ГЛАВА 6

ПОИСК

Взглянем на запись...

— ЭЛ СМИТ (AL SMITH) (1928)

Эта глава могла бы носить более претенциозное название — "Хранение и получение информации"; с другой стороны, ее можно было бы назвать кратко и просто — "Просмотр таблиц". В ней мы займемся вопросами накопления информации в памяти компьютера и рассмотрим способы ее быстрого извлечения. Зачастую мы сталкиваемся с избыточной информацией, и лучшее, что с ней можно сделать, — забыть о ней или уничтожить ее. Однако нередки ситуации, когда крайне важно сохранить материал и организовать его таким образом, чтобы впоследствии обеспечить к нему максимально быстрый доступ.

Основная часть этой главы посвящена изучению простейшей задачи: поиску информации, сохраненной с конкретным идентификатором. Например, в численном приложении может понадобиться найти $f(x)$ по данному x и таблице значений f; другим примером может послужить поиск перевода на английский язык некоторого русского слова.

В целом, мы предполагаем, что имеется набор из N записей и задача состоит в нахождении одной из них. Как и в случае сортировки, мы полагаем, что каждая запись включает специальное поле, именуемое ее *ключом*; данный термин удачен, так как, несомненно, отражает тот печальный факт, что ежедневно миллионы людей тратят время и силы на поиски собственных неизвестно куда запропастившихся ключей... В общем случае нам требуется N различных ключей для того, чтобы каждый из них однозначно идентифицировал связанную с ним запись. Набор всех записей именуется *таблицей* или *файлом*, причем слово "таблица" обычно используется для описания маленького файла, а слово "файл" — для описания большой таблицы. Большой же файл (или группу файлов) часто называют *базой данных*.

Алгоритм поиска имеет так называемый *аргумент*, K, и задача заключается в нахождении записи, для которой K служит ключом. Результатом поиска может быть одно из двух: либо поиск завершился *успешно*, и уникальная запись, содержащая K, найдена, либо поиск оказался *неудачным*, и запись с ключом K не найдена. После неудачного поиска иногда желательно внести новую запись, содержащую K, в таблицу. Метод, осуществляющий это, называется алгоритмом *поиска и вставки*. Некоторые аппаратные устройства, известные как *ассоциативная память*, решают данную проблему автоматически, путем, который можно сравнить с функционированием мозга человека, однако мы все же будем изучать технологию поиска, применяемую в обычных цифровых компьютерах общего назначения.

Хотя цель поиска — нахождение информации, хранящейся в ассоциированной с K записи, алгоритмы в этой главе предназначены для поиска K. После нахождения K поиск связанной с ним информации становится тривиальной задачей, зависящей от способа хранения информации. Например, найдя K в TABLE + i, мы обнаружим связанные с ним данные в TABLE + i + 1, в DATA + i или в еще каком-то месте, которое определяется способом хранения информации. Поэтому мы считаем задачу выполненной в тот момент, когда находим K (или убеждаемся в его отсутствии).

Поиск обычно является наиболее "времяемкой" частью многих программ, и замена плохого метода поиска хорошим может значительно увеличить скорость работы программы. К тому же часто представляется возможным так изменить данные или используемые структуры, что поиска удается избежать совсем. Одним из таких способов может быть связь данных (например, при использовании списка с двойными связями поиск предшествующего и последующего элементов данного становится совершенно излишним). В случае, когда мы можем выбирать ключи, имеется еще один способ избежать поиска: выбрать в качестве ключа натуральные числа $\{1, 2, \ldots, N\}$; при этом запись с ключом K просто размещается в TABLE + K. Обе эти технологии были использованы, чтобы избежать поиска в алгоритме топологической сортировки , обсуждавшемся в разделе 2.2.3. Однако поиск необходим, если объекты для топологической сортировки представлены не числовыми, а символьными значениями. Естественно, в этом случае очень важно подобрать эффективный алгоритм поиска.

Методы поиска могут быть классифицированы несколькими способами. Мы можем разделить их на внутренний и внешний поиск так же, как в главе 5 мы разделяли алгоритмы сортировки на внутренние и внешние. Возможно деление на статические и динамические методы поиска, где термин "статический" означает, что содержимое таблицы остается неизменным и главная задача — уменьшить время поиска без учета времени, необходимого для настройки таблицы. Термин "динамический" означает, что таблица часто изменяется путем вставки (а возможно, и удаления) элементов. Третья возможная схема классификации методов поиска — их разделение в зависимости от того, на чем они основаны: на сравнении ключей или на некоторых числовых свойствах ключей (по аналогии с разделением на сортировку методом сравнения и сортировку методом распределения). И наконец, можно разделить методы сортировки на методы с использованием ключей действительных и преобразованных (трансформированных).

Организация этой главы представляет собой комбинацию двух последних способов классификации. Раздел 6.1 посвящен методам последовательного поиска, т. е. методам поиска "в лоб". В разделе 6.2 рассмотрены усовершенствования, основанные на сравнении ключей с использованием алфавитного или числового порядка для управления решениями. Раздел 6.3 посвящен числовому поиску, а в разделе 6.4 обсуждается важный класс методов с общим названием "хеширование", основанных на арифметическом преобразовании действительных ключей. В каждом из этих разделов рассматривается как внутренний, так и внешний поиск (как для статического, так и для динамического случаев). В каждом разделе вы найдете описание достоинств и недостатков рассматриваемых алгоритмов.

Поиск и сортировка зачастую тесно связаны между собой. Например, рассмотрим следующую задачу. *Даны два числовых множества,* $A = \{a_1, a_2, \ldots, a_m\}$ *и* $B = \{b_1, b_2, \ldots, b_n\}$. *Необходимо определить, является ли множество* A *подмножеством множества* B:$A \subseteq B$. Очевидными представляются такие варианты решения.

1. Сравнивать каждое a_i со всеми b_j последовательно до нахождения совпадения.
2. Сначала рассортировать множества A и B, а затем сделать только один последовательный проход с проверкой по обоим файлам.
3. Внести все элементы b_j в таблицу и выполнить поиск каждого значения a_i.

Каждое из этих решений имеет свои достоинства для различных значений m и n. Решение 1 требует порядка c_1mn единиц времени, где c_1 — некоторая константа. Решение 2 требует порядка $c_2(m \lg m + n \lg n)$ единиц времени, где c_2 — некоторая (бóльшая) константа. При выборе подходящего метода хеширования решение 3 займет около $c_3m + c_4n$ единиц времени для некоторых (еще бóльших) констант c_3 и c_4. Отсюда следует, что решение 1 подходит для очень небóльших значений m и n; при увеличении множеств лучшим становится решение 2. Затем наилучшим станет решение 3 — до тех пор, пока n не превысит размер внутренней памяти. После этого наилучшим обычно вновь становится решение 2, пока n не вырастет до совсем уж громадных значений... Итак, мы видим, что существуют ситуации, в которых сортировка служит хорошей заменой поиску, а поиск — отличной заменой сортировке.

Более сложные задачи поиска зачастую сводятся к более простым (которые мы и рассматриваем). Предположим, например, что ключи представляют собой слова, которые могут быть записаны с небольшими ошибками. Наша задача — найти запись, невзирая на ошибки в ключах. Если мы сделаем две копии файла, в одном из которых ключи расположены в обычном лексикографическом порядке, а в другом — в обратном порядке, справа налево (как если бы их читали задом наперед), искаженный аргумент поиска будет, вероятно, совпадать до половины (или более) своей длины с записью в одном из файлов. Методы поиска, описываемые в разделах 6.2 и 6.3, таким образом, могли бы быть приспособлены для поиска по искаженному ключу.

Подобные задачи привлекли внимание в связи с вводом в действие систем резервирования билетов на самолеты и других подобных им систем, в которых велика вероятность возникновения ошибки из-за плохой слышимости или некаллиграфического почерка. Целью исследований был поиск метода преобразования аргумента в некий код, который позволил бы группировать различные варианты одной фамилии. Далее описан метод "Soundex", первоначально разработанный Маргарет К. Оделл (Margaret K. Odell) и Робертом С. Расселом (Robert C. Russell) [см. *U. S. Patents 1261167* (1918), *1435663* (1922)] и нашедший широкое применение для кодирования фамилий.

1. Оставить первую букву имени и удалить все буквы a, e, h, i, o, u, w, y в других позициях.
2. Назначить оставшимся буквам (кроме первой) следующие числовые значения:

b, f, p, v $\rightarrow$ 1	l $\rightarrow$ 4
c, g, j, k, q, s, x, z $\rightarrow$ 2	m, n $\rightarrow$ 5
d, t $\rightarrow$ 3	r $\rightarrow$ 6

3. Если в исходном слове до выполнения шага 1 две или более буквы с одним и тем же кодом стояли рядом, удалить их все, кроме первой.
4. Преобразовать полученный результат в формат "буква, цифра, цифра, цифра" (приписывая необходимое количество нулей справа, если в результате получилось меньше трех цифр, или отбрасывая лишние цифры, если их больше трех).

Например, фамилии Euler, Gauss, Hilbert, Knuth, Lloyd, Łukasiewicz и Wachs имеют коды E460, G200, H416, K530, L300, L222 и W200 соответственно. Естественно, одинаковые коды могут иметь и совсем разные имена. Так, приведенные выше коды могут быть получены из следующих фамилий: Ellery, Ghosh, Heilbronn, Kant, Liddy, Lissajous и Waugh. С другой стороны, такие схожие имена, как Rogers и Rodgers, Sinclair и St. Clair или Tchebysheff и Chebyshev, имеют разные коды. Тем не менее коды Soundex существенно увеличивают вероятность нахождения имени по одному из вариантов написания. (Чтобы получить более детальную информацию по этому вопросу, обратитесь к работам C. P. Bourne, D. F. Ford, *JACM* **8** (1961), 538–552; Leon Davidson, *CACM* **5** (1962), 169–171; *Federal Population Censuses 1790–1890* (Washington, D.C.: National Archives, 1971), 90).

При использовании схем типа Soundex нет необходимости в предположении, что все ключи различны. Мы можем составить списки записей с одинаковыми кодами, рассматривая каждый список в качестве отдельного модуля.

В больших базах данных наблюдается тенденция к более сложным выборкам. Зачастую пользователи рассматривают различные поля записей как потенциальные ключи с возможностью поиска, если известна только часть информации, содержащейся в ключе. Например, имея большой файл с информацией об артистах, продюсер может захотеть найти незанятых актрис в возрасте от 25 до 30 лет, неплохо танцующих и говорящих с французским акцентом. Имея подобный файл с бейсбольной статистикой, спортивный обозреватель может захотеть узнать общее количество очков, заработанных командой Chicago White Sox в 1964 году в седьмых периодах ночных игр при условии, что подающий был левшой... Люди любят задавать сложные вопросы. Для того чтобы иметь возможность получить на них ответ, в книге имеется раздел 6.5, в котором содержится введение в методы поиска по вторичному ключу (многоатрибутный поиск).

Прежде чем перейти к собственно изучению методов поиска, полезно взглянуть на историю данного вопроса. В докомпьютерную эру имелось множество книг с таблицами логарифмов, тригонометрическими и другими таблицами (те, кто помнят, что означает аббревиатура Б3-21 или Б3-34, несомненно, помнят, что такое "таблицы Брадиса". — *Прим. перев.*). Фактически многие математические вычисления были сведены к поиску. Затем эти таблицы трансформировались в перфокарты, которые использовались для решения научных задач с помощью распознающих, сортирующих и копирующих перфораторных машин. Однако с появлением компьютеров с хранимыми программами стало очевидным, что проще вычислить значение $\log x$ или $\cos x$ заново, чем найти его в таблице. (Следует, однако, заметить, что на определенном этапе развития вычислительной техники для некоторых приложений, особо критичных ко времени расчетов (в основном для игр с графическим интерфейсом) оказалось крайне выгодным вернуться к предвычислению таблиц функций, а в процессе работы вместо вычислений осуществлять поиск. К тому же в подобных

случаях можно было использовать более быструю целочисленную арифметику. — *Прим. перев.*)

Хотя задача сортировки привлекала большое внимание еще на заре компьютерной эры, алгоритмы поиска оставались в забвении достаточно долгое время. Малая внутренняя память и наличие только устройств последовательного доступа (наподобие лент) для хранения больших файлов делали поиск либо тривиальным, либо невозможным.

Развитие и удешевление памяти с произвольным доступом уже в 50-х годах привело к пониманию того, что проблема поиска важна и интересна сама по себе. После многих лет жалоб на ограниченность пространства программисты столкнулись с таким изобилием, которое попросту не могли переварить и эффективно использовать.

Первые обзоры по проблеме поиска опубликованы А. И. Думи (A. I. Dumey), *Computers & Automation* **5**, 12 (December, 1956), 6–9; В. В. Петерсоном (W. W. Peterson), *IBM J. Research & Development* **1** (1957), 130–146; Э. Д. Бутом (A. D. Booth), *Information and Control* **1** (1958), 159–164; А. Ш. Дугласом (A. S. Douglas), *Comp. J.* **2** (1959), 1–9. Более подробный обзор, посвященный проблемам сортировки, был сделан несколько позже Кеннетом Ю. Айверсоном (Kenneth E. Iverson), *A Programming Language* (New York: Wiley, 1962), 133–158, и Вернером Букхольцем (Werner Buchholz), *IBM Systems J.* **2** (1963), 86–111.

В начале 60-х годов было разработано несколько новых процедур поиска, основанных на древовидных структурах (с ними мы встретимся немного позже). Исследования алгоритмов поиска ведутся и в настоящее время.

6.1. ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК

"НАЧАТЬ С НАЧАЛА и продолжать, пока не будет найден искомый ключ; затем остановиться." Эта последовательная процедура представляет собой очевидный путь поиска и может служить отличной отправной точкой для рассмотрения множества алгоритмов поиска, поскольку они основаны на последовательной процедуре. Мы увидим, что за простотой последовательного поиска скрывается ряд очень интересных, несмотря на их простоту, идей.

Вот более точная формулировка алгоритма.

Алгоритм S (*Последовательный поиск* (*Sequential search*)). Дана таблица записей $R_1, R_2, \ldots, R_N$ с ключами $K_1, K_2, \ldots, K_N$ соответственно. Алгоритм предназначен для поиска записи с заданным ключом K. Предполагается, что $N \geq 1$.

S1. [Инициализация.] Установить $i \leftarrow 1$.

S2. [Сравнение.] Если $K = K_i$, алгоритм заканчивается успешно.

S3. [Продвижение.] Увеличить i на 1.

S4. [Конец файла?] Если $i \leq N$, перейти к шагу S2. В противном случае алгоритм заканчивается неудачно. ▮

Обратите внимание на то, что этот алгоритм может завершиться успешно (искомый ключ найден) и неудачно (искомый ключ отсутствует). Данное утверждение справедливо для большинства алгоритмов, рассматриваемых в этой главе.

MIX-программа пишется очень просто.

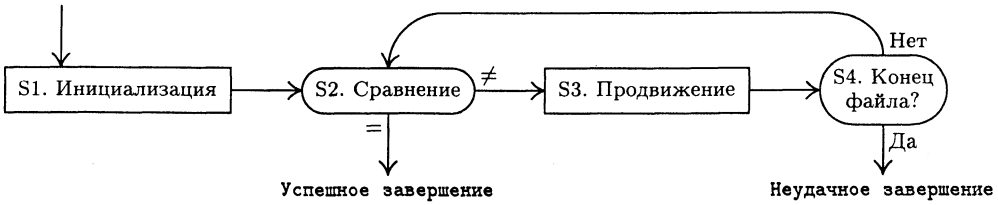

Рис. 1. Последовательный поиск.

Программа S (*Последовательный поиск*). Предположим, что K_i хранится по адресу KEY + i, а оставшаяся часть записи (R_i) — по адресу INFO + i. Программа использует rA ≡ K, rI1 ≡ $i - N$.

```
01  START   LDA  K          1      S1. Инициализация.
02          ENT1 1-N        1      i ← 1.
03  2H      CMPA KEY+N,1    C      S2. Сравнение.
04          JE   SUCCESS    C      Выход, если K = K_i.
05          INC1 1          C − S  S3. Продвижение.
06          J1NP 2B         C − S  S4. Конец файла?
07  FAILURE EQU  *          1 − S  Выход при отсутствии в таблице.
```

По адресу SUCCESS находится команда LDA INFO+N,1, которая помещает необходимую информацию в rA. ▮

Анализ этой программы несложен. Очевидно, что время выполнения алгоритма S зависит от двух факторов:

$$C \text{ — количество сравнений ключей;}$$
$$S = 1 \text{ при успешном окончании поиска, } 0 \text{ — при неудачном.} \qquad (1)$$

Программа S требует для работы $5C - 2S + 3$ единиц времени. Если при поиске успешно найден $K = K_i$, получим $C = i$, $S = 1$; таким образом, полное время составляет $(5i + 1)u$. С другой стороны, если поиск неудачен, мы получим $C = N$, $S = 0$ и общее время работы программы — $(5N + 3)u$. Если все ключи поступают на вход программы с одинаковой вероятностью, то среднее значение C в случае успешного поиска равно

$$\frac{1 + 2 + \cdots + N}{N} = \frac{N + 1}{2}. \qquad (2)$$

При этом значение среднеквадратичного отклонения, естественно, достаточно велико и составляет около $0.289N$ (см. упр. 1).

Данный алгоритм, несомненно, знаком всем программистам, но лишь некоторые из них знают, что этот способ — *не* самая лучшая реализация последовательного поиска! Небольшое изменение — и алгоритм выполняется существенно быстрее (если записей не слишком мало).

Алгоритм Q (*Быстрый последовательный поиск*). Перед вами тот же алгоритм, что и алгоритм S, однако в нем имеется дополнительное предположение о наличии фиктивной записи R_{N+1} в конце файла.

Q1. [Инициализация.] Установить $i \leftarrow 1$ и $K_{N+1} \leftarrow K$.

Q2. [Сравнение.] Если $K = K_i$, перейти к шагу Q4.

Q3. [Продвижение.] Увеличить i на 1 и перейти к шагу Q2.

Q4. [Конец файла?] Если $i \le N$, алгоритм заканчивается успешно; в противном случае алгоритм заканчивается неудачно ($i = N + 1$). ▮

Программа Q (*Быстрый последовательный поиск*). rA $\equiv K$, rI1 $\equiv i - N$.

```
01  START   LDA  K            1        Q1. Инициализация.
02          STA  KEY+N+1       1        K_{N+1} ← K.
03          ENT1 -N            1        i ← 0.
04          INC1 1          C+1-S       Q3. Продвижение.
05          CMPA KEY+N,1    C+1-S       Q2. Сравнение.
06          JNE  *-2        C+1-S       Переход к Q3, если K_i ≠ K.
07          J1NP SUCCESS       1        Q4. Конец файла?
08  FAILURE EQU  *             1-S      Выход при отсутствии в таблице. ▮
```

Используя те же значения C и S, что и в программе S, получим, что значение времени работы равно $(4C - 4S + 10)u$; таким образом, мы получаем выигрыш по сравнению с предыдущим алгоритмом в случае $C \ge 6$ для успешного поиска и $N \ge 8$ — для неудачного.

При переходе от алгоритма S к алгоритму Q использован важный ускоряющий принцип — при нескольких проверках во внутреннем цикле следует постараться свести их к одной.

Вот еще один способ сделать программу Q *еще* быстрее.

Программа Q′ (*Быстрый последовательный поиск*). rA $\equiv K$, rI1 $\equiv i - N$.

```
01  START   LDA  K              1          Q1. Инициализация.
02          STA  KEY+N+1        1          K_{N+1} ← K.
03          ENT1 -1-N           1          i ← -1.
04  3H      INC1 2          ⌊(C-S+2)/2⌋    Q3. Продвижение (дважды).
05          CMPA KEY+N,1    ⌊(C-S+2)/2⌋    Q2. Сравнение.
06          JE   4F         ⌊(C-S+2)/2⌋    Переход к шагу Q4, если K = K_i.
07          CMPA KEY+N+1,1  ⌊(C-S+1)/2⌋    Q2. Сравнение (следующее).
08          JNE  3B         ⌊(C-S+1)/2⌋    Переход к шагу Q3, если K ≠ K_{i+1}.
09          INC1 1          (C-S) mod 2    Продвижение i.
10  4H      J1NP SUCCESS          1        Q4. Конец файла?
11  FAILURE EQU  *              1-S        Выход при отсутствии в таблице. ▮
```

Внутренний цикл дублирован, что позволяет избежать выполнения половины инструкций "$i \leftarrow i + 1$", тем самым снижая затраты времени до

$$3.5C - 3.5S + 10 + \frac{(C - S) \bmod 2}{2}$$

единиц времени. При работе с большими таблицами это сохраняет до 30% нашего времени. Такой способ ускорения применим ко многим существующим программам на языках высокого уровня [см., например, D. E. Knuth, *Computing Surveys* **6** (1974), 266–269].

Можно воспользоваться другим улучшенным алгоритмом, если ключи расположены в порядке возрастания:

Алгоритм Т (*Последовательный поиск в упорядоченной таблице*). Дана таблица записей $R_1, R_2, \ldots, R_N$, ключи которых расположены в порядке возрастания: $K_1 < K_2 < \cdots < K_N$. Алгоритм предназначен для поиска записи с заданным ключом K. Для удобства и ускорения работы алгоритма предполагается наличие фиктивной записи R_{N+1} с ключом $K_{N+1} = \infty > K$.

Т1. [Инициализация.] Установить $i \leftarrow 1$.

Т2. [Сравнение.] Если $K \leq K_i$, перейти к шагу Т4.

Т3. [Продвижение.] Увеличить i на 1 и перейти к шагу Т2.

Т4. [Равенство?] Если $K = K_i$, алгоритм заканчивается успешно. В противном случае — неудачное завершение алгоритма. ▮

В предположении, что все входные аргументы-ключи равновероятны, алгоритм по скорости работы в случае успешного поиска аналогичен алгоритму Q; при неудачном же поиске отсутствие нужного ключа определяется примерно вдвое быстрее.

Во всех приведенных здесь алгоритмах использовалась запись с индексами для элементов таблиц (она более удобна для описания алгоритмов). Однако все описанные алгоритмы применимы и к другим типам данных, например к таблицам со *связанным* представлением данных, поскольку в них данные также расположены последовательно (см. упр. 2–4).

Частота обращений. До сих пор мы предполагали, что все аргументы поиска равновероятны. В общем случае это не так: вероятность запроса на поиск с ключом K_j равна p_j, причем $p_1 + p_2 + \cdots + p_N = 1$. Время, требуемое для успешного завершения поиска, пропорционально количеству сравнений C, которое имеет среднее значение

$$\bar{C}_N = p_1 + 2p_2 + \cdots + Np_N. \tag{3}$$

Если мы можем размещать записи в таблице в любом порядке, значение $\bar{C}_N$ минимально при

$$p_1 \geq p_2 \geq \cdots \geq p_N, \tag{4}$$

т. е. когда наиболее часто используемые записи находятся в начале таблицы.

Рассмотрим случаи различных распределений вероятностей и выясним, какой выигрыш может дать оптимальное расположение записей в таблице, указанное в (4). В случае равновероятного появления ключей $p_1 = p_2 = \cdots = p_N = 1/N$ формула (3) сводится к $\bar{C}_N = (N+1)/2$, т. е. к ранее полученному нами результату (2). Теперь предположим, что распределение вероятностей имеет вид

$$p_1 = \frac{1}{2}, \qquad p_2 = \frac{1}{4}, \qquad \ldots, \qquad p_{N-1} = \frac{1}{2^{N-1}}, \qquad p_N = \frac{1}{2^{N-1}}. \tag{5}$$

Согласно упр. 7 в данном случае $\bar{C}_N = 2 - 2^{1-N}$; при этом среднее количество сравнений, если записи расположены в надлежащем порядке, *меньше двух*.

Еще одно, клиновидное, распределение вероятностей определяется как

$$p_1 = Nc, \quad p_2 = (N-1)c, \quad \ldots, \quad p_N = c, \tag{6}$$

где $c = \frac{2}{N(N+1)}$. Здесь мы не получим такого эффекта, как при распределении (5). В случае клиновидного распределения

$$\bar{C}_N = c\sum_{k=1}^{N} k(N+1-k) = \frac{N+2}{3}, \tag{7}$$

и при оптимальном размещении записей экономится около трети времени, которое уходит на поиск, по сравнению со временем, необходимым при случайном размещении записей*.

Естественно, распределения (5) и (6) сугубо искусственны и не могут служить хорошим приближением реальных примеров. Более типично для реальных ситуаций распределение Зипфа:

$$p_1 = c/1, \quad p_2 = c/2, \quad \ldots, \quad p_N = c/N, \tag{8}$$

где $c = 1/H_N$. Оно получило известность благодаря работам Д. К. Зипфа (G. K. Zipf), который, исследуя естественные языки, обнаружил, что n-е по частоте употребления слово языка встречается с частотой, примерно обратно пропорциональной n [см. *The Psycho-Biology of Language* (Boston, Mass.: Houghton Mifflin, 1935); *Human Behavior and the Principle of Least Effort* (Reading, Mass.: Addison-Wesley, 1949)]. Аналогичные распределения встречаются, например, в таблицах переписи населения, в которых районы расположены в порядке убывания численности населения. В случае подчинения ключей в таблице закону Зипфа находим

$$\bar{C}_N = N/H_N. \tag{9}$$

Поиск в таком файле осуществляется примерно в $\frac{1}{2}\ln N$ раз быстрее, чем в неупорядоченном файле [см. A. D. Booth, L. Brandwood, and J. P. Cleave, *Mechanical Resolution of Linguistic Problems* (New York: Academic Press, 1958), 79].

Другое близкое к реальному распределение — это распределение, соответствующее правилу "80–20", которое часто наблюдается в коммерческих приложениях [см., например, W. P. Heising, *IBM Systems J.* **2** (1963), 114–115]. Это правило гласит, что 80% транзакций работают с 20% файла. Оно фрактально, т. е. оно применимо и к активным 20% файла; следовательно, 64% транзакций работают с 4% файла и т. д. Другими словами,

$$\frac{p_1 + p_2 + \cdots + p_{.20n}}{p_1 + p_2 + p_3 + \cdots + p_n} \approx .80 \tag{10}$$

для всех n. Вот одно из точно удовлетворяющих правилу распределений (при n, кратных 5):

$$p_1 = c, \quad p_2 = (2^\theta - 1)c, \quad p_3 = (3^\theta - 2^\theta)c, \quad \ldots, \quad p_N = \big(N^\theta - (N-1)^\theta\big)c, \tag{11}$$

где

$$c = 1/N^\theta, \qquad \theta = \frac{\log .80}{\log .20} = 0.1386. \tag{12}$$

Как вы можете убедиться, в этом случае для всех n $p_1 + p_2 + \cdots + p_n = cn^\theta$. Вероятности из (11) не очень удобны для работы; однако $n^\theta - (n-1)^\theta = \theta n^{\theta-1}\big(1 + O(1/n)\big)$,

---

* Имеется в виду случай равновероятного появления ключей. — *Прим. ред.*

и поэтому можно воспользоваться более простым распределением, приближенно удовлетворяющим правилу "80–20":

$$p_1 = c/1^{1-\theta}, \quad p_2 = c/2^{1-\theta}, \quad \ldots, \quad p_N = c/N^{1-\theta}, \tag{13}$$

где $c = 1/H_N^{(1-\theta)}$. Здесь, как и ранее, $\theta = \log .80/\log .20$, а $H_N^{(s)}$ — N-е гармоническое число порядка s, а именно — $1^{-s} + 2^{-s} + \cdots + N^{-s}$. Обратите внимание на схожесть этого распределения вероятности с законом Зипфа (8); с изменением θ от 1 до 0 закон распределения изменяется от равномерного к закону Зипфа. Применяя формулу (3) к распределению (13), получим среднее число сравнений для закона "80–20" (см. упр. 8):

$$\bar{C}_N = H_N^{(-\theta)}/H_N^{(1-\theta)} = \frac{\theta N}{\theta+1} + O(N^{1-\theta}) \approx 0.122N. \tag{14}$$

Изучая частоту употребления слов, Ю. С. Шварц (E. S. Schwartz) (см. интересный график на с. 422 в *JACM* **10** (1963)) предложил использовать в качестве более точного приближения распределение (13) с небольшими *отрицательными* значениями θ. Тогда значение

$$\bar{C}_N = H_N^{(-\theta)}/H_N^{(1-\theta)} = \frac{N^{1+\theta}}{(1+\theta)\zeta(1-\theta)} + O(N^{1+2\theta}) \tag{15}$$

получается существенно меньше, чем в случае (9) при $N \to \infty$.

Распределения, подобные (11) и (13), были впервые изучены Вильфредо Парето (Vilfredo Pareto) в связи с распределением богатства людей [*Cours d'Économie Politique* **2** (Lausanne: Rouge, 1897), 304–312]. Пусть p_k пропорционально состоянию k-го по богатству индивидуума, а p_N — состоянию более бедного индивидуума, занимающего в списке богатых N-е место. Тогда k/N представляет собой вероятность того, что состояние более богатого человека не менее чем в $x = p_k/p_N$ раз превосходит состояние более бедного. Таким образом, при $p_k = ck^{\theta-1}$ и $x = (k/N)^{\theta-1}$ описанная вероятность равна $x^{-1/(1-\theta)}$. Такое распределение в настоящее время называется *распределением Парето* с параметром $1/(1-\theta)$.

Любопытно, что Парето не понимал сути собственного распределения; он полагал, что значение θ, близкое к 0, соответствует более уравнительному обществу, чем значение, близкое к 1! Его ошибка была исправлена Коррадо Жини (Corrado Gini) [*Atti della III Riunione della Società Italiana per il Progresso delle Scienze* (1910), переиздана в его *Memorie di Metodologia Statistica* **1** (Rome, 1955), 3–120]. Жино был первым человеком, сформулировавшим и объяснившим важность соотношений, подобных закону "80–20" (10). Люди, как правило, не понимают сути таких распределений; они часто говорят о законе "75–25" или "90–10", как если бы главное, что придает смысл закону, заключалось в том, что в законе "a–b" выполняется равенство $a + b = 100$. Однако, как можно убедиться из (12), сумма $80 + 20$ здесь ни при чем...

Еще одно дискретное распределение, аналогичное (11) и (13), было предложено Д. Удни Юлом (G. Udny Yule) при изучении увеличения со временем количества биологических видов при различных моделях эволюции [*Philos. Trans.* **B213** (1924), 21–87]. Распределение Юла допускает значения $\theta < 2$:

$$p_1 = c, \; p_2 = \frac{c}{2-\theta}, \; p_3 = \frac{2c}{(3-\theta)(2-\theta)}, \; \ldots, \; p_N = \frac{(N-1)!\,c}{(N-\theta)\ldots(2-\theta)} = \frac{c}{\binom{N-\theta}{N-1}};$$

$$c = \frac{\theta}{1-\theta}\,\frac{\binom{N-\theta}{N}}{1-\binom{N-\theta}{N}}. \tag{16}$$

Граничные значения $c = 1/H_N$ и $c = 1/N$ получаются при $\theta = 0$ и $\theta = 1$.

Имеется еще одна часто цитируемая работа, однако ее популярность связана не с ее важностью, а лишь с тем, что это первая работа американского автора на эту тему [Alfred J. Lotka, *J Washington Academy of Sciences* **16** (1926), 317–323].

"Самоорганизующийся" файл. Приведенные вычисления вероятностей хороши, однако в большинстве случаев распределение вероятностей априори не известно. Можно было бы хранить в каждой записи счетчик обращений к ней и на основании полученных показаний переупорядочивать записи. Конечно, во многих ситуациях, как мы видели, такое переупорядочение приведет к значительной экономии времени; тем не менее зачастую не стоит выделять память для счетчиков — гораздо разумнее использовать ее, например, для технологии непоследовательного поиска, о чем будет рассказано ниже в данной главе.

Существует используемая многие годы простая схема, происхождение которой, увы, не известно. Эта схема позволяет получить неплохие результаты без привлечения счетчиков: когда запись успешно обнаружена, она перемещается в начало таблицы.

Идея этой "самоорганизующейся" технологии заключается в том, что наиболее часто используемые записи в результате будут располагаться в начале таблицы. Предположим, что N ключей встречаются среди аргументов поиска с вероятностями $\{p_1, p_2, \ldots, p_N\}$ и при этом каждый поиск совершается *независимо* от других. В таком предположении можно показать, что среднее количество сравнений, необходимых для поиска записи в таком самоорганизующемся файле стремится к предельному значению (см. упр. 11):

$$\widetilde{C}_N = 1 + 2\sum_{1\le i<j\le N}\frac{p_i p_j}{p_i+p_j} = \frac{1}{2} + \sum_{i,j}\frac{p_i p_j}{p_i+p_j}. \tag{17}$$

Например, если $p_i = 1/N$ при $1 \le i \le N$, самоорганизующаяся таблица будет находиться в неупорядоченном состоянии, а приведенная формула сведется к хорошо известному нам значению $(N+1)/2$. В общем случае полученное в (17) значение всегда меньше удвоенного оптимального значения из (3), так как $\widetilde{C}_N \le 1 + 2\sum_{j=1}^{N}(j-1)p_j = 2\bar{C}_N - 1$. В действительности $\widetilde{C}_N$ всегда меньше, чем $(\pi/2)\cdot\bar{C}_N$ [Chung, Hajela, and Seymour, *J. Comp. Syst. Sci.* **36** (1988), 148–157]. Эту константу нельзя улучшить, так как при p_j, пропорциональных $1/j^2$, достигается равенство.

Посмотрим, насколько хорошо этот метод работает при распределении вероятностей по закону Зипфа (8). В этом случае мы имеем:

$$\begin{aligned}\widetilde{C}_N &= \frac{1}{2} + \sum_{1\le i,j\le N}\frac{(c/i)(c/j)}{c/i+c/j} = \frac{1}{2} + c\sum_{1\le i,j\le N}\frac{1}{i+j}\\ &= \frac{1}{2} + c\sum_{i=1}^{N}(H_{N+i}-H_i) = \frac{1}{2} + c\sum_{i=1}^{2N}H_i - 2c\sum_{i=1}^{N}H_i\\ &= \tfrac{1}{2} + c\big((2N+1)H_{2N} - 2N - 2(N+1)H_N + 2N\big)\end{aligned}$$

$$= \tfrac{1}{2} + c\big(N \ln 4 - \ln N + O(1)\big) \approx 2N/\lg N \tag{18}$$

(см. формулы 1.2.7–(8) и 1.2.7–(3)). Полученная величина существенно лучше, чем $\frac{1}{2}N$ при достаточно больших N, и только в $\ln 4 \approx 1.386$ раз превышает количество сравнений при оптимальном размещении записей (см. (9)).

Вычислительные эксперименты с реальными таблицами символов компиляторов показали, что зачастую метод самоорганизации работает даже лучше, чем предсказывается; это связано с тем, что последовательные успешные поиски не являются независимыми и небольшие группы ключей зачастую вместе участвуют в поиске.

Впервые такая самоорганизующаяся схема была исследована Джоном Мак-Кэйбом (John McCabe) [*Operations Research* **13** (1965), 609–618], который и получил соотношение (17).

Мак-Кэйб предложил также другую интересную схему, при которой успешно обнаруженный ключ, не находящийся в начале таблицы, просто *меняется местами с предыдущим*, а не перемещается в начало таблицы. Он также установил, что предельное среднее время поиска для этого метода, в предположении независимости поисков, не превышает значения из (17). Несколькими годами позже Рональд Л. Ривест (Ronald L. Rivest) доказал, что метод перестановки при длительной работе использует строго меньше сравнений, чем метод перемещения в начало таблицы — естественно, исключая случаи, когда $N \leq 2$ или когда все ненулевые вероятности равны [*CACM* **19** (1976), 63–67]. Однако переход к асимптотическому пределу в этом случае происходит более медленно, чем при перемещении записей в начало таблицы [J. R. Bitner, *SICOMP* **8** (1979), 82–110]. Более того, Дж. Л. Бентли (J. L. Bentley), К. К. Мак-Геч (C. C. McGeoch), Д. Д. Слитор (D. D. Sleator) и Р. Е. Таржан (R. E. Tarjan) доказали, что при методе перемещения в начало таблицы количество обращений к памяти никогда не превысит более чем в четыре раза это количество для любого алгоритма при работе с линейными списками, заданного любой последовательностью обращений; методы же подсчета частот и перестановки таким свойством не обладают [*CACM* **28** (1985), 202–208, 404–411]. См. также *SODA* **8** (1997), 53–62, где приведены интересные результаты эмпирического изучения более 40 эвристических методов самоорганизующихся списков, проведенных Р. Бачрачем (R. Bachrach) и Р. Эль-Янивом (R. El-Yaniv).

Поиск на ленте с неравными записями. Рассмотрим теперь несколько иную задачу. Предположим, что таблица, в которой проводится поиск, хранится на ленте и при этом отдельные записи имеют различную длину. Примером такого хранения информации может служить лента системной библиотеки старых операционных систем. Стандартные системные программы (например такие, как компиляторы, ассемблеры, загружаемые подпрограммы и генераторы отчетов) являются "записями" на этой ленте, и большинство пользовательских заданий должно начинать выполнение с поиска необходимого программного обеспечения. Такая постановка задачи делает неприемлемым анализ алгоритма S, поскольку шаг S3 теперь выполняется за разные промежутки времени для разных записей. Поэтому мы не можем ограничиться в нашем анализе только количеством сравнений.

Пусть L_i — длина записи R_i и пусть p_i — вероятность того, что выполняется поиск именно этой записи. В таком случае среднее время поиска примерно

пропорционально

$$p_1L_1 + p_2(L_1 + L_2) + \cdots + p_N(L_1 + L_2 + L_3 + \cdots + L_N). \tag{19}$$

При $L_1 = L_2 = \cdots = L_N = 1$ это выражение сводится к уже исследованному нами случаю (3).

Представляется разумным и логичным разместить записи, обращения к которым происходят чаще, чем к другим, в начале ленты, однако в данном случае это вовсе не такая хорошая мысль, как кажется! Например, предположим, что у нас есть лента с двумя программами — A и B, причем обращения к A происходят в два раза чаще, чем к B, но при этом она в четыре раза длиннее. Тогда $N = 2$, $p_A = \frac{2}{3}$, $L_A = 4$, $p_B = \frac{1}{3}$, $L_B = 1$. Если мы поместим на ленту сначала A, а затем B, руководствуясь описанной "логикой", среднее время поиска станет равным $\frac{2}{3} \cdot 4 + \frac{1}{3} \cdot 5 = \frac{13}{3}$. Если же воспользоваться "нелогичным" решением, поместив на ленту сначала B, а затем A, среднее время поиска сократится до $\frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 5 = \frac{11}{3}$.

Оптимальное расположение информации на ленте (с точки зрения скорости поиска) можно определить, используя следующую теорему.

Теорема S. *Пусть L_i и p_i — числа, определенные выше. Размещение записей в таблице оптимально тогда и только тогда, когда*

$$p_1/L_1 \geq p_2/L_2 \geq \cdots \geq p_N/L_N. \tag{20}$$

Другими словами, минимальное значение

$$p_{a_1}L_{a_1} + p_{a_2}(L_{a_1} + L_{a_2}) + \cdots + p_{a_N}(L_{a_1} + \cdots + L_{a_N})$$

по всем перестановкам $a_1\, a_2 \ldots a_N$ из множества $\{1, 2, \ldots, N\}$ равно (19) тогда и только тогда, когда выполняется условие (20).

Доказательство. Предположим, что мы поменяли местами на ленте R_i и R_{i+1}. Тогда значение величины (19) станет равным не

$$\cdots + p_i(L_1 + \cdots + L_{i-1} + L_i) + p_{i+1}(L_1 + \cdots + L_{i+1}) + \cdots,$$

а

$$\cdots + p_{i+1}(L_1 + \cdots + L_{i-1} + L_{i+1}) + p_i(L_1 + \cdots + L_{i+1}) + \cdots.$$

При этом изменение будет равно $p_iL_{i+1} - p_{i+1}L_i$. Если предположить оптимальность размещения (19), то любая перемена мест двух соседних записей должна приводить к увеличению времени работы, т. е. $p_i/L_i > p_{i+1}/L_{i+1}$. Таким образом, поскольку из оптимальности размещения следует набор неравенств (20), нами доказана необходимость условия (20) для оптимального размещения.

Докажем теперь достаточность выполнения условия (20) для оптимальности размещения. Приведенные выше рассуждения доказывают "локальную оптимальность" расположения — в том смысле, что любая перестановка двух рядом стоящих записей приведет к увеличению среднего времени работы. Однако это не доказывает невозможности сложного многоступенчатого обмена для улучшения производительности поиска, не доказывает, так сказать, "глобальной оптимальности". Мы рассмотрим два доказательства, в одном из которых используются знания из области компьютерных наук, а другое основано на некоторой математической хитрости.

Первое доказательство. Предположим, что условие (20) выполнено. Известно, что любую перестановку записей можно привести к "рассортированному" состоянию, т. е. привести ее к виду $R_1 R_2 \dots R_N$ с использованием только перестановок двух соседних элементов. Каждая из таких перестановок заменяет $\dots R_j R_i \dots$ на $\dots R_i R_j \dots$ для некоторых $i < j$, тем самым уменьшая время поиска на неотрицательную величину $p_i L_j - p_j L_i$. Следовательно, порядок расположения записей $R_1 R_2 \dots R_N$ должен иметь минимальное время поиска, т. е. быть оптимальным.

Второе доказательство. Заменим каждую вероятность p_i на

$$p_i(\epsilon) = p_i + \epsilon^i - (\epsilon^1 + \epsilon^2 + \cdots + \epsilon^N)/N, \tag{21}$$

где ϵ — очень малое положительное число. При этом равенство $x_1 p_1(\epsilon) + \cdots + x_N p_N(\epsilon) = y_1 p_1(\epsilon) + \cdots + y_N p_N(\epsilon)$ справедливо только в том случае, когда $x_1 = y_1$, $\dots$, $x_N = y_N$. Отсюда, в частности, следует, что в (20) равенство никогда не будет выполняться. Рассмотрим $N!$ перестановок записей. Среди них есть, по меньшей мере, одна оптимальная, которая в соответствии с первой частью доказательства удовлетворяет условию (20). Однако поскольку теперь в условии (20) равенства невозможны, то и оптимальная перестановка может быть только одна. Следовательно, условие (20) однозначно определяет некоторую оптимальную перестановку для вероятностей $p_i(\epsilon)$, причем ϵ достаточно мало. Исходя из непрерывности тот же порядок должен быть оптимален и при $\epsilon = 0$. (Такой тип доказательств нередко используется в комбинаторной оптимизации.)

Теорема S была доказана В. И. Смитом (W. E. Smith) [*Naval Research Logistics Quarterly* **3** (1956), 59–66]. В приведенных ниже упражнениях содержатся дополнительные результаты оптимальной организации файлов.

УПРАЖНЕНИЯ

1. [*M20*] Пусть все ключи поиска равновероятны. Определите стандартное среднеквадратичное отклонение числа сравнений при успешном последовательном поиске в таблице с N записями.

2. [*15*] Измените алгоритм S для использования связанных записей вместо индексов. (Если P указывает на запись в таблице, полагаем, что KEY(P) — ключ, INFO(P) — связанная с ключом информация и LINK(P) — указатель на следующую запись. Полагаем также, что FIRST указывает на первую запись, а последняя запись указывает на Λ.)

3. [*16*] Напишите MIX-программу для алгоритма из упр. 2. Чему равно время выполнения программы (с использованием C и S из (1))?

▶ **4.** [*17*] Можно ли использовать идею алгоритма Q для таблиц в виде связанных записей (см. упр. 2)?

5. [*20*] Программа Q′ выполняется существенно быстрее программы Q при больших значениях C. Существуют ли значения C и S, при которых программа Q′ будет выполняться дольше, чем программа Q?

▶ **6.** [*20*] Добавьте три инструкции в программу Q′, которые позволят ей выполняться за время около $(3.33C + \text{constant})u$.

7. [*M20*] Определите среднее число сравнений (3) в случае "бинарного" распределения вероятности (5).

8. [*HM22*] Найдите асимптотический ряд для $H_n^{(x)}$ при $n \to \infty$; $x \neq 1$.

▸ **9.** [*HM28*] В тексте отмечено, что распределения вероятностей, данные в (11), (13) и (16), приблизительно одинаковы при $0 < \theta < 1$ и что среднее число сравнений с использованием (13) равно $\frac{\theta}{\theta+1}N + O(N^{1-\theta})$.

a) Означает ли это, что число сравнений равно $\frac{\theta}{\theta+1}N + O(N^{1-\theta})$ при использовании распределения (11)?

b) Верно ли это утверждение для распределения (16)?

c) Сравните (11) и (16) с (13) при $\theta < 0$.

10. [*M20*] Наилучшее расположение записей в последовательной таблице определяется условием (4). А что собой представляет *наихудшее* расположение? Покажите, что имеется простое соотношение между средним количеством сравнений при наилучшем и наихудшем размещениях записей.

11. [*M30*] Цель этого упражнения заключается в анализе предельного поведения самоорганизующегося файла при использовании эвристического метода перемещения записи в начало файла. Сначала введем некоторые обозначения. Пусть $f_m(x_1, x_2, \ldots, x_m)$ равно бесконечной сумме всех различных упорядоченных произведений $x_{i_1} x_{i_2} \ldots x_{i_k}$, таких, что $1 \le i_1, \ldots, i_k \le m$, причем каждое $x_1, x_2, \ldots, x_m$ входит во все произведения. Например,

$$f_2(x,y) = \sum_{j,k\ge 0} (x^{1+j}y(x+y)^k + y^{1+j}x(x+y)^k) = \frac{xy}{1-x-y}\left(\frac{1}{1-x} + \frac{1}{1-y}\right).$$

Исходя из множества X из n переменных $\{x_1, \ldots, x_n\}$, положим

$$P_{nm} = \sum_{1\le j_1<\cdots<j_m\le n} f_m(x_{j_1}, \ldots, x_{j_m}); \qquad Q_{nm} = \sum_{1\le j_1<\cdots<j_m\le n} \frac{1}{1-x_{j_1}-\cdots-x_{j_m}}.$$

Например, $P_{32} = f_2(x_1,x_2) + f_2(x_1,x_3) + f_2(x_2,x_3)$ и $Q_{32} = 1/(1-x_1-x_2) + 1/(1-x_1-x_3) + 1/(1-x_2-x_3)$. По определению полагаем $P_{n0} = Q_{n0} = 1$.

a) Предположим, что в самоорганизующийся файл запросы на поиск элемента R_i поступают с вероятностью p_i. Покажите, что после достаточно длительной работы системы элемент R_i оказывается на m-м месте с предельной вероятностью $p_i P_{(N-1)(m-1)}$, где множество X представляет собой $\{p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_N\}$.

b) Суммируя результат (a) для $m = 1, 2, \ldots$, получаем тождество

$$P_{nn} + P_{n(n-1)} + \cdots + P_{n0} = Q_{nn}.$$

Получите отсюда следующие соотношения:

$$P_{nm} + \binom{n-m+1}{1} P_{n(m-1)} + \cdots + \binom{n-m+m}{m} P_{n0} = Q_{nm};$$

$$Q_{nm} - \binom{n-m+1}{1} Q_{n(m-1)} + \cdots + (-1)^m \binom{n-m+m}{m} Q_{n0} = P_{nm}.$$

c) Вычислите предельное среднее расстояние $d_i = \sum_{m\ge 1} m p_i P_{N-1,m-1}$ записи R_i от начала таблицы; затем вычислите $\widetilde{C}_N = \sum_{i=1}^N p_i d_i$.

12. [*M23*] Используйте (17) для вычисления среднего количества сравнений, необходимого для поиска в самоорганизующемся файле при бинарном распределении вероятностей ключей поиска (5).

13. [*M27*] Используя (17), вычислите $\widetilde{C}_N$ для распределения вероятностей (6).

14. [*M21*] Даны две последовательности действительных чисел — $\langle x_1, x_2, \ldots, x_n\rangle$ и $\langle y_1, y_2, \ldots, y_n\rangle$. Какая перестановка индексов $a_1\, a_2 \ldots a_n$ делает сумму $\sum_i x_i y_{a_i}$ максимальной, минимальной?

▸ **15.** [*M22*] В тексте было показано, как оптимально расположить программы на ленте системной библиотеки для поиска только одной программы. Однако при работе с библиотекой *подпрограмм*, когда для работы программы пользователя необходимо загрузить различные подпрограммы, следует принять другой набор предположений.

В этом случае предположим, что запрос на подпрограмму j поступает с вероятностью P_j, причем вызовы различных подпрограмм независимы. Тогда, например, вероятность того, что не потребуется ни одна подпрограмма, равна $(1-P_1)(1-P_2)\dots(1-P_N)$, а вероятность того, что поиск прекратится после загрузки j-й подпрограммы, равна $P_j(1-P_{j+1})\dots(1-P_N)$. Если L_j — длина j-й подпрограммы, то среднее время поиска будет пропорционально

$$L_1P_1(1-P_2)\dots(1-P_N)+(L_1+L_2)P_2(1-P_3)\dots(1-P_N)+\cdots+(L_1+\cdots+L_N)P_N.$$

Каким в этом случае должно быть оптимальное расположение подпрограмм на ленте?

16. [*M22*] (Г. Ризель (H. Riesel).) Зачастую необходимо проверить, выполняются ли одновременно n заданных условий. (Например, может понадобиться проверить, что $x>0$ и $y<z^2$, и при этом не ясно, какое условие должно проверяться первым.) Предположим, что проверка j-го условия занимает T_j единиц времени и что условие выполняется с вероятностью p_j (причем эта вероятность не зависит от результатов выполнения других условий). В каком порядке должны выполняться проверки?

17. [*M23*] (В. И. Смит (W. E. Smith).) Предположим, имеется n заданий; j-е задание занимает T_j единиц времени; крайний срок его выполнения — D_j. Другими словами, j-е задание должно быть выполнено не позже момента D_j. Какое расписание работ $a_1\,a_2\dots a_n$ минимизирует *максимальное запаздывание*, т. е.

$$\max(T_{a_1}-D_{a_1},\,T_{a_1}+T_{a_2}-D_{a_2},\,\dots,\,T_{a_1}+T_{a_2}+\cdots+T_{a_n}-D_{a_n})\,?$$

18. [*M30*] (*Сцепленный поиск.*) Предположим, что N записей расположены в памяти в виде линейного массива $R_1\dots R_N$ и вероятность поиска записи R_j равна p_j. Поиск называется "сцепленным", если каждый последующий поиск начинается с того места, где завершился предыдущий. Если последовательные поиски независимы, то среднее время поиска составляет $\sum_{1\le i,j\le N} p_ip_jd(i,j)$, где $d(i,j)$ — время, которое необходимо для поиска, начинающегося в позиции i и заканчивающегося в позиции j. Эта модель может быть применима, например, ко времени поиска дискового файла (при этом $d(i,j)$ представляет собой время перемещения между i- и j-м цилиндрами диска).

Цель данного упражнения — описать оптимальное размещение записей для сцепленного поиска в случае, когда $d(i,j)$ представляет собой монотонно возрастающую функцию от $|i-j|$, т. е. $d(i,j)=d_{|i-j|}$ и $d_1<d_2<\cdots<d_{N-1}$ (величина d_0 не существенна). Докажите, что размещение будет оптимально тогда и только тогда, когда будет выполняться условие либо $p_1\le p_N\le p_2\le p_{N-1}\le\cdots\le p_{\lfloor N/2\rfloor+1}$, либо $p_N\le p_1\le p_{N-1}\le p_2\le\cdots\le p_{\lceil N/2\rceil}$. (Другими словами, наилучшее расположение — в виде "органных труб", показанных на рис. 2.) *Указание.* Рассмотрите расположение, которому соответствуют вероятности $q_1\,q_2\dots q_k\,s\,r_k\dots r_2\,r_1\,t_1\dots t_m$ для некоторых $m\ge 0$ и $k>0$; $N=2k+m+1$. Покажите, что расположение $q_1'\,q_2'\dots q_k'\,s\,r_k'\dots r_2'\,r_1'\,t_1\dots t_m$ лучше, если $q_i'=\min(q_i,r_i)$ и $r_i'=\max(q_i,r_i)$, за исключением случая, когда $q_i'=q_i$ и $r_i'=r_i$ для всех i, и случая $q_i'=r_i$, $r_i'=q_i$ и $t_j=0$ для всех i и j. То же самое справедливо при отсутствии s и $N=2k+m$.

19. [*M20*] Продолжая выполнять упр. 18, найдите оптимальное расположение для сцепленного поиска, если функция $d(i,j)$ обладает следующим свойством: $d(i,j)+d(j,i)=c$ для всех $i\ne j$. (Такая ситуация возможна, например, при поиске на ленте без возможности обратной перемотки и с неизвестным нам направлением поиска; для $i<j$ имеем $d(i,j)=$

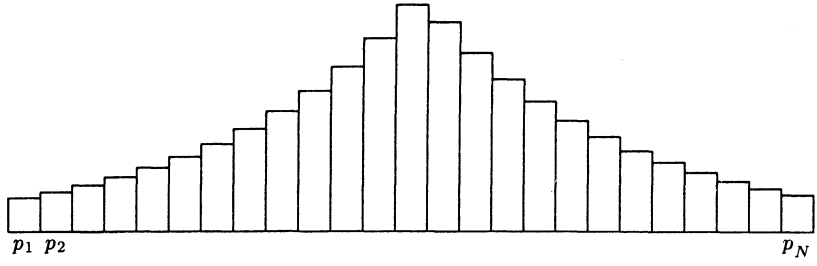

Рис. 2. Расположение в виде "органных труб" минимизирует среднее время сцепленного поиска.

$a + b(L_{i+1} + \cdots + L_j)$ и $d(j,i) = a + b(L_{j+1} + \cdots + L_N) + r + b(L_1 + \cdots + L_i)$, где r — время перемотки.)

20. [*M28*] Продолжая выполнять упр. 18, найдите оптимальное расположение для сцепленного поиска в случае, когда $d(i,j) = \min(d_{|i-j|}, d_{n-|i-j|})$ и $d_1 < d_2 < \cdots$. (Эта ситуация встречается, например, в двусвязном циклическом списке или в запоминающем устройстве с возможностью перемещения в обе стороны.)

21. [*M28*] Рассмотрим n-мерный куб с координатами вершин $(d_1, \ldots, d_n)$, где $d_j = 0$ или 1; две вершины называются *соседними*, если они различаются только одной координатой. Предположим, что набор из 2^n чисел $x_0 \le x_1 \le \cdots \le x_{2^n-1}$ должен быть сопоставлен 2^n вершинам таким образом, чтобы минимизировать сумму $\sum_{i,j} |x_i - x_j|$; сумма берется по всем i и j, таким, что x_i и x_j сопоставлены соседним вершинам. Докажите, что этот минимум достигается тогда, когда для всех j x_j сопоставлено вершине, координаты которой являются двоичным представлением числа j.

▸ **22.** [*20*] Предположим, что в большом файле вам необходимо найти 1 000 *ближайших* к данному ключу записей, т. е. записей, для которых функция расстояния $d(K_j, K)$ принимает наименьшие значения. Какая структура данных будет самой подходящей для такого последовательного поиска?

Пытайся до конца, сомнений прочь оскал,
И, все преодолев, найдешь ты, что искал.*

— РОБЕРТ ХЕРРИК (ROBERT HERRICK),
Ищите и обрящете (Seeke and finde) (1648)

---

* Перевод Светланы Тригуб.