблона $T$.

$S[4\,..\,6] = abc$, $\sum_{k=4}^{6} \text{Код}(S[k]) = 97 + 98 + 99 = 294$, коллизия.

Для дальнейшего нам потребуется понятие хэш-функции. **Хэш-функцией** $h(s)$, определенной на строке $s$, мы будем называть произвольную ограниченную целочисленную функцию этой строки. Сумма ASCII кодов символов строки являет собой пример простейшей хэш-функции. Часто также рассматривают сумму кодов символов строки по модулю большого числа.

Приступим к построению алгоритма поиска подстроки в строке с использованием хэшей. Главное отличие от метода brute force состоит в том, что на шаге, где проводится сопоставление $S[i\,..\,i+m-1]$ с $T$, сперва вычисляется хэш $h(S[i\,..\,i+m-1])$, который потом сравнивается с хэшем $h(T)$, и только в случае равенства хэшей осуществляется посимвольная проверка, как в brute force. Если хэш шаблона можно вычислить однажды, а именно до начала всех проверок, то хэши фрагментов строки $S$ необходимо вычислять отдельно для каждого исследуемого фрагмента $S[i\,..\,i+m-1]$, $1 \le i \le n-m+1$, т. е. $n-m+1$ раз. Возможны различные подходы к вычислению хэшей фрагментов $S[i\,..\,i+m-1]$; сравним два способа.

**Первый способ** вычисления: $h(S[i\,..\,i+m-1]),\ 1 \le i \le n-m+1$.

```
hs=0
k=0
Пока (k<=m-1)
Нчало_цикла
   hs+=Код_символа(S[i+k])
   k++
Конец_цикла
```

**Второй способ** вычисления: $h(S[i\,..\,i+m-1]),\ 1 < i \le n-m+1$, при этом предполагается, что $h(S[1\,..\,m])$ будет вычислено заранее.

```
hs=hs + Код_символа(S[i+m-1]) - Код_символа(S[i-1])
```

Если в первом способе хэш фрагмента каждый раз вычисляется с нуля, то во втором используются ранее полученные результаты, чтобы после «небольшой» поправки получить требуемое. Эта оптимизация, называемая *рекуррентное вычисление*, дает особенно

ощутимый эффект, если длина шаблона $m$ велика. Вычисление хэш-функции должно производиться быстро, независимо от $m$. Собственно только в этом случае использование хэшей дает преимущество над методом грубой силы!

Наряду с рассмотренной ранее хэш-функцией, вычисляемой как сумма кодов символов, можно предложить и другие. Например сумма квадратов кодов также допускает рекуррентное вычисление:
$h_2(S[i\,..\,i+m-1]) = h_2(S[i-1\,..\,i+m]) + \text{Код}^2(S[i+m-1]) - \text{Код}^2(S[i-1])$.
Возникает естественный вопрос — какую предпочесть? При прочих равный условиях отдавать предпочтение надо той хэш-функции, которая дает меньше коллизий.

Ранее были рассмотрены примеры строк, доставляющих коллизию хэш-функции типа «сумма кодов символов»: $S = bdfh$, $T = aegg$ и $S = abc$, $T = cab$. Проверим, имеет ли место коллизия для хэш-функции типа «сумма квадратов кодов символов» $h_2(bdfh) = 40828 \neq h_2(aegg) = 40625$, $h_2(abc) = h_2(cab)$. То есть в первом случае нам удалось избежать коллизии, а во втором нет, так как строки $abc$ и $cab$ отличаются только порядком букв, а сумма квадратов от порядка слагаемых не зависит.

Нельзя ли предложить хэш-функцию, которая дает различные значения для строк, отличающихся лишь порядком символов, как то: $abc$ и $cab$? Рабином и Карпом была предложена функция $h_r(s) = \sum_{k=1}^{n} \text{Код}(s[k]) \cdot 2^{n-k}$. Эта функция допускает рекуррентное вычисление

$$h_r(S[i\,..\,i+m-1]) = \\ \Big(h_r(S[i-1\,..\,i+m]) - \text{Код}(S[i-1]) \cdot 2^{m-1}\Big) \cdot 2 + \text{Код}(S[i+m-1]).$$

Практика показывает, что на среднестатистическом тексте хэш-функция Рабина–Карпа дает меньше коллизий, чем хэш-функция типа «сумма кодов» или «сумма квадратов кодов символов».

В методических целях ещё раз назовем свойства, которым должна удовлетворять «хорошая» хэш-функция.

1. Возможность рекуррентного вычисления; как следствие, высокая скорость нахождения значений функции, не зависящая от длины строки.

2. Низкая вероятность коллизий.

Вопрос о том, как оценить вероятность коллизий, является сложным и далеко выходящим за рамки настоящего пособия. Однако приведем некоторые соображения:

1. Эта вероятность зависит от соотношения между мощностью множества определения хэш-функции и мощностью множества значений этой хэш-функции. Для примера рассмотрим строки, состоящие из пяти маленьких английских букв, т. е. $aaaaa \div zzzzz$. Определим две хэш-функции: $h(s) = \sum_{k=1}^{5} \text{Код}(s[k])$ и $h^*(s) = \Big(\sum_{k=1}^{5} \text{Код}(s[k])\Big) \mod 2$. Вторая функция принимает ровно два значения (0 и 1), т. е. разбивает строки на два класса; ясно, что вероятность коллизии для первой хэш функции значительно меньше, чем для второй.

Стоит, однако, помнить, что увеличение мощности множества значений хэш-функции может увеличить время сравнения хэшей. Сравнение однобайтных чисел $0 \div 2^8 - 1$ производится быстрее[1], чем четырехбайтных $0 \div 2^{32} - 1$. Выигрыш во времени достигается

[1] Вообще это зависти от архитектуры.

именно за счет того, что сравниваются короткие числа, а не длинные строки. Здесь имеет место борьба противоположностей: время сравнения — вероятность коллизии.

2. Вероятность коллизии зависит от закона распределения значений хэш-функции. Хорошим законом такой, когда вероятность появления любого значения одинаковая. Хэш-функция типа «сумма кодов» таким свойством не обладает. Действительно, значение 485 появляется только на строке *aaaaa*, а значение 486 появляется в пять раз чаще — на строках *aaaab*, *aaaba*, ..., *baaaa*. Читателю в качестве упражнения предлагается посчитать количество строк, имеющих ту же сумму кодов, что и строка $mmmmm$.

## 5.4. О вычислительной сложности алгоритмов

Известно, что для решения одной и той же задачи, как правило, можно предложить несколько (иногда существенно различающихся) алгоритмов. Возникает необходимость сравнения этих алгоритмов. Теория Computer Science уделяет наибольшее внимание таким критериям, как скорость работы и объем требуемой памяти. На практике, кроме того, учитывается простота (следовательно, дешевизна) реализации и простота использования.

В теории алгоритмов вычислительная сложность алгоритма — это функция, определяющая зависимость объема работы, выполняемой некоторым алгоритмом, от размера входных данных. Раздел, изучающий вычислительную сложность, называется теорией сложности вычислений. Объем работы обычно измеряется абстрактными понятиями времени и пространства, называемыми вычислительными ресурсами. Время определяется количеством элементарных шагов, необходимых для решения задачи, тогда как пространство определяется объемом памяти или места на носителе данных. Таким образом, теория сложности вычислений ставит перед собой задачу ответить на центральный вопрос разработки алгоритмов: как изменится время исполнения и объем занятой памяти в зависимости от размера входных данных? Здесь под размером входных данных понимается длина описания данных задачи в битах; например, в задаче поиска подстроки в строке размер входа — это длины строки и подстроки.

Поясним вышесказанное на примерах. Том Сойер тратит на побелку забора время $t$. Если длина забора увеличится в два раза, во сколько раз больше потратит времени Том Сойер? Ответ очевиден — в два раза. А если бы длина забора увеличилась в пять раз, то и время побелки — в пять раз.

Василий красит пол в квадратной комнате. Если длина стороны комнаты увеличится в два раза, во сколько раз больше времени потратит Василий? Несложные вычисления показывают — в четыре раза. А если длина стороны комнаты увеличится в пять раза, то затраченное время возрастет в двадцать пять раз. Рекомендуем читателю задать себе аналогичные вопросы для случая, когда Василий копает яму в виде куба.

В итоге приходим к выводу, что в случае с побелкой забора затрачиваемое время возрастает *линейно* с увеличением длины забора, в случае с покраской пола — *квадратично*, в случае с рытьем ямы — *кубически*.

Можно показать, что сложность алгоритма грубой силы в задаче поиска подстроки длиной $m$ в строке длиной $n$ пропорциональна произведению $nm$.

## 5.5. Вычислительные эксперименты

Предлагаем провести несколько тестов, которые позволят оценить время работы алгоритмов поиска подстроки в строке. Проводя эксперименты, надо соблюдать рад требований. Во-первых, не нагружать систему посторонними задачами. Если параллельно с программой поиска работает еще десяток приложений (решается сложная система уравнений в MATLAB, просчитывается трехмерная сцена в 3d Max, играется музыка и т. д.), то все это существенно исказит результаты. Во-вторых, эксперимент должен быть проведен неоднократно, чтобы усреднить полученные результаты и тем самым снизить влияние случайных факторов.

Готовя отчет по результатам экспериментов, необходимо описывать систему: указывать по крайней мере тактовую частоту процессора, объем оперативной памяти, операционную систему. Отчет должен быть таким, чтобы посторонний человек мог, прочитав его, в точности воспроизвести описанный эксперимент и получить аналогичный результат.

ТЕСТ 1. Увеличение длины строки, в которой ищется подстрока (brute force).
В тексте романа «Война и мир» Л. Н. Толстого ищем подстроку «князь Андрей». Сравнить время поиска в первом томе, первых двух томах, первых трех томах, во всех четырех.

ТЕСТ 2. Увеличение длины искомой подстроки (brute force).
В тексте романа «Война и мир» Л. Н. Толстого ищем последовательно подстроки «князь», «князь Андрей», «князь Андрей Болконский». Сравнить время поиска. Соотносится ли время поиска как $5 : 12 : 23$? Объяснить результат. Как это согласуется (не согласуется) с общей теорией?
Сделать то же для подстрок «фортификация которых производилась» и « которых производилсь» (первый символ — пробел).

ТЕСТ 3. Сравнение brute force и хэшей.
Строка: миллион букв $a$. Подстроки: $T_1 = \underbrace{a \ldots a}_{100 \text{ раз}} b$ и $T_2 = b \underbrace{a \ldots a}_{100 \text{ раз}}$. Сравнить время работы двух вариантов поиска (brute force и хэши) в каждом из случаев. Результаты объяснить. Правда ли, что хэши всегда быстрее brute force?

# 6. Поиск подстроки в строке. Конечный детерминированный автомат

## 6.1. Вспомогательные примеры

Постановка задачи остается такой же, как и в предыдущем параграфе. Дана строка $S$ длины $n$ и строка $T$ длины $m$; символы строк нумеруются начиная с 1. Требуется найти все вхождения строки $T$ в строку $S$.

Как было указано в предыдущем параграфе, для решения одной и той же задачи, как правило, можно предложить несколько алгоритмов. В этом параграфе представлен алгоритм, который за счет однократной предварительной обработки шаблона $T$ потом осуществляет поиск за время $O(n)$, при этом каждый символ строки $S$ просматривается ровно один раз. Алгоритм оперирует такими понятиями, как «префикс-функция», «таблица переходов», «максимальный сдвиг». Прежде чем вводить эти формальные понятия, разберем несколько примеров; наблюдения, полученные в них, позволят естественным образом ввести указанные понятия[1].

Пусть $S = abxabcdaaabcdac$, $T = abcdac$. Введем следующие обозначения: $k_1$ — позиция в строке $S$, начиная с которой осуществляется поиск, т. е. сопоставление с шаблоном $T$; $k_2$ — позиция в строке $S$, с которой начинается/продолжается просмотр строки $S$; $k_3$ — позиция в строке $S$, до которой продвинулись, осуществляя просмотр.

Для удобства восприятия составим таблицу:

| a | b | x | a | b | c | d | a | a | a | b | c | d | a | c |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

1. $k_1 = 1$; $k_2 = 1$; $k_3 = 3$.
$S[3] = x$, символа $x$ в шаблоне $T$ нет, следовательно, поиск с позиций 2 или 3 не приведет к успеху. Сдвиг на 3.

2. $k_1 = 4$; $k_2 = 4$; $k_3 = 9$.
Внимательное рассмотрение фрагмента $S[4\,..\,9]$ показывает, что поиск с позиций 4,..., 8 не приведет к успеху. Например, поиск с 8-й позиции (9-ю букву строки $S$ мы уже просмотрели — это буква $a$) означал бы, что мы надеемся фрагмент $S[8\,..\,13] = a\,a\star\star\star\star$ (звездочками обозначены еще не просмотренные символы $S$) удачно сопоставить с шаблоном $a\,b\,c\,d\,a\,c$. Очевидно, что эта надежда обречена на провал.

От первого пункта ситуация, однако, отличается тем, что искомая подстрока может начинаться с последнего просмотренного символа — ведь это символ $a$! Именно с него начнется поиск в следующем пункте. Сдвиг на 5 (это не опечатка — на 5, а не на 6).

3. $k_1 = 9$; $k_2 = 10$; $k_3 = 10$.
Хотя просмотр и начинается с 10-й позиции, фактически поиск начался с 9-й позиции.

[1]Надо понимать, что классики Computer Science Кнут, Моррис, Пратт, Бойер, Мур и дугие пришли к своим «хитрым» алгоритмам в результате глубокого анализа частных (хотя и специально подобранных) примеров, и лишь затем закономерности, подмеченные ими, были обобщены, появился соответствующий понятийный аппарат и алгоритмы. Именно по этой причине в пособии выбран индуктивный подход — от частных примеров к общим методам.

Поясним, что эта фраза означает. В предыдущем пункте был обнаружен символ $a$, шаблон тоже начинался с символа $a$ ($S[9] = T[1]$). Теперь, просматривая 10-й символ строки $S$, мы будем сопоставлять $S[10]$ и $T[2]$. Обнаруживаем несоответствие $S[10] \neq T[2]$. В следующем пункте поиск начнется с символа $S[10]$. Сдвиг на 1.

4. $k_1 = 10$; $k_2 = 11$; $k_3 = 15$.

Так же как и в прошлый раз, первая буква найдена уже в предыдущем пункте. Начинаем сопоставление $S[11]$ и $T[2]$. Посимвольная проверка показывает, что обнаружено соответствие шаблону, $S[10\,..\,15] = T$.

Обратим внимание на то, что мы не делали откатов назад по строке $S$, как в алгоритме brute force. Остается несколько непонятным, как во втором пункте была выбрана величина сдвига — компьютер не умеет «внимательно рассматривать» строки. Именно на этапе предварительной обработки шаблона $T$, о которой упоминалось в самом начале параграфа, и определяется величина допустимого сдвига.

Пусть $S = abcdabcdabcc$, $T = abcdabcc$. Читатель наверняка заметил, что эти строки обладают большой степенью «самоподобия», т. е. $T[1\,..\,3] = T[5\,..\,7]$ и $S[1\,..\,4] = S[5\,..\,8] = T[1\,..\,4]$. Это обстоятельство существенно себя проявит во время поиска. В русском языке встречаются такие слова: *ананас*, *бабай*, *колокол*, *нанаец*.

Для удобства восприятия составим таблицу

| a | b | c | d | a | b | c | d | a | b | c | c |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

1. $k_1 = 1$; $k_2 = 1$; $k_3 = 8$.

На 8-м символе обнаружено несовпадение. Внимательный анализ показывает, что окончание $S[5\,..\,8]$ рассмотренного фрагмента $S[1\,..\,8]$ может служить началом вхождения подстроки $T$ в строку $S$, и следует сдинуться до 5-го символа в $S$. Сдвиг на 4. Попытки искать с 2, 3, 4 обречены на провал, а сдвиг более чем на 4 символа опасен тем, что можно пропустить вхождение. Действительно, $S[5\,..\,12] = a\,b\,c\,d\star\star\star\star$ (звездочками обозначены еще не просмотренные символы $S$) может (если повезет) совпасть с шаблоном $T = a\,b\,c\,d\,a\,b\,c\,c$.

2. $k_1 = 5$; $k_2 = 9$; $k_3 = 12$.

Первые 4 буквы уже найдены в предыдущем пункте. Начинаем сопоставление $S[9]$ и $T[5]$. Посимвольная проверка показывает, что обнаружено соответствие шаблону, $S[5\,..\,15] = T$.

Как и в первом примере главный вопрос — как алгоритмически определить величину сдвига? Как построить функцию, которая по просмотренной части строки $S$ и известному шаблону $T$ определит максимальный допустимый сдвиг? Эта задача решается в алгоритме Морриса–Пратта, где величина максимального допустимого сдвига находится за несколько итераций. Специальным образом построенный конечный детерминированный автомат за одно обращение к нему выдает величину максимального допустимого сдвига, однако он является требовательным к памяти.

## 6.2. Префикс-функция

**Определение 1** *Префиксом строки $S$ называется подстрока вида $S[1\,..\,i]$, $1 \leq i \leq n$, где $n$ — это длина строки $S$.*

**Определение 2** *Суффиксом строки $S$ называется подстрока вида $S[i\,..\,n]$, $1 \le i \le n$, где $n$ — это длина строки $S$.*

Например, для строки $Windows\ XP$ строки $W$ и $Win$ являются префиксами, а строка $ows\ XP$ — суффиксом. Интересна строка *колокол*, для которой трехбуквенный префикс *кол* одновременно является и трехбуквенным суффиксом[1]!

**Определение 3** *Префикс-функцией от строки $S$ называется $n$-мерный вектор, $i$-я компонента которого — это длина наибольшего префикса строки $S[1\,..\,i]$, который не совпадает с самой строкой $S[1\,..\,i]$ и одновременно является ее суффиксом. Если такого префикса не существует, то $i$-я компонента рана нулю. Здесь $n$ — это длина строки $S$.*

Префикс-функцию от строки $S$ будем обозначать $\pi(S)$, а $i$-ю ее компоненту — $\pi(S, i)$. Из определения следует, что для любой строки $S$ справедливо $\pi(S, 1) = 0$.

Найдем префикс-функцию для строки $S = abcadabccabca$.

$\pi(S, 1) = 0$. Для строки $S[1] = a$ не существует префикса, не совпадающего с самой строкой $S[1]$.

$\pi(S, 2) = 0$. Для строки $S[1\,..\,2] = ab$ не существует префикса, который не совпадает с самой строкой $S[1\,..\,2]$ и одновременно является ее суффиксом.

$\pi(S, 3) = 0$. Для строки $S[1\,..\,3] = abc$ не существует префикса, который не совпадает с самой строкой $S[1\,..\,3]$ и одновременно является ее суффиксом.

$\pi(S, 4) = 1$. Для строки $S[1\,..\,4] = abca$ существует префикс $a$, который не совпадает с самой строкой $S[1\,..\,4]$ и одновременно является ее суффиксом. Он является максимальным, длина этого префикса равна 1.

$\pi(S, 8) = 3$. Для строки $S[1\,..\,8] = abcadabc$ существует префикс $abc$, который не совпадает с самой строкой $S[1\,..\,8]$ и одновременно является ее суффиксом. Он является максимальным, длина этого префикса равна 3.

$\pi(S, 13) = 4$. Для строки $S[1\,..\,13] = abcadabccabca$ существует префикс $abca$, который не совпадает с самой строкой $S[1\,..\,13]$ и одновременно является ее суффиксом. Он является максимальным, длина этого префикса равна 4.

$\pi(S) = \{0\,0\,0\,1\,0\,1\,2\,3\,0\,1\,2\,3\,4\}$.

Найдем префикс-функцию для строки $S = abcabdxacabcabca$.

Для удобства восприятия составим таблицу:

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | a | b | c | a | b | d | x | a | c | a | b | c | a | b | c | a |
| $\pi(S, i)$ | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 3 | 1 |

Наиболее интересным является переход от $\pi(S, 14) = 5$ к $\pi(S, 15) = 3$ и от $\pi(S, 15) = 3$ к $\pi(S, 16) = 1$.

Можно предложить различные алгоритмы построения префикс-функции $\pi(S)$. Простейший (и наиболее неэффективный) состоит в том, что для каждого $i$, $1 \le i \le n$, последовательно перебираем все префиксы строки $S[1\,..\,i]$, не совпадающие с самой строкой $S[1\,..\,i]$, и проверяем, не являются ли они ее суффиксами. Сложность такого переборного алгоритма $O(n^2)$.

[1] Что именно тут интересного, очень скоро станет ясно.

Разберем алгоритм, строящий префикс-функцию за линейное от длины строки время. Находя значение $i$-й, $1 < i \le n$, компоненты префикс-функции, будем использовать ранее накопленные результаты, т. е. $\pi(S, j)$, $1 \le j < i$.

1. По определению $\pi(S, 0) = 0$.

2. Пусть найдено, что $\pi(S, i) = k$, $1 \le i < n$. Вычислим $\pi(S, i+1)$.

3. Если $S[i+1] = S[k+1]$, то $\pi(S, i+1) = k+1$, иначе пробуем меньшие суффиксы (разумеется, если эти меньшие суффиксы есть, т. е. если $k > 0$); разберем, как их искать. По определению префикс-функции для строки $S[1\,..\,k]$ префикс $S[1\,..\,\pi(S, k)]$ является суффиксом. Любой суффикс строки $S[1\,..\,k]$ является суффиксом строки $S[1\,..\,i]$, следовательно, $S[1\,..\,\pi(S, k)]$ является суффиксом $S[1\,..\,i]$. Кроме того, для любого $j$, $k < j < i$, строка $S[1\,..\,j]$ не является суффиксом $S[1\,..\,i]$.

Таким образом, получаем следующий алгоритм нахождения $\pi(S, i+1)$.

1. При $S[i+1] = S[k+1]$ положить $\pi(S, i+1) = k+1$.

2. Иначе при $k = 0$ положить $\pi(S, i+1) = 0$.

3. Иначе установить $k = \pi(S, k)$, GOTO 1.

Несмотря на то, что пункт 3 представляет собой внутренний цикл, время вычисления префикс-функции оценивается в $O(n)$.

Представим программу, которая строит префикс-функцию. С учетом того, что в JScript строки нумеруются с нуля, а мы исходили из предположения, что строки нумеруются с единицы, нужно произвести соответствующие изменения в индексации строк.

```
t=WScript.StdIn.ReadLine()
//t='qqwqqwerqqwqqwetqqwqqwerqqwqqwerqqwqqq'
m=t.length
pi=new Array(m)
pi[0]=0
k=0

for(i=1;i<m;i++) {
    while((k>0)&&(t.charAt(k)!=t.charAt(i)))
        k=pi[k-1];
    if(t.charAt(k)==t.charAt(i))
        k++
    pi[i]=k
}
WScript.Echo(pi)
```

Во второй строке в комментарии приведен хороший тестовый пример.

## 6.3. Алгоритм Морриса–Пратта

Вернемся к задаче поиска подстроки $T$ в строке $S$. Пусть поиск начинается с $i$-й позиции строки $S$, $1 \le i \le n-m+1$, т. е. мы сопоставляем $T[1]$ и $S[i]$. Пусть первое несоответствие произошло на $(k+1)$-й, $1 < k+1 \le m$, позиции шаблона $T$, тогда $S[i\,..\,i+k-1] =$

$T[1\,..\,k]$. При сдвиге шаблона $T$ можно ожидать, что префикс фрагмента $T[1\,..\,k]$ совпадет с соответствующим суффиксом фрагмента $S[i\,..\,i+k-1]$. Тогда сдвинуться можно будет не на один символ, как в brute force, а на $k - \pi(T,k)$ и продолжить сопоставление с той позиции, на которой было обнаружено рассогласование.

Рассмотрим пример поиска подстроки $T = abaxabaz$ в длинной строке $S$. На схеме будет приведен лишь ее фрагмент $S[100\,..\,113]$.

1. Построим префикс-функцию для $T$.

| $j$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $T$ | a | b | a | x | a | b | a | z |
| $\pi(T,j)$ | 0 | 0 | 1 | 0 | 1 | 2 | 3 | 0 |

2. Пусть поиск начался с 100-й позиции строки $S$, несоответствие произошло на 8-м символе $T$, таким образом, $i = 100$, $k = 7$.

| | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | a | b | a | x | *a* | *b* | *a* | **b** | a | x | a | b | a | z |
| T | *a* | *b* | *a* | x | a | b | a | **z** | | | | | | |
| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | | | | | |

Префикс фрагмента $T[1\,..\,7]$ совпадает с суффиксом фрагмента $S[100\,..\,106]$, а именно $T[1\,..\,3] = S[104\,..\,106]$ (в таблице выделено курсивом). Надо сдвинуть шаблон на 4 позиции.

Теперь мы готовы ответить на вопрос об автоматическом выборе величины сдвига! Величина сдвига равна $k - \pi(T,k)$; в данном случае $k - \pi(T,k) = 7 - 3$, т. е. 4 позиции.

Делаем новую попытку сопоставления; знаками вопроса в таблице обозначены те символы шаблона $T$, которые даже не сопоставлялись со строкой $S$ — они не важны для наших рассуждений. Поиск начался со 104-й позиции строки $S$ (в прошлый раз начинали с 100-й и на 4 сдвинулись), несоответствие произошло на 4-м символе $T$, таким образом, $i = 104$, $k = 3$. Важно отметить, что сопоставление $S$ и $T$ началось со сравнения $S[107]$ и $T[4]$; то, что $T[1\,..\,3] = S[104\,..\,106]$, известно с предыдущего шага.

| | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | a | b | a | x | a | b | *a* | **b** | a | x | a | b | a | z |
| T | | | | | *a* | b | a | **x** | ? | ? | ? | ? | | |
| j | | | | | 1 | 2 | 3 | 4 | 5 | | | | | |

Не задумываясь над тем, что префикс фрагмента $T[1\,..\,3]$ совпадает с суффиксом фрагмента $S[104\,..\,106]$, сразу же определяем величину сдвига по формуле $k - \pi(T,k)$. Имеем $3 - \pi(T,3) = 3 - 1 = 2$.

Делаем новую попытку сопоставления. Поиск начался со 106-й позиции строки $S$, таким образом, $i = 106$. Сопоставление $S$ и $T$ началось со сравнения $S[107]$ и $T[2]$; то, что $T[1] = S[106]$, известно с предыдущего шага. Посимвольная проверка показывает совпадение.

| | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | a | b | a | x | a | b | a | b | a | x | a | b | a | z |
| T | | | | | | | a | b | a | x | a | b | a | z |
| j | | | | | | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

## 6.4. Построение и использование конечного детерминированного автомата

Конечным детерминированным автоматом $A$ называется пятерка $< Q, \Sigma, q_0, F, \delta >$, где
$Q$ — конечное множество состояний автомата;
$\Sigma$ — конечный алфавит, т. е. множество букв, которое может читать автомат;
$q_0$ — начальное состояние автомата, $q_0 \in Q$;
$F$ — множество терминальных состояний $F \subseteq Q$;
$\delta$ — функция перехода, т. е. отображение вида $Q \times \Sigma \to Q$.

Полезной является графовая интерпретация автомата. Состояния изображаются кругами — это вершины графа. Вершины соединены дугами; на дугах пишутся буквы в соответствии с функцией перехода $\delta$. Именно вершины $q_i$, $q_j \in Q$ соединены дугой, ведущей от $q_i$ к $q_j$, а на дуге приписана буква $\alpha \in \Sigma$, если $\delta(q_i, \alpha) = q_j$.

Рассмотрим пример.

Здесь множество состояний $Q = \{0, 1, 2, 3\}$, алфавит $\Sigma = \{a, b, c, d\}$, начальное состояние $q_0 = 0$, множество терминальных состояний состоит из двух элементов $F = \{2, 3\}$, функция перехода $\delta(0, a) = 1$, $\delta(0, b) = 2$, $\delta(1, a) = 3$, $\delta(1, b) = 1$, $\delta(1, d) = 0$, $\delta(2, c) = 1$.

Имея строку $T$ и зная алфавит строки $S$[1], можно построить специальный конечный детерминированный автомат, который способен распознать вхождение подстроки $T$ в строку $S$. Состояниями $Q$ будут всевозможные префиксы строки $T$ и состояние $\lambda$, которое соответствует пустому префиксу — пустой строке. Алфавит $\Sigma$ состоит из объединения алфавитов строк $T$ и $S$. Начальным состоянием является $\lambda$, терминальным — состояние, соответствующее префиксу, совпадающему со всей строкой $T$. Переход из состояния $q \in Q$ по букве $\alpha \in \Sigma$ осуществляется по правилу $\forall q \in Q \quad \forall \alpha \in \Sigma \quad \delta(q, \alpha) = d$, где $d$ — максимальный суффикс строки $q\alpha$, являющийся префиксом строки $T$. В частности, возможно, что $d = q\alpha$.

Рассмотрим пример автомата, распознающего вхождение подстроки $T = abc$ в длинную строку $S$. Пусть априори известно, что алфавит строки $S$ состоит из символов $a, b, c, d$.

Состояния автомата $Q = \{\lambda, a, ab, abc\}$.
Алфавит $\Sigma = \{a, b, c, d\}$.
Начальное состояние $\lambda$.
Терминальное состояние $abc$.
Опишем функцию переходов.

| | | | | |
|---|---|---|---|---|
| Пусть $q = \lambda$, | имеем $\delta(\lambda, a) = a$, | $\delta(\lambda, b) = \lambda$, | $\delta(\lambda, c) = \lambda$, | $\delta(\lambda, d) = \lambda$. |
| Пусть $q = a$, | имеем $\delta(a, a) = a$, | $\delta(a, b) = ab$, | $\delta(a, c) = \lambda$, | $\delta(a, d) = \lambda$. |
| Пусть $q = ab$, | имеем $\delta(ab, a) = a$, | $\delta(ab, b) = \lambda$, | $\delta(ab, c) = abc$, | $\delta(ab, d) = \lambda$. |
| Пусть $q = abc$, | имеем $\delta(abc, a) = a$, | $\delta(abc, b) = \lambda$, | $\delta(abc, c) = \lambda$, | $\delta(abc, d) = \lambda$. |

Разберем, как происходит поиск. Автомат находится в состоянии $\lambda$, на вход подается строка $S$. Читая последовательно буквы строки $S$, автомат осуществляет переходы из состояния в состояние. Как только автомат оказывается в терминальном состоянии, мы

[1] Алфавит строки $S$ нужен лишь для формального определения автомата; для программной реализации он не требуется.

констатируем, что только что была обнаружена подстрока $T$ в строке $S$. Точнее, если по прочтении $i$-го символа строки $S$ автомат оказался в терминальном состоянии, то фрагмент $S[i-m+1\,..\,i]$ — искомый; в файл вывода записываем номер позиции $i-m+1$ и продолжаем поиск.

Например, для $S = ababcadabcc$ поиск пройдет так:

| Находился | Прочитал | Перешел | Нашел вхождение |
|---|---|---|---|
| $\lambda$ | $S[1] = a$ | $a$ | |
| $a$ | $S[2] = b$ | $ab$ | |
| $ab$ | $S[3] = a$ | $a$ | |
| $a$ | $S[4] = b$ | $ab$ | |
| $ab$ | $S[5] = c$ | $abc$ | Да, в 5–3+1=3 позиции |
| $abc$ | $S[6] = a$ | $a$ | |
| $a$ | $S[7] = d$ | $\lambda$ | |
| $\lambda$ | $S[8] = a$ | $a$ | |
| $a$ | $S[9] = b$ | $ab$ | |
| $ab$ | $S[10] = c$ | $abc$ | Да, в 10–3+1=8 позиции |
| $abc$ | $S[11] = c$ | $\lambda$ | |

Обратим внимание на такую особенность автомата, что он каждый символ строки $S$ просматривает *ровно один раз* и ничего не сравнивает с символами шаблона $T$ (строку $T$ мы используем только на этапе построения автомата, а на этапе поиска используем только функцию переходов $\delta$). Автомат не делает никаких откатов, возвратов, прыжков по строке $S$, как алгоритм brute force. Более того, автомат не обращается к одному символу строки $S$ дважды (ср. с алгоритмом Морриса–Пратта, который обратился к символу $S[107]$ трижды, когда искал $abaxabaz$, см. разд. 6.3). Автомат можно сравнить с «черным ящиком», который «пожирает» символы строки $S$, переключает свои внутренние состояния и в определенные моменты (когда оказывается в терминальном состоянии) сигнализирует об обнаружении подстроки.

Итак, мы видим, что конечные детерминированные автоматы — удобное и мощное средство для решения задачи поиска подстроки в строке; однако для того, чтобы применить их на практике, необходимо приведенные выше формальные определения и конструкции воплотить в конкретные структуры данных и четко прописать правила их использования. Действительно, графовая интерпретация удобна для человека, но не для компьютера.

Автомат, распознающий вхождение подстроки в строку, можно хранить в виде таблицы переходов — двумерного массива. Для удобства (и краткости записи) будем обозначать состояния числами от 0 до $m$, именно состояние $i$, $1 \le i \le m$, обозначает состояние, соответствующее префиксу $T[1\,..\,i]$, а состояние 0 обозначает $\lambda$. Специальным символом * будем обозначать любой символ строки $S$, не принадлежащий строке $T$. Использование символа * позволяет заполнить таблицу переходов без априорного знания алфавита строки $S$ (см. сноску на предыдущей странице) и, кроме того, зачастую сократить объем таблицы переходов. Например, для шаблона $T = abc$ имеем

| | $a$ | $b$ | $c$ | $*$ |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 2 | 0 | 0 |
| 2 | 1 | 0 | 3 | 0 |
| 3 | 1 | 0 | 0 | 0 |

Для систематизации и закрепления полученных знаний построим автомат, распознающий вхождение подстроки $T = ananas$ в строку $S$.

Состояния автомата $Q = \{\lambda, a, an, ana, anan, anana, ananas\} = \{0, 1, 2, 3, 4, 5, 6\}$.
Алфавит $\Sigma = \{a, n, s, *\}$.
Начальное состояние $\lambda$.
Терминальное состояние $ananas$ или в числовом виде 6.
Опишем функцию переходов. Внимание надо обратить на переходы из 5-го состояния.

| | $a$ | $n$ | $s$ | $*$ |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 2 | 0 | 0 |
| 2 | 3 | 0 | 0 | 0 |
| 3 | 1 | 4 | 0 | 0 |
| 4 | 5 | 0 | 0 | 0 |
| 5 | 1 | 4 | 6 | 0 |
| 6 | 1 | 0 | 0 | 0 |

Попытаемся найти слово $T = ananas$ в строке $S = anananaananas$.

| Находился | Прочитал | Перешел | Исследуемый фрагмент $S$ |
|---|---|---|---|
| 0 | $S[1] = a$ | 1 | **a***nananaananas* |
| 1 | $S[2] = n$ | 2 | **an***ananaananas* |
| 2 | $S[3] = a$ | 3 | **ana***nanaananas* |
| 3 | $S[4] = n$ | 4 | **anan***anaananas* |
| 4 | $S[5] = a$ | 5 | **anana***naananas* |
| **5** | $S[6] = n$ | **4** | *an***anan***aananas* |
| 4 | $S[7] = a$ | 5 | *an***anana***ananas* |
| **5** | $S[8] = a$ | **1** | *anana***a***nanas* |
| 1 | $S[9] = n$ | 2 | *anana***an***anas* |
| 2 | $S[10] = a$ | 3 | *anana***ana***nas* |
| 3 | $S[11] = n$ | 4 | *anana***anan***as* |
| 4 | $S[12] = a$ | 5 | *anana***anana***s* |
| **5** | $S[13] = s$ | **6** | *anana***ananas** |

Стартуем в состоянии 0 и, читая последовательно буквы строки $S$, пытаемся найти $ananas$. До $S[5]$ все шло как нельзя лучше — мы нашли $anana$. Находясь в состоянии 5, мы встретили $S[6] = n$. Автомат велит переходить в состояние 4; кстати, не случайно в просмотренном фрагменте $S[1\,..\,6] = ananan$ **четырех**буквенный суффикс $anan$ являлся префиксом шаблона $T$.

Прочитав $S[7] = a$, автомат переходит в состояние 5, т. е. «5 букв прочитано».

Находясь в состоянии 5, мы встретили $S[8] = a$. Автомат велит переходить в состояние 1; кстати, не случайно в просмотренном фрагменте $S[4\,..\,8] = ananaa$ **одно**буквенный суффикс $a$ являлся префиксом шаблона $T$.

Прочитав $S[13]$, автомат оказался в состоянии 6. Автомат сигнализирует, что нашел вхождение 8-ой позиции строки $S$.

Приведем фрагмент кода, который позволяет эффективно строить таблицу переходов автомата.

```
t=WScript.StdIn.ReadLine()
//t='abaxabaz'
m=t.length
alph=new Array()
//Определяем алфавит строки t
for(i=0;i<m;i++)
    alph[t.charAt(i)]=0
//В двумерном массиве del будем хранить таблицу переходов
del=new Array(m+1)
for(j=0;j<=m;j++)
    del[j]=new Array()
//Инициализируем таблицу переходов
for(i in alph)
    del[0][i]=0
//Формируем таблицу переходов
for(j=0;j<m;j++){
    prev=del[j][t.charAt(j)]
    del[j][t.charAt(j)]=j+1
    for(i in alph)
        del[j+1][i]=del[prev][i]
}
//Выводим таблицу переходов
for(j=0;j<=m;j++){
    out=''
    for(i in alph)
        out+=del[j][i]+' '
    WScript.Echo(out)
}
```

Во второй строке в комментарии приведен хороший тестовый пример.

## 6.5. Задания для самостоятельной работы

1. Какова вычислительная сложность поиска подстроки длиной $m$ в строке длиной $n$ с использованием конечного детерминированного автомата?

2. Постройте конечный детерминированный автомат для поиска подстроки «колокол» в строке «колокольчик и колокол». Проиллюстрируйте по шагам работу алгоритма.

# 7. Поиск подстроки в строке. Алгоритм Бойера–Мура

## 7.1. Идея алгоритма

Специфика алгоритма Бойера–Мура состоит в том, что за счет предварительной обработки шаблона $T$ и специальным образом организованного процесса сопоставления шаблона и фрагмента строки $S$ удается пропустить некоторые символы (а иногда и весьма длинные последовательности символов[1]) строки $S$ как заведомо не дающие совпадения с шаблоном $T$.

На очередном шаге алгоритма пытаемся сопоставить фрагмент $S[i\,..\,i+m-1]$, $1 \le i \le n-m+1$, с шаблоном $T$. В частности, на первом шаге пытаемся сопоставить $S[1\,..\,m]$ и $T$. Сопоставление начнем с конца фрагмента $S[i\,..\,i+m-1]$ и будем двигаться к его началу. Причина в том, что сопоставление с конца дает в отдельных случаях больше информации, нежели если делать сравнение с начала (т. е. с левого конца), и позволяет совершать «длинные прыжки» по строке $S$.

## 7.2. Эвристика «плохого символа»

Сделаем сначала два простых наблюдения, а потом перейдем к разбору общих правил, лежащих в основе алгоритма Бойера–Мура. Обозначим через *char* последний символ сопоставляемого фрагмента $S[i\,..\,i+m-1]$, т. е. $char = S[i+m-1]$.

Если известно, что *char* не принадлежит $T$, то вхождение подстроки $T$ в строку $S$ начиная с позиций $i$, $i+1$, $...\,i+m-1$ невозможно — иначе это потребовало бы, чтобы *char* встретился в $T$. Таким образом, можно сдвинуть шаблон $T$ на $m$ символов вправо. В таблице приведен пример; в первой строке — номера позиций в строке $S$, во второй сама строка $S$, в третьей — расположение шаблона до сдвига, в последней — расположение шаблона до сдвига.

| $i$ | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | $\star$ | $\star$ | $\star$ | $\star$ | $\star$ | f | $\star$ | $\star$ | $\star$ | $\star$ | $\star$ | $\star$ |
| $T$ | a | b | a | c | d | b | | | | | | |
| $T$ | | | | | | | a | b | a | c | d | b |

Обобщение этого наблюдения состоит в том, что если $char \ne T[m]$, но $char \in T$, то можно сдвинуть шаблон $T$ так, чтобы сопоставить последнее (т. е. наиболее правое) вхождение *char* в $T$ с $S[i+m-1]$. Пусть последнее (т. е. наиболее правое) вхождение *char* в $T$ произошло на $N(char;\,T)$ позиции, тогда сдвинуть шаблон можно на $m - N(char;\,T)$ позиций вправо.

Разберем пример подробно: $char = S[i+m-1] = S[100+6-1] = S[105] = a$; последнее вхождение $a$ в $T$ произошло на $N(char;T) = 3$ позиции; сдвинуть шаблон можно на $m - N(char;T) = 6-3 = 3$ позиции вправо.

[1] Длина такой последовательности не более $m$.

| $i$ | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 |
|---|---|---|---|---|---|---|---|---|---|
| $S$ | $\star$ | $\star$ | $\star$ | $\star$ | $\star$ | **a** | $\star$ | $\star$ | $\star$ |
| $T$ | a | b | **a** | c | d | b | | | |
| $T$ | | | | a | b | **a** | c | d | b |

Таблица $N$ наиболее правых вхождений символов в строку $T$ заполняется элементарно:

| *char* | a | b | c | d |
|---|---|---|---|---|
| $N(char;T)$ | 3 | 6 | 4 | 5 |

```
N=new Array()
for(j=0;j<m;j++)
   N[T.charAt(j)]=j+1
for(j in N)
   WScript.Echo('N[',j,']=',N[j])
```

Хотя нами принята договоренность о нумерации строк с единицы, в этом фрагменте кода мы, разумеется, нумеруем строки с нуля (как это реализовано в JScript).

Ранее мы выдвигали требования «известно, что *char* принадлежит строке $T$» или «известно, что *char* не принадлежит строке $T$». Как эффективно проверить их выполнение? Действительно, если каждый раз, отвечая на этот вопрос, просматривать весь шаблон $T$, то процесс поиска будет долгим и неэффективным. Используя таблицу $N$, можно дать ответ.

```
if(N[char]==undefined)
   WScript.Echo(char,'не входит в строку T. Сдвиг на ',m)
else
   WScript.Echo(char,'входит в строку T. Сдвиг на ',m-N[char])
```

Перейдем к формулировке и описанию первого правила.
Пусть $l$ последних символов совпали, т. е. $S[i+m-l\,..\,i+m-1] = T[m-l+1\,..\,m]$, $0 \le l < m$, расхождение случилось на $(l+1)$-м с правого конца символе[1], т. е. $S[i+m-l-1] \neq T[m-l]$. Необходимо определить, на сколько сдвинуть шаблон $T$, чтобы начать новое сопоставление.

Переопределим *char* — это самый правый символ фрагмента $S[i\,..\,i+m-1]$, на котором произошло рассогласование, т. е. $char = S[i+m-l-1]$.
Если $char\neg \in T$, то надо сдвинуть шаблон $T$ за $(i+m-l-1)-$ю позицию и начать новое сопоставление. Таким образом, величина сдвига $m-l$.
Если $char \in T$ и *char* не встречается в «просмотренном хвосте», т. е. $char\neg \in T[m-l+1\,..\,m]$, то надо сдвинуть шаблон $T$ так, чтобы сопоставить последнее (т. е. наиболее правое) вхождение *char* в $T$ с $S[i+m-l-1]$. Таким образом, величина сдвига $m-N(char;T)-l$. Например, для $T = bcbabba$ и длинной строки $S$, фрагмент которой приведен в таблице, имеем

[1]Если $l=0$, т. е. совпавших символов нет, то $S[i+m\,..\,i+m-1]$ и $T[m+1\,..\,m]$ есть пустые строки.

| $i$ | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 |
|---|---|---|---|---|---|---|---|---|---|
| $S$ | $\star$ | $\star$ | $\star$ | **c** | b | b | a | $\star$ | $\star$ |
| $T$ | b | **c** | b | a | b | b | a | | |
| $T$ | | | b | **c** | b | a | b | b | a |

$l = 3$ — три совпавших символа $S[103\,..\,105] = T[5\,..\,7] = bba$;

$N(c; T) = 2$ — символ $c$ последний раз встретился на второй позиции шаблона;

$m = 7$ — длина шаблона $T$;

Сдвиг на $m - N(char; T) - l = 7 - 2 - 3 = 2$.

Возникает вопрос, как понять, что $char$ не встречается в «просмотренном хвосте» и что делать, если все-таки он там встретился. Снова таблица $N$ нам поможет дать ответ: если $char \in T[m - l + 1\,..\,m]$, то $m - N(char;\, T) - l < 0$.

Сдвиг на отрицательную величину означает смещение шаблона $T$ влево; такие сдвиги допускать нельзя. В своей оригинальной статье Бойер и Мур рекомендуют в этом «неприятном» случае делать сдвиг на одну позицию вправо[1]. Например, для $T = baadccba$ и длинной строки $S$, фрагмент которой приведен в таблице, имеем

| $i$ | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 |
|---|---|---|---|---|---|---|---|---|---|
| $S$ | $\star$ | $\star$ | $\star$ | **b** | c | c | b | a | $\star$ |
| $T$ | b | a | a | d | c | c | **b** | a | |

$l = 4$ — четыре совпавших символа $S[104\,..\,107] = T[5\,..\,5] = ccba$;

$N(b; T) = 7$ — символ $b$ последний раз встретился на седьмой позиции шаблона;

$m = 8$ — длина шаблона $T$;

Величина $m - N(char;\, T) - l = 8 - 7 - 4 = -3 < 0$. Сдвигать на отрицательную величину (т. е. влево) нельзя. Формально сдвигаем на одну позицию вправо.

Все готово, чтобы сформулировать **первое правило** (эвристика плохого символа). Пусть $l$ последних символов совпали, т. е. $S[i+m-l\,..\,i+m-1] = T[m-l+1\,..\,m]$, $0 \le l < m$, расхождение случилось на $(l+1)$-м с правого конца символе, т. е. $S[i+m-l-1] \ne T[m-l]$, плохой символ определен следующим образом: $char = S[i\,..\,i+m-1]$,
таблица самых правых вхождений $N$ определена следующим образом:

1) $\forall x \in T \;\; N(x; T) = \max\limits_{1 \le k \le m} k : (T(k) = x)\ \&\ (x\neg \in T[k+1\,..\,m])$;

2) $\forall x\neg \in T \; N(x; T) = 0$,

тогда величина сдвига рассчитывается так $Shift_1(char) = \max\Big\{m - N(char; T) - l; 1\Big\}$.

Пусть совпали $m$ символов, т. е. обнаружено вхождение подстроки $T$ в строку $S$, тогда $Shift_1 = 1$.

## 7.3. Эвристика «хорошего суффикса»

Прежде чем сформулировать второе правило, рассмотрим три поясняющих примера.

[1]При этом, забегая вперед, они сразу же пишут, что существует второе правило, которое гарантирует в таких «неприятных» случаях сдвиг не менее чем на единицу. Мы сформулируем это правило чуть ниже. Таким образом, сдвиг на единицу в «неприятном» случае — это временная формальность, нужная лишь для того, чтобы придать первому правилу завершенный вид.

Пример 1. Строка $T = bdcabc$ ищется в длинной строке $S$, фрагмент которой приведен в таблице.

| $i$ | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | $\star$ | $\star$ | $\star$ | d | **b** | **c** | $\star$ | $\star$ | $\star$ | $\star$ | $\star$ | $\star$ |
| $T$ | b | d | c | a | **b** | **c** | | | | | | |
| $T$ плохой вариант | | | b | d | c | a | b | c | | | | |
| $T$ хороший вариант | | | | | | | b | d | c | a | b | c |

При сопоставлении шаблона $T$ и фрагмента $S$ совпали два последних символа — «хороший суффикс» $bc$; расхождение случилось на символе $d$. Если пользоваться эвристикой плохого символа, то нужно сделать сдвиг на 2 позиции — подвести $T[2] = d$ под $S[103] = d$ (в таблице строка четвертая). Очевидно, правда, что при этом под суффикс $S[104\,..\,105] = bc$ будет подведен фрагмент $T[3\,..\,4] = ca$; надеяться, что после такого сдвига мы обнаружим вхождение подстроки в строку бессмысленно.

Было бы неплохо, выбирая величину сдвига, руководствоваться не только значением плохого символа $S[103]$, но и обращать внимание на всю открывшуюся информацию, т. е. учитывать еще и значение суффикса. Поскольку в $T$ нет ничего, что могло бы быть подведено под $S[104\,..\,105]$, надо смещаться за 105-ю позицию. Сдвиг на 6, т. е. на длину шаблона. В таблице строка пятая.

Пример 2. Строка $T = bdbdabd$ ищется в длинной строке $S$, фрагмент которой приведен в таблице.

| $i$ | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 |
|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | $\star$ | $\star$ | $\star$ | $\star$ | d | **b** | **d** | $\star$ | $\star$ | $\star$ |
| $T$ | b | d | **b** | **d** | a | b | d | | | |
| $T$ | | | | b | d | **b** | **d** | a | b | d |

При сопоставлении шаблона $T$ и фрагмента $S$ совпали два последних символа — «хороший суффикс» $bd$; расхождение случилось на символе $S[104] = d$. Здесь эвристика плохого символа работает особенно плохо, так как d уже встречался в «просмотренном хвосте» шаблона $T$; соответственно, первое правило предлагает сделать сдвиг на 1 позицию вправо. Однако можно предложить лучший вариант! Попытаемся подвести под хороший суффикс $bd$ соответствующий фрагмент шаблона $T$. В поисках этого фрагмента будем сканировать шаблон справа налево — найдем $T[3\,..\,4] = bd$. Сделаем сдвиг на 3 позиции вправо.

Отметим очень важное обстоятельство: именно из-за символа $T[5] = a$ произошло рассогласование — $S[104] \neq a$. Найденный фрагмент $T[3\,..\,4]$ предворяется символом, отличным от $a$ (то, что это именно $d$ абсолютно не важно); это дает основание надеяться, что совпадение произойдет. Если бы $T$ был равен $babdabd$, то сдвиг на 3 позиции вправо (подведение $T[3\,..\,4]$ под $S[105\,..\,106]$) гарантированно привел бы к несоответствию — по крайней мере $S[104] = d$ не совпал бы с $T[2] = a$; мы бы «наступили на те же грабли».

Пример 3. Строка $T = bcbcdbc$ ищется в длинной строке $S$, фрагмент которой приведен в таблице.

| $i$ | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | $\star$ | $\star$ | $\star$ | d | d | **b** | **c** | $\star$ | $\star$ | $\star$ | $\star$ | $\star$ |
| $T$ | **b** | **c** | b | c | d | b | c | | | | | |
| $T$ | | | | | | **b** | **c** | b | c | d | b | c |

Хотя нельзя найти в $T$ фрагмент, полностью совпадающий с суффиксом *dbc*, можно надеяться, что сдвиг шаблона на 4 позиции вправо приведет к успеху. Тем самым мы сопоставляем «хвост» (а именно *bc*) суффикса *dbc* и начало шаблона $T$.

Эти примеры очень сильно отличаются, несмотря на то, что имеют общую природу; в каждом из них проявляются своеобразные особенности. Чтобы формально описать общее правило нахождения фрагмента шаблона $T$, соответствующего хорошему суффиксу, а следовательно, рассчитать величину сдвига, нам потребуется специальная конструкция $T^*$ — расширение шаблона $T$, полученное путем приписывания слева специальных символов *.

Рассмотрим $T^* = \underbrace{** \ \ldots *}_{m \text{ раз}} T$ т.е. $T^*[j] = *,\ -m+1 \le j \le 0$, и $T^*[j] = T[j]\ , 1 \le j \le m$;

символ $*$ интерпретируем как символ равный произвольному символу.

Например, для строки $T = bacaca$ получим $T^* = ******bacaca$. Имеют место следующие равенства: $T^*[6] = T^*[4],\ T^*[4\,..\,6] = T^*[2\,..\,4],\ T^*[3\,..\,6] = T^*[-3\,..\,0]$.

Для строки $T = bcabbc$ получим $T^* = ******bcabbc$. Имеют место следующие равенства: $T^*[6] = T^*[2],\ T^*[5\,..\,6] = T^*[1\,..\,2],\ T^*[4\,..\,6] = T^*[0\,..\,2],\ T^*[3\,..\,6] = T^*[-1\,..\,2]$.

Все готово, чтобы сформулировать **второе правило** (эвристика хорошего суффикса). Пусть при сопоставлении фрагмента $S[i\,..\,i+m-1]$ с $T$ совпали $l$ последних символов[1], т. е. $S[i+m-l\,..\,i+m-1] = T[m-l+1\,..\,m]$, $0 \le l \le m$ (при этом либо расхождение случилось на $(l+1)$-м с правого конца символе, либо $l = m$). Величина сдвига определяется следующим образом: $Shift_2(l) = m - rpr(l;T) - l + 1$, где $rpr(l;T) = \max k$ :

1) $k \le m - l$;

2) $T^*[k\,..\,k+l-1] = T[m-l+1\,..\,m]$;

3) либо $k > 1$ & $T[k-1] \ne T[m-l]$, либо $k \le 1$.

Величина $rpr(l;T)$ (англ. rightmost plausible reoccurrence, дословный перевод — наиправейшее возможное перепоявление), по сути, есть номер позиции в расширении шаблона $T^*$.

Хотя решается задача поиска подстроки $T$ в строке $S$, чтобы заполнить $rpr(l;T)$ и $Shift_2(l)$, достаточно обработать только $T$. Сначала научимся строить $rpr(l;T)$, потом поймем, как это применять для поиска подстроки $T$ в строке $S$.

Пример 1. Пусть $T = abcdabc$, построить таблицу для $rpr(l;T)$ и $Shift_2(l)$. Используя эвристику хорошего суффикса, проделать поиск подстроки $T$ в строке $S = abccabcbbccabcdabcdabc$.

| $l$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $rpr(l)$ | 7 | 0 | –1 | 1 | 0 | –1 | –2 |
| $Shift_2(l)$ | 1 | 7 | 7 | 4 | 4 | 4 | 4 |

Первым делом напишем $T^* = *******abcdabc$.

Пусть $l = 0$. Необходимо найти максимальное $k$, такое, что

1) $k \le m$;

[1]Обратите внимание, что возможно совпадение всех $m$ символов. При формулировке правила плохого символа этот случай рассматривался отдельно.

2) $T^*[k\,..\,k-1] = T[m+1\,..\,m]$;

3) либо $k > 1$ & $T[k-1] \neq T[m]$, либо $k \leq 1$.

Очевидно, что строки $T^*[k\,..\,k-1]$ и $T[m+1\,..\,m]$ являются пустыми, поэтому на второй пункт внимания не обращаем. Ясно, что $k = 7$. Действительно, $T[7-1] = b \neq T[m] = c$.

Пусть $l = 1$. Необходимо найти максимальное $k$, такое, что

1) $k \leq m-1$;

2) $T^*[k] = T[m]$

3) либо $k > 1$ & $T[k-1] \neq T[m-1]$, либо $k \leq 1$.

Типичной ошибкой является утверждение, что $k = 3$; пункт 2 будет выполнен ($T^*[3] = T[7]$), но пункт 3 нарушен ($T[3-1] = b = T[7-1]$). На самом деле, $k = 0$. Действительно, пункт 2 будет выполнен ($T[0] = * = T[7] = c$) и пункт 3 будет выполнен ($k \leq 1$).

Пусть $l = 2$. Необходимо найти максимальное $k$, такое, что

1) $k \leq m-2$;

2) $T^*[k\,..\,k+1] = T[m-1\,..\,m]$;

3) либо $k > 1$ & $T[k-1] \neq T[m-2]$, либо $k \leq 1$.

Ситуация полностью аналогична предыдущему случаю. Здесь $k = -1$. Действительно, пункт 2 будет выполнен ($T[-1\,..\,0] = ** = T[6\,..\,7] = bc$) и пункт 3 будет выполнен ($k \leq 1$).

Пусть $l = 3$. Необходимо найти максимальное $k$, такое, что

1) $k \leq m-3$;

2) $T^*[k\,..\,k+2] = T[m-2\,..\,m]$;

3) либо $k > 1$ & $T[k-1] \neq T[m-3]$, либо $k \leq 1$. Здесь $k = 1$. Действительно, пункт 2 будет выполнен ($T[1\,..\,3] = abc = T[5\,..\,7]$) и пункт 3 будет выполнен ($k \leq 1$).

Пусть $l = 4$. Необходимо найти максимальное $k$, такое, что

1) $k \leq m-4$;

2) $T^*[k\,..\,k+3] = T[m-3\,..\,m]$;

3) либо $k > 1$ & $T[k-1] \neq T[m-4]$, либо $k \leq 1$.

Ситуация полностью аналогична предыдущему случаю. Здесь $k = 0$. Действительно, пункт 2 будет выполнен ($T[0\,..\,3] = *abc = T[5\,..\,7] = dabc$) и пункт 3 будет выполнен ($k \leq 1$).

Для $l \in \{5, 6, 7\}$ аналогично. Заметим, что для $l = 7$ пункт 1 проявляет себя как существенное ограничение, во всех остальных случаях он выполнялся автоматически.

Теперь переходим к задаче поиска. Процесс поиска показан на схеме. Здесь цифра рядом с $T$ — это номер попытки; первые две неудачные, а третья и четвертая показывают соответствие.

```
Номер 1 2 3 4 5 6 7 8 9 10  12  14  16  18  20  22
  S = a b c c a b c b b c c a b c d a b c d a b c
  T1= a b c d a b c
  T2=         a b c d a b c
  T3=                       a b c d a b c
  T4=                               a b c d a b c
```

Для $i = 1$ имеем $S[5\,..\,7] = T[5\,..\,7]$, $S[4] \neq T[4]$, следовательно, $l = 3$. Сдвиг смотрим в таблице — он равен 4.

Для $i = 5$ имеем $S[10] = T[7]$, $S[9] \neq T[6]$, следовательно, $l = 1$. Сдвиг равен 7.

Для $i = 12$ имеем $S[12\,..\,18] = T$, следовательно, $l = 7$. Сдвиг равен 4.
Для $i = 16$ имеем $S[16\,..\,22] = T$. Сдвигаться больше некуда — строка $S$ закончилась.

Итог: соответствие найдено в позициях 12 и 16.

Пример 2. Пусть $T = abccdbccabcc$, построить таблицу для $rpr(l;T)$ и $Shift_2(l)$. Используя эвристику хорошего суффикса, проделать поиск подстроки $T$ в строке $S = abccdbcccbccabccabcc$.

| $l$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $rpr(l)$ | 11 | 11 | –1 | 6 | 1 | 0 | –1 | –2 | –3 | –4 | –5 | –6 | –7 |
| $Shift_2(l)$ | 2 | 1 | 12 | 4 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |

```
Номер 1 2 3 4 5 6 7 8 9 10  12  14  16  18  20  22  24
  S = a b c c d b c c c b c c a b c c a b c c a b c b c
  T1= a b c c d b c c a b c c
  T2=         a b c c d b c c a b c c
  T3=                         a b c c d b c c a b c c
```

Для $i = 1$ имеем $S[10\,..\,12] = T[10\,..\,12]$, $S[9] \neq T[9]$, следовательно, $l = 3$. Сдвиг равен 4.
Для $i = 5$ имеем $S[10\,..\,16] = T[6\,..\,12]$, $S[9] \neq T[5]$, следовательно, $l = 7$. Сдвиг равен 8.
Для $i = 13$ имеем $S[24] \neq T[12]$, следовательно, $l = 0$. Сдвиг равен 2. Однако сдвиг больше чем на 1 сделать нельзя, так как «выйдем за правый край» строки $S$. Поиск закончен.

Итог: соответствие не найдено.

Пример 3. Пусть $T = aaaa$, построить таблицу для $rpr(l;T)$ и $Shift_2(l)$. Используя эвристику хорошего суффикса, проделать поиск подстроки $T$ в строке $S = abaaaaaabcaaa$.

| $l$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $rpr(l)$ | 1 | 1 | 1 | 1 | 0 |
| $Shift_2(l)$ | 4 | 3 | 2 | 1 | 1 |

```
Номер 1 2 3 4 5 6 7 8 9 10  12
  S = a b a a a a a a b c a a a
  T1= a a a a
  T2=     a a a a
  T3=       a a a a
  T4=         a a a a
  T5=           a a a a
  T6=                   a a a a
```

Для $i = 1$ имеем $S[3\,..\,4] = T[3\,..\,4]$, $S[2] \neq T[2]$, следовательно, $l = 2$. Сдвиг равен 2.
Для $i = 3$ имеем $S[3\,..\,6] = T$, следовательно, $l = 4$. Сдвиг равен 1.
Для $i = 4$ имеем $S[4\,..\,7] = T$, следовательно, $l = 4$. Сдвиг равен 1.
Для $i = 5$ имеем $S[5\,..\,8] = T$, следовательно, $l = 4$. Сдвиг равен 1.
Для $i = 6$ имеем $S[9] \neq T[4]$, следовательно, $l = 0$. Сдвиг равен 4.

Для $i = 10$ имеем $S[11\,..\,12] = T[3\,..\,4],\ S[10] \neq T[1]$. Сдвигаться больше некуда — строка $S$ закончилась.

Итог: соответствие найдено в позициях 3, 4 и 5.

Отметим, что таблицу $Shift_2$ можно эффективно заполнить, используя понятие префикс функции.

## 7.4. Алгоритм Бойера–Мура

Алгоритм Бойера–Мура состоит в следующем:

1) обработать шаблон $T$ — составить таблицы сдвигов $Shift_1$ и $Shift_2$;

2) инициализировать указатель $i = 1$;

3) начать посимвольное сравнение фрагмента $S[i\,..\,i+m-1]$ и шаблона $T$ справа налево и продолжать до тех пор, пока не наступит одно из двух событий: обнаружится несоответствие символов, обнаружится полное соответствие фрагмента $S[i\,..\,i+m-1]$ и шаблона $T$;

4) если обнаружилось полное соответствие фрагмента $S[i\,..\,i+m-1]$ и шаблона $T$, запомнить удачную позицию $i$;

5) сдвинуться на $\max\{Shift_1\,;\,Shift_2\}$;

6) если $i \leq n - m + 1$, перейти к шагу 3.

## 7.5. Задания для самостоятельной работы

1. Построить таблицу $N$ наиболее правых вхождений символов в строку $T = acdbdcaexx$.

| *char* | a | b | c | d | x |
|---|---|---|---|---|---|
| $N(char; T)$ | | | | | |

2. Построить таблицу $rpr(l; T)$ и $Shift_2$ для строки $T = bcdbcabcabc$. Пользуясь эвристикой хорошего суффикса, найти $T$ в строке $S = bcdbcabcdbcabcabcabcabcabcbc$.

3. Для строки $S = WHICH\_FINALLY\_HALTS.\_\_AT\_THAT\_PIONT$ и подстроки $T = AT\_THAT$ проиллюстрируйте по шагам работу алгоритма Бойера–Мура.

4. Прочитайте статью: Boyer R. S., Moore J. S. A fast string searching algorithm // Comm. ACM. 1977. P. 762–772.

# Оглавление