Столяр Сергей Ефимович

ЗАНЯТИЕ 10. КОДЫ И СЖАТИЕ ДАННЫХ ЧАСТЬ 2

Все уменьшилось в мире.
Всем везде стало тесно.
Вот и в моей квартире
От давки окошко треснуло.
Олег Григорьев [Григорьев]

Предыдущее **Занятие 9** мы завершили обсуждением азбуки Морзе. Там же мы установили, что ее применение для кодирования дискретных сообщений не имеет смысла, поскольку обеспечить однозначное декодирование будет невозможно без передачи «пауз» между кодовыми словами.

Тем не менее, для разминки вам предлагается следующая задача.

Задача 1.

Предположим, вопрос об обратимости отображения F: $A_{Morse} \to \{0,1\}_{bit}$, то есть возможности восстановления закодированного азбукой Морзе сообщения, нас не волнует. Тогда это отображение, очевидно, обеспечивает сжатие входного текста.

Уровень 1. Переведите в код Морзе (без «пауз») стихотворение Олега Григорьева и подсчитайте коэффициент сжатия текста.

Уровень 2. Напишите программу **MorseCode**, которая, получая на вход кодовую таблицу и далее ASCII текст, перекодирует его в код Морзе «без пауз».

Формат ввода:

Натуральное число N – длина кодовой таблицы.

Следующие *N* строк содержат в алфавитном порядке саму таблицу кодируемых символов. В каждой строке таблицы записана пара значений: очередной ASCII символ и, через пробел, соответствующая кодовая комбинация, составленная из символов «точка» и «тире».

Следующие строки входного файла содержат предназначенный для кодирования текст, причем он состоит только из символов, приведенных в таблице.

Формат вывода:

Закодированный текст. Символ «точка» заменяется битовым нулем, «тире» – битовой единицей, все кодовые слова склеиваются в одну двоичную последовательность, «конец строки» игнорируется. При этом полученная двоичная последовательность, имеющая длину, не кратную 8, дополняется битовыми нулями.

Пример.

Ввод:

```
6
A .–
E .
H ....
L .–..
M ––
T –
HAMLET
```

Вывод:

• D {hex-коды 07 и 44}

Включенный в задачу 1 тестовый пример демонстрирует довольно существенное сжатие: исходная шестисимвольная строка «HAMLET» заменяется 2-символьной строкой «• D», то есть укорачивается в 3 раза.

Вообще говоря, удивляться тут нечему, поскольку несколько символов ис-

ходной строки, будучи «более популярными» в английском языке, получили от Сэмюэля Морзе совсем короткие *кодовые слова*. Но и без того коэффициенту сжатия гарантировано условие K = 5/8, так как кодовые слова алфавита Морзе не длиннее 5. Скажем, последовательность 0123456789 пакуется в точности с коэффициентом K = 5/8.

Пожалуй, пора для обсуждаемого нами круга вопросов ввести подходящую терминологию. Устанавливая «неравноправие» при двоичном кодировании символов, мы естественным образом приходим к понятию кодов переменной длины.

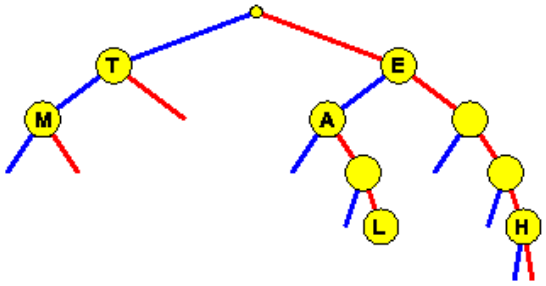
КОДЫ ПЕРЕМЕННОЙ ДЛИНЫ

Почему же у него тогда ветки? Откуда здесь деревья? <...>
Стоило ей подойти поближе, как все вокруг превращалось в деревья <...>
Льюис Кэрролл [ЛК]

Возможно, наши рассуждения кажутся пока далекими от практических нужд. И впрямь, какой смысл *паковать* текст, не имея шансов в точности восстановить его в нужный момент? Это же не коктейль, субстанции которого после смешивания нет смысла вновь разделять.

Что ж, практическое использование уже близко. Но дальнейшее обсуждение механизма кодирования целесообразно связать с рассмотрением одной из разновидностей *двоичных* (иначе *бинарных*) *деревьев*, а именно – *кодовых деревьев*.

На рисунке вы видите часть кодового дерева для алфавита Морзе, где представлены только символы из знакомого уже примера. Правые ребра дерева соответствуют битовым 0 («точкам» азбуки Морзе), левые – 1 («тире» азбуки Морзе).



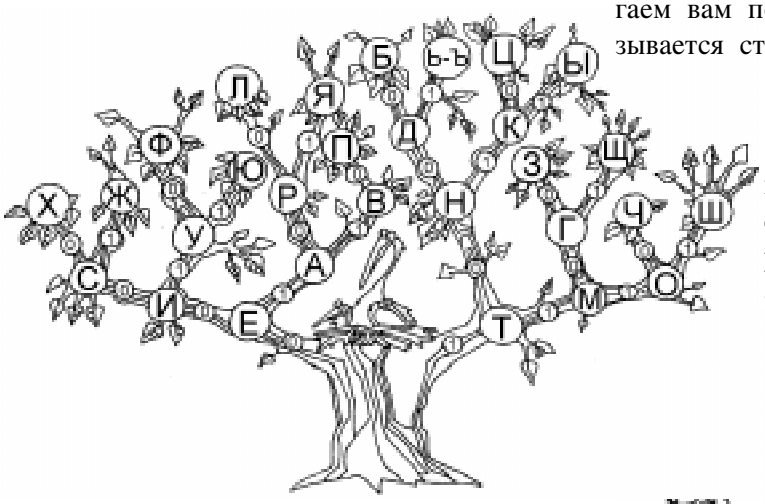
Каждому кодовому слову соответствует *путь* по ребрам дерева от *корневой* вершины (на рисунке она сверху) до вершины, конечной для кодируемого символа. Но неприятность состоит в том, что для многих символов этот путь составляет часть более длинного маршрута, ведущего к другой букве алфавита (на рисунке незавершенные пути иллюстрируются «свободным» ребром). Тут бы и нужен знак «дальнейший проезд запрещен» (он же – «пауза» в технических системах передачи данных, реально использующих азбуку Морзе), но двоичное дерево такой услуги не предоставляет. Так, в нашем примере кодовое слово символа E составляет часть другого слова – A, а оно, в свою очередь, входит целиком в код для L.

В структуре *дерева* некорневые вершины, из которых возможно дальнейшее продвижение в направлении от корня, называются *внутренними*, остальные – *терминальными* (или *оконечными*, или *листьями*).

Например, в кодовом дереве Морзе листом является всякая вершина, находящаяся «в 5 ребрах пути» от корня, но не только. Будь дерево *полным*, – то есть деревом, у которого все терминальные вершины расположены на одинаковом расстоянии от корня и, кроме того, каждая внутренняя вершина имеет двух *сыновей*, – таких вершин было бы ровно 2^5. Но в дереве Морзе их гораздо меньше – это видно из таблицы кодов. В частности, продолжая путь из вершины H, за оставшийся шаг можно добраться либо до терминальной вершины 4, либо до вершины 5.

Задача 2.

Уровень 1. Достройте рисунок, разместив на нем все кодовое дерево Морзе.



Кодовое дерево нам понадобится отнюдь не только как иллюстрация. Оно найдет применение при выполнении *распаковки* закодированного текста – в качестве *дерева поиска*. Вновь обратимся к тому же примеру – строке HAMLET. При ее упаковке в код Морзе была получена битовая строка

0000 01 11 0100 0 1,

в которой промежутки мы оставили для наглядности, чтобы они разделяли «бывшие» буквы. (Выше, в упражнении, мы предлагали еще добавлять в хвост нули – для кратности восьми длины последовательности, но сейчас нам это уже не нужно.)

Теперь, просматривая двоичную строку слева направо, станем для каждого очередного бита – как указателя направления – выбирать соответствующую ветвь. Поскольку данными о «паузах» мы не обладаем, то должны следовать одной из альтернативных стратегий: либо сразу останавливаться по достижении узла, определяющего какой-нибудь ASCII символ, либо всякий раз «идти до конца», то есть до вершины, из которой подходящая ветвь не растет. «Сняв с дерева» очередной символ, дальнейшую обработку мы вновь продолжаем от корня.

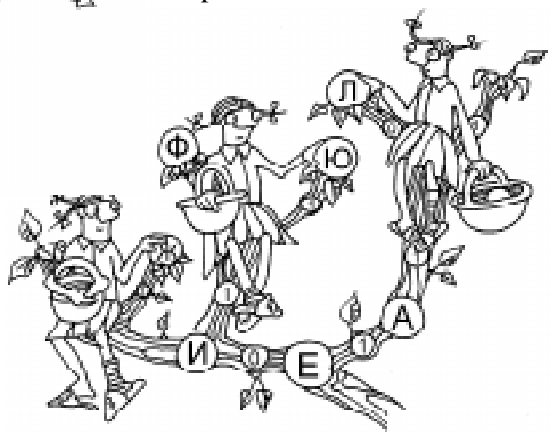
Очевидно, на выходе первого алгоритма мы получим символьную строку

EEEE ET TT ETEE E T

имеющую мало общего с изначально кодировавшимся текстом. Результат работы второго алгоритма, который мы предлагаем вам получить самостоятельно, оказывается столь же обескураживающим.

Задача 3.

Уровень 1. Распакуйте приведенную выше битовую строку, пользуясь кодовым деревом Морзе и руководствуясь алгоритмом «идти до конца».



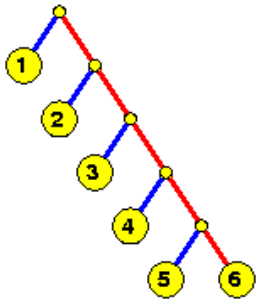
Но не будем унывать: на алфавите Морзе, как говорится, свет клином не сошелся. Зато сам механизм кодового дерева оказывается перспективным для распаковки. Надо лишь при формировании дерева «развешивать» все символы в качестве листьев и применять вторую из описанных стратегий.

Иначе говоря, никакое кодовое слово не может являться началом никакого другого кодового слова. Это требование называют *условием Фано* – по имени итальянского математика Gino Fano (1871–1952).

Сформировать с учетом условия Фано какое-нибудь кодовое дерево (назовем его деревом Фано) большого труда не составляет. Можно, например, воспользоваться следующим рекурсивным механизмом. В каждом узле, начиная от корня, станем отращивать две ветви – левую и правую. Одну из ветвей – договоримся, что ею будет левая – сразу подрежем, разместив на ее конце лист, то есть очередной символ алфавита, а для другой – пра-

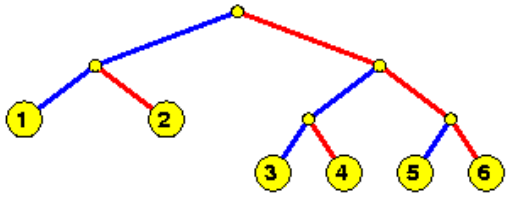
вой повторим процедуру; и так до исчерпания алфавита, но когда останется единственный, последний, символ, то его сделаем листом в текущем узле.

Например, ограничиваясь алфавитом из 6-ти символов {1, 2, 3, 4, 5, 6}, по этой технологии мы вырастим дерево Фано #1 – оно на рисунке.



Высота дерева, то есть длина пути от корня до наиболее удаленного листа, оказывается лишь на единицу меньше мощности «развешенного» алфавита.

Собственно условие Фано не накладывает особо жестких ограничений на механизм выращивания дерева. Приведем еще один вариант – дерево #2.

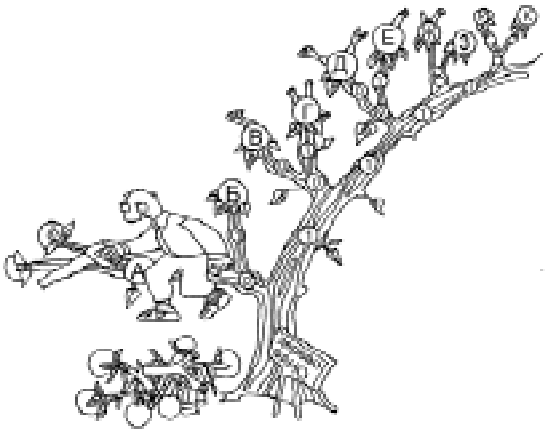


В отличие от дерева #1, теперь в каждом узле мы выращиваем слева, пока это возможно, *строго бинарное* (у *СБ-дерева* любой внутренний узел имеет ровно двух сыновей) *поддерево* с двумя листьями. Критическая ситуация наступает, когда от алфавита останется не более пары символов: из 2-х мы строим в текущем узле СБ - поддерево, а единственный символ размещаем тут же. При ненулевой мощности развешиваемого алфавита |A| высота дерева #2 составляет (|A|+1) div 2.

Итак, остановимся на этих двух примерах деревьев Фано, осознавая, что «лесопосадки» можно было бы и расширить. Задавшись целью закодировать весь ASCII алфавит, первое из них пришлось бы вырастить до высоты 255, а второе – до 128. Естественно заключить, что использование обоих деревьев, в общем случае, совершенно нерентабельно (что в отдельных случаях отнюдь не исключает достижения хорошего коэффициента сжатия).

Задача 4.

Уровень 1. Предложите отличный от двух продемонстрированных вариант построения дерева Фано.



Развесим на обоих приведенных деревьях Фано, в порядке поступления, элементы 26-символьного алфавита {A .. Z}. Закодируйте, используя каждое из деревьев, символьную строку HAMLET и определите соответствующие коэффициенты сжатия.

Уровень 2. Напишите программу **FanoCode**, которая для заданного алфавита строит кодовое дерево Фано #2, а затем использует дерево для упаковки предложенного ASCII текста.

Формат ввода:

Символьная строка, составленная из символов развешиваемого алфавита (порядок символов в строке сохраняется при размещении их на дереве). Далее идут строки, содержащие предназначенный для кодирования текст, причем он состоит только из символов, приведенных в таблице.

Формат вывода:

Закодированный текст, все кодовые слова склеены в общую двоичную последовательность, «конец строки» игнорируется. При этом полученная двоичная последовательность, имеющая длину, не кратную 8, дополняется битовыми нулями.

Пример:

Ввод:

ABCDEF
BAD {10 11 010}
FACED {000 11 011 000 010}

Вывод:

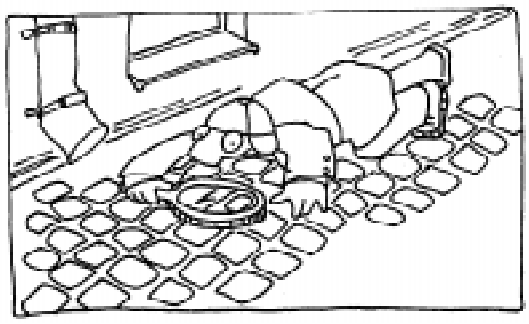
┤ 6 ▸ {hex-строка: ‘B4 36 10’}

Код, построенный в согласии с условием Фано, принято называть *префиксным*. Если упаковать ASCII текст с помощью префиксного кода, то дальнейшая распаковка полученной двоичной последовательности осуществляется без потерь.

Алгоритм распаковки, как легко заметить, имеет *линейную трудоемкость* – O(n), диктуемую входной последовательностью. При чтении из нее очередного бита мы выбираем на кодовом дереве соответствующую ветвь из текущего узла; достигнув же терминальной вершины, помещаем висящий в ней символ в выходной текст и вновь возвращаемся к корню.

Задача 5.

Уровень 2. Напишите программу-декодер **FanoDecode**, осуществляющую декодирование файла, сформированного при работе кодера **FanoCode**.



Замечание: наш кодер допускал формирование двоичной последовательности с лишними нулевыми битами в конце – для кратности длины числу 8. Поэтому во входном потоке будем указывать, сколько в тексте лишних бит.

Формат ввода:

Символьная строка, оставленная из символов развешиваемого алфавита; порядок символов в строке сохраняется при размещении их на дереве.

Натуральное число – количество лишних бит (от 0 до 7).

Следующие строки входного файла содержат текст, полученный на выходе кодера.

Формат вывода:

Распакованный текст.

Пример:

Ввод:

ABCDEF
3
┤ 6 ▸

Вывод:

BADFACED

Итак, мы выяснили, что префиксный код:

– является кодом переменной длины;
– гарантирует однозначное кодирование/декодирование текста;
– обеспечивает линейную трудоемкость при декодировании;
– допускает различные реализации для заданного алфавита.

Если первые три обстоятельства мы принимаем вполне благосклонно, то вот последнее дает повод задуматься. Если решений несколько, почему бы нам не выбирать *наилучшее*?

Вполне разумный вопрос требует математической формализации и порождает целый «букет» проблем:

– во-первых, следует «договориться» о *количественном критерии* для сравнения решений;
– во-вторых, установить достижимость «наилучшего» из них и научиться строить интересующий нас код, не прибегая к переборному механизму.

Решение первой задачи, к счастью, больших усилий не требует: удачным выбором представляется хорошо нам знакомый *коэффициент сжатия* текста K. Очевидно, при конечном алфавите наше кодовое дерево должно иметь конечную высоту: строить бесконечное дерево бессмысленно. При длине L_S пути от корня

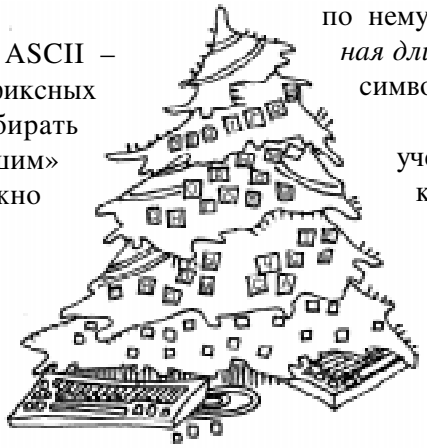
до листа s – то есть количестве бит в кодовом слове символа S – и кодируемом тексте, состоящем из N символов, получаем следующее выражение для коэффициента сжатия:

$$K = 0.125 * \Sigma_{i=1,N} L_i / N$$

Значение константы 0.125 диктуется здесь одинаковой длиной всех кодовых слов входного набора. Действительно, исходный текст закодирован в алфавите ASCII, чье кодовое дерево является *полным* с высотой 8.

Задача 6.

Уровень 1. Постройте кодовое дерево ASCII алфавита.



Имея базу – дерево ASCII – для сравнения разных префиксных кодов, мы уже можем выбирать между «плохим» и «хорошим» кодом: при K < 1 код можно отнести ко второй группе.

Задача 7.

Уровень 1. Выше приводились два примера префиксного кода для алфавита из 6-ти символов {1, 2, 3, 4, 5, 6} – деревья Фано #1 и #2. Подсчитайте по ним и сравните коэффициенты сжатия

a) для входного текста: 122333444455555666666;

b) и для другого текста: 655444333322222211111.

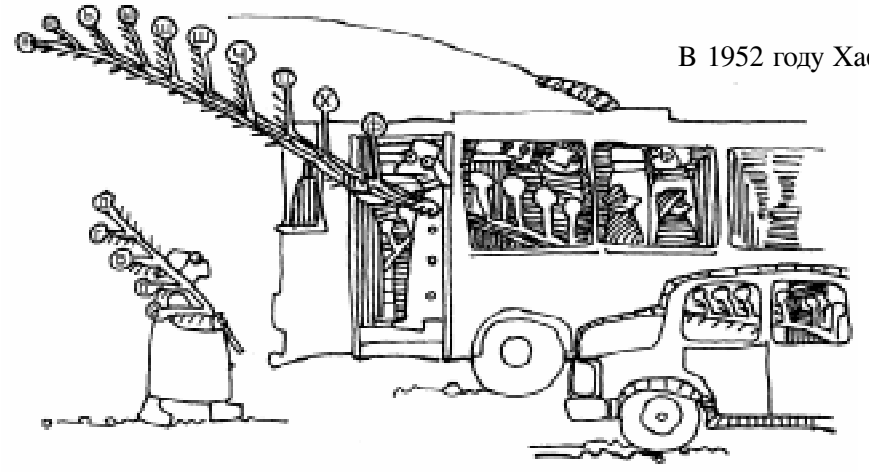
Во всех четырех случаях из приведенного упражнения обеспечен «хороший» коэффициент сжатия, и это связано с малой высотой обоих деревьев. Однако их сравнение дает повод выбирать, в каком порядке развешивать кодируемые символы. Часто встречающиеся в тексте – выгоднее вешать «поближе», чтобы уменьшить их суммарный вклад в формулу для K, то есть выбирать «лучшее» дерево.

Это означает, что при использовании кодов переменной длины не столь важна такая характеристика, как *средняя длина пути* для всех символов кодового дерева (подсчитываемая непосредственно по нему), сколько *средневзвешенная длина пути* по дереву для всех символов входного текста.

Как мы знаем, именно с учетом второй характеристики построен алфавит Морзе. Правда, не совсем понятно, по какому тексту оценивается «средневзвешенность». В данном случае речь идет о назначении длин кодовым словам на основе статистического анализа, который проведен для англоязычных текстов значительного объема. Об этом мы уже рассказывали.

Отсюда следует, что для конкретного текста вполне может существовать более подходящее, с точки зрения улучшения сжатия, кодовое дерево.

СЖАТИЕ ХАФФМЕНА



В 1952 году Хаффмен (D.A. Huffman) предложил механизм построения префиксного кода. Одновременно он доказал, что этот механизм позволяет строить *оптимальное*, по отношению к входному тексту, кодовое дерево. Иными словами, код

Хаффмена обеспечивает наилучшее сжатие входного текста в сравнении с любым другим префиксным кодом.

Реализация алгоритма сжатия Хаффмена включает несколько этапов. На первом – проводится частотный анализ появления символов в кодируемом тексте. Исходный текстовый файл при этом просматривается последовательно от начала до конца, и трудоемкость этой части алгоритма – линейная относительно количества символов на входе. На втором этапе предстоит воспользоваться полученной таблицей частот для собственно кодирования текста, вновь осуществляя его просмотр; трудоемкость этой работы мы оценим чуть позже. Но до перехода ко второму этапу целесообразно полученную таблицу еще и упорядочить по частотам. Ясно, впрочем, что трудоемкость такой сортировки, когда легко удается обойтись только ресурсами оперативной памяти, несущественна в сравнении с чтением внешнего файла. Но вот структура данных, выбранная программистом для хранения этой таблицы – и дальше мы это увидим – имеет значение.

Задача 8.

Уровень 1. Разработайте алгоритм, согласно которому для произвольного входного текста строится таблица «частота появления символа в тексте – символ», упорядоченная по неубыванию частот.

Примените этот алгоритм к короткому стихотворению поэта Олега Григорьева:

Старик сторук,
Старуха сторука.
В двести рук
Колотят друг друга.

– и не забудьте, что символ «пробел» является столь же неотъемлемой частью входного текста, как и синтаксические символы.

Уровень 2. Напишите программу **TextAnalysis**, на вход которой подается текстовый файл, а на выходе формируется упорядоченная по неубыванию таблица частот встречающихся в нем символов.

Формат ввода:

Некоторый текст.

Формат вывода:

Таблица, в каждой строке которой находится очередной символ и, через пробел, натуральное число – количество его повторений в тексте.

Рекомендация. Для подсчета частот удобно использовать стандартный прием: обзавестись линейным массивом длиной с ASCII таблицу и обнулить его ячейки-счетчики для всех 256 элементов. Далее каждый очередной символ из входного потока напрямую указывает адрес «своего» счетчика.

Пример:

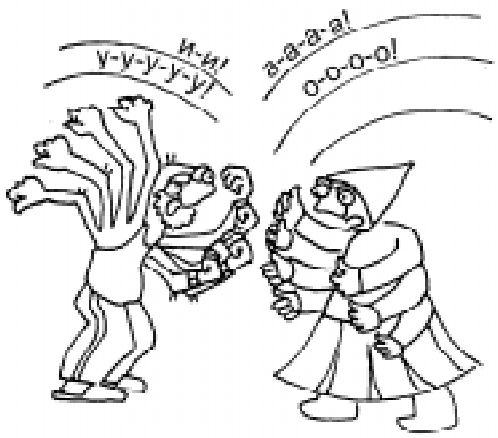
Ввод:

```
TO_BE_OR_NOT_TO_BE
```

Вывод:

```
N 1
R 1
B 2
E 2
T 3
O 4
_ 5
```

Здесь необходима небольшая пауза. К дальнейшему описанию алгоритма мы вернемся чуть позже, а сейчас предлагаю обратить внимание на время его опубликования. В далекие уже 50-е годы XX века о вычислительных машинах мало кто даже слышал. Предвидеть столь стремительное развитие ЭВМ и их программного обеспечения, свидетелями которого мы являемся, и, тем более, представить круг решаемых с их помощью задач, было трудно. Надо полагать, Хаф-



фмен не рассчитывал на сколько-нибудь широкое применение *двухпроходного* алгоритма, публикуя свое исследование.

Во всяком случае, помимо математической части работы, он провел обработку весьма солидных по объему англоязычных текстов и построил, следуя предложенному механизму, кодовую таблицу. Этот код, который мы будем называть *статическим кодом Хаффмена*, неплох «в среднем», примерно как код Морзе. Его применение отменяет первый этап при сжатии конкретного текста, делая алгоритм в этом случае *однопроходным*. Разумеется, в нем не фигурируют все 256 символов ASCII таблицы, поскольку она еще не существовала в то время.

Вот небольшая часть статического кода Хаффмена:

E 100
T 001
A 1111
O 1110
N 1100
... ...
J 110100000
Q 1101000101
Z 1101000100

Любопытно сравнить этот, пусть даже небольшой, фрагмент с кодом Морзе. Мы видим, что длина кода у «часто используемых» символов не столь мала, как у Морзе, а «редкие» символы получили весьма длинные коды. Потому, даже не прибегая к вычислению коэффициента сжатия для конкретного входного текста, можно утверждать, что он явно «хуже», чем у Морзе. Но этого и следовало ожидать, рассчитывая на возможность декодирования.

Обратите также внимание на нижние строки таблицы: коды соответствующих символов «даже» длиннее, чем 8-битовые из ASCII таблицы. Это неизбежная дань, которую мы платим за укорачивание кодов других символов. И, скажем, если бы нам пришлось паковать текст, «перенасыщенный» такими символами, то на выходе алгоритма мы, вместо сжатия, обнаружили бы растяжение файла.

Применение готовой кодовой таблицы к обыкновенному, а не столь экзотическому тексту, конечно, не приведет к указанному обескураживающему результату. Однако исключение первого – предварительного – этапа анализа текста не дает возможности учесть фактические частоты появления отдельных символов. Стало быть, об оптимальности статического сжатия говорить просто не приходится.

Напротив, двухпроходный алгоритм Хаффмена обеспечивает оптимальное сжатие любого текста с коэффициентом K = 1. А это значит, что «обычный» ASCII код несет в себе избыточность, поскольку можно посимвольно представить текстовую информацию более компактно (кроме, конечно, случая K = 1).

Задача 9.

Уровень 1. Практически, для «обыкновенного» текста, пакуемого двухпроходным алгоритмом Хаффмена, коэффициент сжатия меньше 1. Но в приведенной выше оценке для K отражена возможность «крайнего случая». Приведите пример (текст), когда эта ситуация возникает.

Уровень 2. Уместен вопрос, почему же коды переменной длины не используются вместо менее компактного ASCII кодирования? Причин тому, как минимум, две. Во-первых, второй этап работы алгоритма Хаффмена не всегда технологически осуществим, поскольку входной поток может поступать не только из дискового файла, но и по иному каналу ввода (через модем, к примеру), да и «накладно» обыкновенный файл обрабатывать дважды. Однако еще существенней вторая причина: текст-то в формате ASCII читается «порциями» – по 8 бит одновременно, то есть имеет место параллельная обработка данных, а это совершенно исключено при использовании кодов переменной длины, требующих строго последовательного просмотра.

Вернемся вновь к описанию алгоритма Хаффмена и выясним, как же стро-

ится кодовое дерево (Н-дерево) по сформированной к тому времени таблице частот. В отличие от знакомого нам механизма «выращивания» дерева от корня к листьям, у Н-дерева изначально заданы будущие листья: в их роли выступают элементы упорядоченного списка частот. Так, в уже знакомом нам примере входного текста

TO_BE_OR_NOT_TO_BE

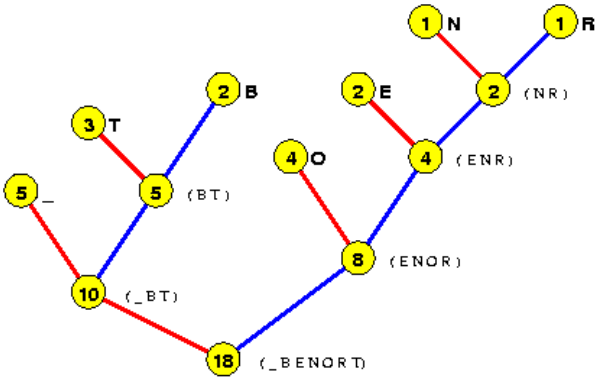
фигурировал и соответствующий список:

1(N) 1(R) 2(E) 2(B) 3(T) 4(O) 5(_)

Далее, при текущем состоянии списка, в нем выбираются два узла с наименьшим «весом», и для них создается «родитель», которому приписывается суммарный вес обоих сыновей. Тем самым, на этом шаге у дерева появляется новый узел с двумя исходящими из него к обработанным узлам дугами. Список узлов теперь обновляется: обработанная пара детей исключается, а родитель добавляется в него с учетом своего веса. Обратите внимание, что термин «список» в данном контексте относится именно к структуре представления данных, поскольку программная реализация указанных операций наиболее рационально реализуется как раз для списка. Впрочем, при желании можно обойтись и обыкновенным вектором, однако придется всякий раз заново отыскивать очередную пару узлов для их объединения, тогда как список начинается именно с них.

В конце концов, в списке остается единственный узел, который становится корнем Н-дерева. Остается приписать каждой паре дуг, выходящих из одного родителя, альтернативные значения – битовые 0 и 1. При этом никакого приоритета в отношении дуг нет, так как механизм выращивания дерева обеспечивает построение префиксного кода; но программист, надо полагать, руководствуется каким-то определенным правилом.

На рисунке вы видите дерево Хаффмена, выращенное для последнего обсуждавшегося примера, и примененное нами правило выбора из {0,1} вполне очевидно.



Теперь, когда готово кодовое дерево, остается вторично обработать входной текст, помещая в выходной поток, вместо каждого очередного символа, соответствующее ему кодовое слово. Чтобы подбор кодового слова не занимал лишнего времени, хорошо бы заготовить, вместо дерева, упорядоченную по алфавиту таблицу кодов, которая для нашего примера выглядит так:

Пробел	00
B	011
E	110
N	1110
O	10
R	1111
T	010

Эта таблица обеспечивает возможность дихотомического поиска по ней очередного кодового слова. Общая же трудоемкость этапа собственно кодирования, очевидно, мало отличается от работы на первом этапе и линейно зависит от N.

Однако такая таблица лишь на картинке выглядит столь простой; в действительности надо позаботиться о том, как для значений из правого ее столбца указывать количество бит, составляющих кодовое слово. Здесь возможны разные варианты решения, но все они относятся, скорее, к технике программирования, а не к алгоритмическому механизму.

Задача 10.

Уровень 1. При определенном соотношении частот длины кодовых слов Н-дерева могут оказаться такими: 1, 2, 3, ..., s-1, s, s. Сконструируйте такой текст.

В отличие от статического – однопроходного – алгоритма, двухпроходный алгоритм уместно назвать *динамическим кодом Хаффмена*. Как уже сказано, его кодовое дерево является оптимальным среди своих собратьев, если применять его для упаковки всего текста.

К чему последняя оговорка? Оказывается, механизму выращивания Н-дерева после предварительной обработки всего входного потока существует альтернатива. Но об этом – в другой раз. А пока – еще пара замечаний в отношении динамического кода Хаффмена.

Возможность сжатия, очевидно, обусловлена неравномерным распределением частот вхождения в текст разных символов. И чем существенней «неравномерность», тем лучше удается паковать входной текст. В самом деле, при одинаковых (или достаточно близких) частотах разница в длинах кодовых слов была бы минимальна. Соответственно, мы имели бы нечто вроде дерева ASCII.

Механизм компрессии/декомпрессии связан с использованием одного и того же кодового дерева. Очевидно, к упакованному тексту надо «приложить» это дерево, чтобы обеспечить возможность распаковки. Естественно, размер файла, в который помещается выходной код, увеличивается за счет такого добавления. По этой причине применение динамического кода Хаффмена к «коротким» файлам совершенно непродуктивно. Что же касается способа размещения кодового дерева в выходном файле, то это опять-таки относится к технике программирования, а не к алгоритмическому содержанию.

Уровень 2. Напишите программу, конструирующую такой текст.

Формат ввода:

Натуральное число s.

Формат вывода:

Текст, частоты появления символов в котором таковы, что длины кодовых слов следующие: 1,2,3,...,s-1,s,s.

Пример:

Ввод:

1

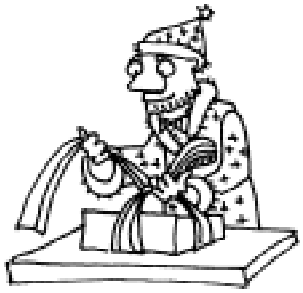
Вывод:

AB

Задача 11.

Уровень 1. Для предлагаемого слова постройте динамический код Хаффмена и упакуйте слово. Вычислите коэффициент сжатия.

Слово: МАТЕМАТИКА



Столяр Сергей Ефимович,
учитель информатики,
гимназия № 470,
лицей «Физико-Техническая школа»,
Санкт-Петербург.