ФЕДЕРАЛЬНОЕ АГЕНСТВО ПО ОБРАЗОВАНИЮ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

В. А. ЦИКОЗА, Т. Г. ЧУРИНА

МЕТОДЫ ПРОГРАММИРОВАНИЯ: ПЕРЕСТАНОВКИ, ПОИСК И СОРТИРОВКА

Учебное пособие

Часть 2

Новосибирск

2006

УДК 519.6
ББК В185
Ц 598

Цикоза В. А., Чурина Т. Г. Методы программирования: перестановки, поиск и сортировка: Учеб. пособие / Новосиб. гос. ун-т. Новосибирск, 2006. Ч. 2. 59 с.

Данное учебное пособие составлено к курсу «Программирование на языке высокого уровня». Цель данного курса заключается в знакомстве с типовыми задачами программирования и основными моделями и методами их решения, на примере которых дается представление об искусстве программирования. В пособии рассмотрены вопросы, касающиеся таких важных понятий, как перестановки, поиск и сортировки, приведены оценки сложности рассмотренных алгоритмов.

Пособие подготовлено на кафедре систем информатики факультета информационных технологий. Предназначено для студентов 1-го курса этого факультета, а также может быть полезно студентам 1-го курса механико-математического факультета и слушателям межфакультетского спецкурса «Разработка сложных программ и методы программирования» и рекомендуется студентам 1—2-го курсов Высшего колледжа информатики НГУ.

Рецензент
канд. физ.-мат. наук Л. В. Городняя

Оглавление

1. Перестановки

Перестановкой порядка N называется расположение N различных объектов в ряд в некотором порядке. Например, для трех объектов — a, b и c — существует шесть перестановок: abc, acb, bac, bca, cab, cba. Для множества из N элементов можно построить $N!$ различных перестановок: первую позицию можно занять N способами, вторую — $N-1$ способом, так как один элемент уже занят, и т. д. На последнее место можно поставить только один оставшийся элемент. Следовательно, общее количество вариантов расстановки равно $N \cdot (N-1) \cdot (N-2) \cdot ... \cdot 1 = N!$.

Перестановки важны при изучении алгоритмов сортировки, так как они хорошо формализуют варианты представления неупорядоченных исходных данных. Для исследования эффективности различных методов сортировки нужно уметь подсчитывать число перестановок, которые заставляют повторять некоторый шаг алгоритма определенное число раз.

Прежде чем рассмотреть основные методы получения всех перестановок из N объектов, рассмотрим некоторые важные понятия.

1.1. Инверсии

Пусть даны базовое множество из N элементов $1, 2, 3, ..., N$ и его перестановка $a_1, a_2, ..., a_N$. Пара (a_i, a_j) называется *инверсией* перестановки, если $a_i > a_j$ при $i < j$. Например, перестановка 4, 1, 3, 2 имеет четыре инверсии: $(4,1)$, $(3,2)$, $(4,3)$ и $(4,2)$. Единственной перестановкой, не содержащей инверсий, является упорядоченная перестановка $1, 2, 3, ..., N$. Таким образом, каждая инверсия — это пара элементов перестановки, нарушающих ее упорядоченность.

Таблицей инверсий перестановки $a_1, a_2, ..., a_N$ называется последовательность чисел $b_1, b_2, ..., b_N$, где b_j есть число элементов перестановки, больших j и расположенных левее j (т. е. число инверсий вида (x, j)). Например, для перестановки

$$5\ 9\ 1\ 8\ 2\ 6\ 4\ 7\ 3 \tag{1}$$

таблица инверсий будет следующая:

$$2\ 3\ 6\ 4\ 0\ 2\ 2\ 1\ 0. \tag{2}$$

По определению таблицы инверсий ее элементы удовлетворяют соотношениям $0 \le b_j \le N - j$, $b_N = 0$.

Утверждение. *Таблица инверсий единственным образом определяет соответствующую перестановку.* Построим перестановку (1) по таблице инверсий (2). Создадим заготовку перестановки из одного максимального числа 9, на последующих шагах будем ее достраивать. Восьмой элемент таблицы содержит число 1, т. е. перед числом 8 в перестановке стоит один элемент, больший 8, следовательно, в перестановке число 8 находится после 9. Вставляем 8 после 9: $9, 8$. Перед числом 7 в перестановке два элемента, больших 7 (так как в седьмой позиции таблицы находится 2), следовательно, 7 находится в перестановке после $9, 8$. Получаем $9, 8, 7$. Аналогично, перед числом 6 два элемента больших, чем оно само: следовательно, 6 находится между числами 8 и 7: $9, 8, 6, 7$. Перед 5 нет элементов, больших, чем 5, следовательно, получим $5, 9, 8, 6, 7$. Перед 4 — четыре элемента, результат — $5, 9, 8, 6, 4, 7$. Процесс продолжается до тех пор, пока не будут рассмотрены все элементы таблицы.

Проиллюстрированный процесс формально описывается следующим алгоритмом.

Алгоритм А1:

- вход: $N > 0$ – число элементов, $b_1, b_2, ..., b_N$ – таблица инверсий, $0 \le b_j \le N - j$;
  $P :=$ пустая последовательность;
  цикл по j от N вниз до 1
  вставить число j в P после элемента b_j;
  конец цикла;
- выход: P – перестановка, соответствующая данной таблице инверсий.

Заметим, что если поменять местами соседние элементы перестановки, то общее число инверсий увеличится или уменьшится на единицу. Для анализа некоторых алгоритмов сортировки необходимо знать число перестановок N элементов, содержащих ровно k инверсий.

Рассмотрим теперь основные методы построения перестановок из N объектов.

1.2. Поиск следующей по алфавиту перестановки

Пусть даны две перестановки $b_1, b_2, ..., b_N$ и $c_1, c_2, ..., c_N$ набора $1, 2, ..., N$. *Перестановка b предшествует перестановке c в алфавитном (лексикографическом) порядке*, если для минимального значения k, такого что

$b_k \neq c_k$, справедливо $b_k < c_k$. Например, перестановка 12345 предшествует перестановке 12453 (здесь $k = 3$).

Первой перестановкой в алфавитном порядке всегда является перестановка $1, 2, 3, ..., N-1, N$, а последней — $N, N-1, ..., 1$.

Алфавитное упорядочение перестановок аналогично принятому алфавитному упорядочению слов в словарях. Данное выше описание алфавитного порядка можно обобщить для слов из произвольного алфавита A, имеющих произвольную длину. Для этого необходимо определить линейный порядок на символах из алфавита A и дополнить условие для случая сравнения слов разной длины, например, так: если слово b является началом слова c, то $b < c$.

Возникает следующая идея перебора всех перестановок длины N: определить каким-либо образом функцию, которая по заданной перестановке выдает непосредственно следующую за ней в алфавитном порядке, и применять ее последовательно к собственным результатам начиная с самой первой перестановки, пока не будет получена последняя. Эту идею реализует алгоритм Дейкстры [3].

Например, для перестановки

$$1\ 4\ 6\ 2\ 9\ 5\ 8\ 7\ 3. \tag{3}$$

следующей по алфавиту является перестановка

$$1\ 4\ 6\ 2\ 9\ 7\ 3\ 5\ 8.$$

Алгоритм А2:

• вход :N>0—число элементов; $a_1, a_2, ..., a_N$ – предыдущая перестановка.

1. Просматривая перестановку начиная с последнего элемента, найдем такой номер i, что $a_{i+1} > ... > a_{N-1} > a_N$ и $a_i < a_{i+1}$. Если такого i нет, то последовательность упорядочена по убыванию и следующей перестановки нет: конец алгоритма.

Пояснение. Тот факт, что исходная последовательность не является последней в алфавитном порядке, гарантирует нам существование такого i. Для перестановки (3) $i = 6$. Смысл этого шага в том, что из всех перестановок с началом $a_1, a_2, ..., a_i$ данная перестановка является последней в алфавитном порядке, так как ее хвост $a_{i+1}, ..., a_N$ упорядочен по убыванию и образует последнюю в алфавитном порядке перестановку элементов $a_{i+1}, ..., a_N$.

2. После того как номер i найден, требуется найти элемент, который встанет на i-е место в следующей перестановке. Поскольку в «хвосте»

$a_{i+1}, ..., a_N$ существует элемент a_j, который больше a_i (по построению в шаге 1), то, обменяв его местами с a_i, мы получим одну из следующих по порядку перестановок. При нескольких возможных a_j надо взять минимальный из них. Таким образом, поскольку мы ищем непосредственного преемника по порядку, мы должны найти такой номер j в интервале $i+1 \le j \le N$, что a_j есть наименьшее значение, удовлетворяющее условию $a_j > a_i$. После этого надо переставить a_i и a_j. Так как хвост перестановки, в силу условия шага 1, упорядочен по убыванию, искомый элемент a_j является первым элементом с конца, для которого выполнено условие $a_j > a_i$.

В нашем примере в таким элементом a_j будет $a_8 = 7$. Переставив a_6 и a_8, мы получим перестановку $1, 4, 6, 2, 9, 7, 8, 5, 3$.

3. Последний шаг состоит в том, чтобы из элементов нового хвоста $a_{i+1}, ..., a_N$ составить минимальную в алфавитном порядке перестановку. Для этого достаточно упорядочить хвост по возрастанию. Однако после шага 1 элементы хвоста уже образовывали строго убывающую последовательность, а перестановка на шаге 2 эту упорядоченность не нарушила, так как $a_{j-1} > a'_j = a_i > a_{j+1}$. Поэтому для упорядочивания элементов хвоста по возрастанию вместо квадратичной по времени выполнения сортировки можно применить линейную процедуру инвертирования последовательности – обменов местами элементов a_{i+1+k} с элементами a_{N-k}.

В нашем примере результат этой операции — искомая перестановка $1, 4, 6, 2, 9, 7, 3, 5, 8$.

1.3. Инверсионный метод поиска всех перестановок

Идею поиска «следующего по алфавиту» можно применить для перебора перестановок и другим способом. Из разд. 1.1 мы знаем, что таблица инверсий однозначно определяет перестановку. Верно и обратное: каждая перестановка имеет только одну таблицу инверсий. Следовательно, если мы сумеем перебрать все таблицы инверсий, то с помощью алгоритма $A1$ сможем перебрать все перестановки.

Таблицы инверсий же можно перебрать следуюшим образом. Рассмотрим таблицу инверсий как N-значное число в такой необычной «системе счисления»: количество цифр, которое можно использовать в i-м разряде (с конца, начиная с 0) равно i. Возьмем «минимальную» таблицу и будем последовательно прибавлять к ней как к числу единицу, пользуясь, например, алгоритмом сложения с переносом для много-

разрядных чисел, модифицированным для нашей «системы счисления». Мы получим такую последовательность:

$$\begin{array}{c} 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0 \\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0 \\ 0\ 0\ 0\ 0\ 0\ 0\ 2\ 0\ 0 \\ \ldots \\ 8\ 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0 \end{array}$$

.

Нетрудно убедиться, что каждое полученное «число» представляет корректную таблицу инверсий, что все они различны и что их общее количество — $N!$. Следовательно, они соответствуют всем возможным перестановкам. Можно заметить также, что выписанные «числа» находятся в возрастающем лексикографическом порядке, если рассматривать их как слова. Проиллюстрированный алгоритм «прибавления единицы» можно, таким образом, рассматривать как корректный метод получения «следующей по алфавиту» таблицы инверсий.

1.4. Алгоритм Кнута поиска всех перестановок

Опишем еще один алгоритм, предложенный Кнутом в работе [1], в котором используется индуктивный способ получения перестановок. Для **каждой** перестановки $a_1, a_2, ..., a_{N-1}$ из элементов множества $\{1, 2, 3, .., N-1\}$ образуем N других перестановок из элементов множества $\{1, 2, 3, .., N-1, N\}$ следующим образом:

1) построим ряд

$$a_1, a_2, .., a_{N-1}, \frac{1}{2};\quad a_1, a_2, .., a_{N-1}, \frac{3}{2}\ ; ..;\ a_1, a_2, .., a_{N-1}, \frac{N-1}{2};$$

2) сохраняя порядок элементов, перенумеруем их числами $1, 2, 3, .., N$ в порядке возрастания.

Для множества $1, 2, 3$ перестановками будут последовательности $1, 2, 3$; $1, 3, 2$; $2, 1, 3$; $2, 3, 1$; $3, 1, 2$; $3, 2, 1$. Например, в результате выполнения первого шага алгоритма для перестановки $2, 3, 1$ мы получим ряд

$$2, 3, 1, \frac{1}{2};\quad 2, 3, 1, \frac{3}{2};\quad 2, 3, 1, \frac{5}{2};\quad 2, 3, 1, \frac{7}{2}.$$

Перенумеровав элементы полученного ряда в порядке возрастания, получим следующие перестановки:

$$3\ \ 4\ \ 2\ \ 1; \quad 3\ \ 4\ \ 1\ \ 2; \quad 2\ \ 4\ \ 1\ \ 3; \quad 2\ \ 3\ \ 1\ \ 4.$$

В работе [1] Кнут приводит описание еще одного алгоритма конструирования новых перестановок. Он состоит в том, чтобы взять перестановку $a_1, a_2, .., a_{N-1}$ и число $k, 1 \leq k \leq N$. К каждому a_j, значение которого больше либо равно k, надо добавить 1. В результате получится перестановка $b_1, b_2, .., b_{N-1}$ из элементов множества $1, .., k-1, k+1, .., N$. Затем нужно создать перестановку $b_1, b_2, ..., b_{N-1}, k$ из элементов множества $1, 2, .., N$. Каждая перестановка в соответствии с этим алгоритмом получится только один раз. Можно использовать этот метод, помещая k не справа, а слева или же в любом другом фиксированном месте.

1.5. Рекурсивный метод поиска всех перестановок

Метод рекурсивного перебора перестановок основан на идее сведения исходной задачи к аналогичной задаче на меньшем наборе входных данных. Вспомним неформальное соображение, использованное при подсчете общего числа перестановок: поставим на первое место поочередно каждый из данных N элементов множества; тогда оставшиеся $N-1$ мест можно заполнить всевозможными перестановками оставшихся $N-1$ элементов. На основании этого можно выписать систему рекуррентных соотношений, определяющих множество $Per(M)$ всех перестановок базового множества M произвольной природы:

$$\begin{array}{l} Per(\varnothing) = \{\text{“ ”}\}, \\ Per(M) = \bigcup_{i \in M} Permut(i, M \backslash \{i\}), \\ Permut(i, S) = \{s + \text{“}i\text{”} \mid s \in Per(S)\}. \end{array} \tag{4}$$

Первое равенство задает условие обрыва рекурсивного спуска: пустое множество элементов порождает пустую перестановку. Два последних равенства определяют правила рекурсивного перехода. Эту систему можно переписать, приведя ее к одноуровневой рекурсии:

$$\begin{array}{l} Per(M) = Permut(\text{“ ”}, M), \\ Permut(p, S) = \bigcup_{i \in S} Permut(p + \text{“}i\text{”}, S \backslash \{i\}), \\ Permut(p, \varnothing) = \{p\}. \end{array} \tag{5}$$

Второе равенство задает правило перехода, а третье — условие обрыва. Здесь первый параметр функции *Permut* «накапливает» уже построенное начало перестановки, а второй содержит множество оставшихся элементов.

При описании алгоритма, реализующего построения по формулам (5), участвующие множества и готовые части перестановок удобно представить строками символов, поскольку над строковым типом данных определены интересующие нас операции выборки и сцепления (конкатенации). На языке Си этот процесс можно описать следующим образом:

```
typedef char string[10];
void process(string s)
{
    printf("%s\n", s);
}

void permut(string start, string rest)
{   int lenr = strlen(rest);
    int lens = strlen(start);
    int i=0;
    string s1="";
    string s2="";
    if (lenr==0)
       process(start);
    else {
       for (i=0; i<lenr; i++) {
       /* Добавляем i-ый символ к строке start */
           strcpy(s1,start);
           strncpy(s1+lens,rest+i,1);
           strncpy(s1+lens+1,"\0",1);
       /* Удаляем i-ый символ из строки rest */
           strncpy(s2,rest,i);
           strncpy(s2+i,rest+i+1,lenr-i-1);
           strncpy(s2+lenr-1,"\0", 1);
       /* Рекурсивный переход */
           permut( s1, s2 );
      }
   }
}
```

Вызывать эту процедуру следует так: $permut(\text{“ ”}, s)$; строка s изначально представляет собой базовое множество с произвольным порядком задания элементов. Оно передается в *rest*. При главном вызове *permut* в *start* передается пустая строка, соответствующая пустой построенной части перестановки. В дальнейшем символы из *rest* по одному перекачиваются в *start*, а ветвящаяся рекурсия обеспечивает последовательные спуск и возвраты с откатом обработанных вариантов. Обратим внимание, что готовые перестановки получаются на самом глубоком уровне рекурсии в функции *permut*: для их обработки вызывается функция *process* (в данном примере просто выводящая очередную перестановку).

1.6. Оценка эффективности алгоритмов перебора перестановок

Для множества из N элементов количество перестановок равно $N!$, следовательно, время работы любого алгоритма перебора как функция от N пропорционально $N!$. Коэффициент пропорциональности, определяющий затраты времени на порождение перестановки, во всех рассмотренных алгоритмах может быть оценен некоторой константой. Таким образом,

$$T_{\text{перебор перестановок}}(N) \leq c \cdot N!$$

2. Поиск

2.1. Задача поиска

Задача поиска состоит в том, чтобы найти определенный объект в некоторой совокупности объектов, т. е. установить факт его существования или отсутствия и, возможно, местоположение.

Рассматриваемая совокупность считается либо изначально линейно упорядоченной, либо линейно упорядочиваемой в процессе просмотра. Различаются структуры *прямого доступа* (массивы, строки), любой элемент которых может быть выбран непосредственно по заданному адресу (номеру, индексу), и структуры *последовательного доступа* (файлы, списки), элементы которых доступны только после обращения к предыдущим элементам. Методы поиска (и сортировки) для этих структур различны. В дальнейшем, если не оговорено иное, рассматриваемая совокупность будет считаться линейным массивом.

Объекты в общем случае рассматриваются как записи произвольной природы, однако имеющие в своей структуре один и тот же *ключ* — поле, содержащее значение, которое сравнивается в процессе поиска с искомым ключом. В более общем случае ключ можно рассматривать как числовую функцию, которая строит значение ключа на основании сколь угодно сложного анализа всех данных, представленных в записи. Далее при рассмотрении методов поиска и сортировки мы для простоты будем отождествлять записи с их ключами.

Следующие описания структур данных будут использоваться в дальнейших алгоритмах:

```
/* описания глобальных констант, типов и данных */
#define N 100     /* размер входного массива */
typedef int key;  /* тип ключа */
static key a[N];  /* входной массив */
```

Обратим внимание, что индексы элементов массива в Си находятся в интервале $0..N-1$. Соответственно индексы в приводимых программах на языке Си на единицу меньше приводимых при обсуждении методов.

2.2. Последовательный поиск

Идея последовательного поиска очевидна. Она состоит в том, чтобы, начав просмотр с первого элемента массива, продвигаться дальше

до тех пор, пока не будет найден нужный элемент или пока не будут просмотрены все элементы массива.

Следующая функция осуществляет поиск ключа x в массиве a длины N и выдает значением индекс его первого вхождения или -1, если ключа в массиве нет.

```
int seek_linear(key x, key a[], int N)
{  int i=0;
   while ((i<N) && (a[i]!=x))
      i++;
   if (i<N)
      return i;       /* элемент найден */
   else
      return -1;      /* элемент не найден */
}
```

Обратим внимание, что при нарушении первого условия невыхода индекса за границы массива второе условие может стать некорректным (обращение к $a[i]$ при $i = N$ может вызвать ошибку доступа к памяти). К счастью, семантика операции && в Си такова, что при ложности первого аргумента второй просто не вычисляется. В других языках (например, в Паскале) это может быть не так, поэтому данной схемой следует пользоваться с осторожностью.

Время работы алгоритма зависит от количества сравнений ключей: в худшем случае алгоритм последовательного поиска делает полный проход по всем элементам массива, поэтому время работы этого алгоритма линейно по N — количеству элементов в массиве.

Данный метод, очевидно, применим и к структурам с последовательным доступом.

2.3. Бинарный поиск

Бинарный поиск называют также методом *дихотомии* или *деления отрезка пополам*. Его применимость ограничена двумя достаточно сильными условиями:

– структурами с прямым доступом к элементам;

– необходимой упорядоченностью элементов структуры.

Идея бинарного поиска состоит в том, что на каждом шаге искомый ключ сравнивается со средним элементом рабочего отрезка исходной последовательности. При этом, в силу ее упорядоченности, по результату

сравнения можно судить, в какой половине — левой или правой – надо продолжать поиск, исключив всю другую половину из рассмотрения. Таким образом, при каждом сравнении исключается не один элемент, как при линейном поиске, а сразу половина непросмотренных элементов.

Введем две переменные $left$ и $right$, которые будут указывать границы (диапазон индексов) подмассива для поиска элемента x. Начально $left = 1$, $right = N$, где N — количество элементов в массиве. Выберем из данного диапазона элемент, находящийся в позиции $[\frac{left+right}{2}]$, где $[\,]$ — целая часть, и сравним его с x. При совпадении поиск заканчивается. Если средний элемент меньше x, то переменной $left$ (левой границе) присвоим значение $[\frac{left+right}{2}] + 1$, так как все элементы с меньшими номерами меньше x и, следовательно, их можно исключить из дальнейшего поиска. Если выбранный элемент больше x, переменной $right$ присвоим значение $[\frac{left+right}{2}] - 1$. Исключив из дальнейшего поиска все элементы с большими номерами, далее повторим те же действия с элементами массива в новом диапазоне $left...right$. Процесс продолжается пока либо не найдется элемент x, либо не выполнится условие $right = left$.

```
int seek_binary(key x, key a[], int N)
{  int left=0;
   int right=N-1;
   int middle;
   do {
      middle=(left+right)/2;
      if (x == a[middle])
         return middle;
      else if(a[middle]<x)
              left=middle+1;
           else right=middle-1;
   } while (left<=right);
   return -1;
}
```

Заметим, что средний элемент для сравнения можно выбирать произвольным образом, например деля отрезок в пропорции $1 : 2$, но математически доказывается, что наилучшая в среднем эффективность метода достигается при делении отрезка на одинаковые части. В этом случае максимальное число сравнений равно $\lceil log_2 N \rceil$.

2.4. Прямой поиск подстроки

Еще одна распространенная задача поиска — поиск заданной подпоследовательности (или *подстроки*, поскольку чаще всего эта задача возникает при обработке символьных последовательностей, или *строк*).

Пусть заданы строка s из N элементов и строка q из M элементов, где $0 < M \leq N$. Требуется определить, содержит ли строка s подстроку q, и в случае положительного результата выдать позицию k, с которой начинается вхождение q в s, т. е.:

$$q[1] = s[k],\ q[2] = s[k+1], ..., q[M] = s[k+M-1].$$

Будем называть строку q *шаблоном* поиска. Задача прямого поиска заключается в поиске индекса k, указывающего на первое с начала строки s совпадение с шаблоном q.

Простейший алгоритм поиска сводится к повторяющимся сравнениям отдельных символов из строки и шаблона. Сначала шаблон q «накладывается» на строку s начиная с первого символа строки и выполняется последовательное сравнение соответствующих символов. Если до i-й позиции символы строки и шаблона совпали, а $q[i] \neq s[i]$, то надо «сдвинуть» шаблон, т. е. «наложить» его на строку, начиная со второго символа строки, повторить сравнения и т. д.

Процесс повторяется до тех пор, пока смещение q по s не достигнет $N-M+1$ (очевидно, что при больших смещениях хвост шаблона выходит за хвост строки) или пока не будет найден индекс k такой, что $q[1..M] = s[k..k+M-1]$. Очевидно, что данный алгоритм реализуется с помощью двух вложенных циклов и в худшем случае требуется произвести $(N-M) \cdot M$ сравнений.

Ниже приведена функция, реализующая алгоритм прямого поиска шаблона q в строке s и выдающая результатом индекс первого вхождения или -1:

```
int seek_substring_A (char s[], char q[])
{  int i,j,k,N,M;
   N = strlen(s);
   M = strlen(q);
   k = -1;
   do {
       k++;    /* k - смещение шаблона по строке */
       j = 0;  /* j - смещение по шаблону; */
       while ((j<M) && (s[k+j]==q[j]))
           j++;
```

```
      if (j==M)
          return k; /* шаблон найден */
  } while (k<N-M);
  return -1;          /* шаблон не найден */
}
```

Другой вариант этой же функции не требует изначального знания длин строк M и N, а предполагает наличие в конце строки выделенного символа-терминатора (в Си это '\0'). Этот вариант применим в том случае, если строка s представима файлом, для определения длины которого потребовался бы отдельный просмотр. В этом случае внешний цикл заканчивается не при $k = N - M$, а немного позже.

```
int seek_substring_B(char s[], char q[])
{  int i,j,k;
   k = -1;
   do {
       k++;         /* k - смещение шаблона по строке */
       j= -1;       /* j - смещение по шаблону */
       do {
          j++;
          if (q[j]==0)
             return k;
       } while (s[k+j]==q[j]);
  } while (s[k+j]!=0); /* здесь j меньше длины q */
  return -1;
}
```

2.5. Алгоритм Кнута—Морриса—Пратта

У метода прямого поиска подстроки есть очевидный недостаток. Каждый раз при неудачном сравнении шаблона с подстрокой оба индекса-указателя возвращаются к некоторым уже просмотренным позициям, и сравнение начинается, по сути, заново. Иначе говоря, уже полученная информация о том, что некоторый префикс (начало) шаблона q совпал с суффиксом (концом) просмотренной подстроки s, теряется. Из-за возвратов общее время поиска становится нелинейным: $\sim N \cdot M$.

Какую пользу можно извлечь из этой информации? Рассмотрим следующий пример. Пусть q наложен на s начиная с позиции k, до позиций i и j они совпадают, а в i и j расходятся:

$$
\begin{array}{lccccccccccccc}
 & 1 & & k & & & & i & & & & & & N \\
s: & b & a & a & b & a & b & a & b & a & c & a & b & a & b \\
q: & & & \underline{a} & \underline{b} & \underline{a} & \underline{b} & \underline{a} & c & a & & & & & \\
 & & & 1 & & & & & jM & & & & & &
\end{array}
$$

Здесь $j = 6$ символов строки, следующих за позицией k, уже известны, поэтому можно, не выполняя сравнений, установить, что некоторые последующие сдвиги шаблона заведомо бесперспективны. Например, сдвиг на 1 позицию бесперспективен, так как при этом $q[1] =' a'$ сравнится с уже известным $s[k+1] =' b'$ и совпадения не будет. А вот сдвиг на 2 позиции сразу отвергнуть нельзя: $q[1...4]$ совпадает с уже известной подстрокой $s[k+2...k+5]$. Совпадут ли остальные $M-4$ символа, станет известно только при рассмотрении последующих символов s, причем сравнение можно начинать сразу с 5-й позиции шаблона. Таким образом, при неудаче очередного сравнения надо сдвинуть шаблон вперед так, чтобы его начало совпало с уже прочитанными символами строки. Если таких сдвигов можно указать несколько, следует выбрать кратчайший из них.

Эту идею реализует КМП-алгоритм (Кнут, Моррис, Пратт) [4].

Алгоритм А4:

- вход: q – шаблон, s – строка, M – длина шаблона, N – длина строки, $M < N$.

  $i := 1; j := 1;$

  пока $i \le N$ и $j \le M$ цикл

  пока $j > 0$ и $s[i] \ne q[j]$ цикл

  $j := d_j;$ /* $0 \le d_j < j$ */

  конец цикла;

  $i := i + 1;\ j := j + 1;$

  конец цикла;
- выход: если $j > M$ то шаблон q найден в позиции $i - M$;

  иначе /* $i > N$ */ шаблон q не найден.

Индекс-указатель i пробегает строку s без возвратов (что обеспечивает линейность времени работы алгоритма). Индекс j синхронно пробегает шаблон q, однако может возвращаться к некоторым предыдущим позициям d_j, которые будут выбираться так, чтобы обеспечить на всем протяжении алгоритма инвариантность следующего условия $Eq(i, j)$: «все символы шаблона, предшествующие позиции j, совпадают с таким же числом символов строки, предшествующих позиции i»:

$Eq(i,j)$:
$1 \le i \le N{+}1$ и $1 \le j \le M{+}1$ и $pref(q, j{-}1) = suff(pref(s, i{-}1), j{-}1)$,
где $pref(str, k) = str[1..k]$ — k-префикс str (6)
$suff(str, k) = str[l{-}k{+}1..l]$ — k-суффикс str[] (l — длина str)
(примем $str[i..i{-}1] =$ “ ”)

Истинность условия $Eq(i, M{+}1)$ означает, что шаблон q входит в s начиная с позиции $i{-}M$. Выполнение условия $Eq(N{+}1, j)$ при $j \le M$ означает, что шаблона в строке нет.

Займемся обеспечением свойства Eq. При первом входе в цикл индексы указывают на начала строк и $Eq(i,j) = Eq(1,1)$, очевидно, истинно. На каждом проходе цикла указатель i сдвигается на одну позицию строки вперед без возвратов. Пока очередные символы совпадают, внутренний цикл не выполняется и j просто увеличивается синхронно с i; это обеспечивает сохранение условия $Eq(i_{\text{нов}}, j_{\text{нов}}) = Eq(i{+}1, j{+}1)$ без сдвига шаблона относительно строки.

При несовпадении очередных символов надо сдвинуть шаблон так, чтобы некоторый d_j-префикс q продолжал совпадать с d_j-суффиксом просмотренной строки $s[1..i]$, тем самым сохраняя инвариант $Eq(i_{\text{нов}}, j_{\text{нов}}) = Eq(i+1, d_j+1)$ для следующей итерации цикла. Изменение соответствия позиций с (i,j) на $(i{+}1, d_j{+}1)$ означает сдвиг q относительно s на $D = j - d_j > 0$ позиций вперед. Отсюда $d_j < j$. Если таких d_j-префиксов можно указать несколько, надо выбрать из них наибольший по длине, чтобы сдвиг D был кратчайшим. Если таких префиксов нет, возьмем $d_j = 0$, так как $Eq(i{+}1, 1)$ всегда истинно. Это соответствует сдвигу шаблона на $D{=}j$, к позиции $s[i{+}1]$; т. е. следующее сравнение начнется со следующей непрочитанной позиции строки, имея «нулевую историю» совпадений.

Теперь займемся определением d_j. До сдвига $pref(q, j{-}1)$ совпадает с $suff(pref(s, i{-}1), j{-}1)$. Чтобы сдвиг шаблона на $D{=}j{-}d_j$ был перспективен, префикс $pref(q, j{-}D{-}1){=}pref(q, d_j{-}1)$ должен совпадать с суффиксом $suff(pref(s, i{-}1), d_j{-}1)$, с которым до сдвига совпадал $suff(pref(q, j{-}1), d_j{-}1)$. Отсюда $pref(q, d_j{-}1){=}suff(pref(q, j{-}1), d_j{-}1)$, т. е.

$$q[1...d_j{-}1]{=}q[j{-}d_j{+}1...j{-}1]. \tag{7}$$

Это условие необходимо для перспективности сдвига на $D = j - d_j$, но еще не достаточно; из сравнения нам еще известно, что $s[i]$ не совпадает с $q[j]$. Поэтому если $q[d_j]{=}q[j]$, то сдвиг бесперспективен. Сделаем

соответствующее уточнение в формуле (7):

$$q[1...d_j-1] = q[j-d_j+1...j-1] \text{ и } q[d_j] \neq q[j] \qquad (8)$$

Добавив теперь условие максимальности длины префикса d_j, выразим зависимость d_j от j следующей *префикс-функцией*:

$$d[j] = max\{d \mid d < j \text{ и } q[1...d-1] = q[j-d+1...j-1] \text{ и } q[d] \neq q[j]\}. \qquad (9)$$

Как можно видеть, префикс-функция зависит только от шаблона q, но не от строки s, поэтому она может быть вычислена ещё до начала поиска и задана в алгоритме таблицей значений.

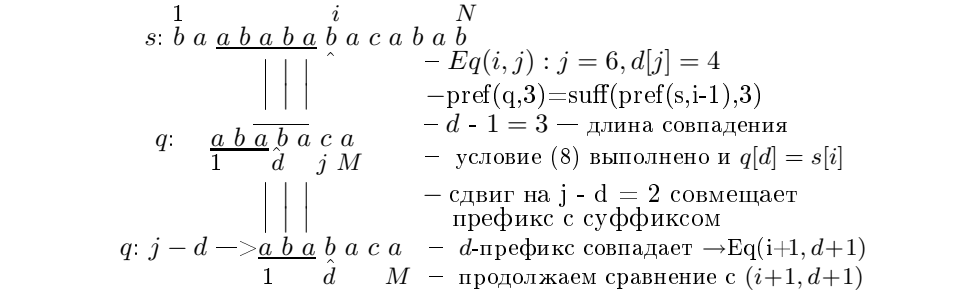
Однако зависимость d_j от строки всё же имеется: если $q[d_j] \neq s[i]$, то сдвиг тоже заведомо бесперспективен. В этом случае вычисленное $d[j]$ следует отвергнуть и выбрать предыдущее по величине, которое, как нетрудно увидеть, равно $d[d[j]]$. Так как $j > d_j$, все длины перспективных префиксов q образуют последовательность, убывающую до нуля: $d[j] > d[d[j]] > d[d[d[j]]] > ... > d[...d[j]...] = 0$.

Выбором подходящего d_j, с учетом всего сказанного, занимается внутренний цикл КМП-алгоритма.

Ниже приведены значения префикс-функции и величины сдвига для шаблона *ababaca* из примера выше:

j:	1	2	3	4	5	6	7	
$q[j]$:	a	b	a	b	a	c	a	
$d[j]$:	0	1	0	1	0	4	0	– по формуле (9)
D=$j-d[j]$:	1	1	3	3	5	2	7	

Проиллюстрируем сказанное на примере:



Как видим, префикс-функция устроена достаточно хорошо: она выдает много нулей, т. е. почти для всех j при несовпадении происходит сдвиг шаблона на всю длину уже проверенной части.

Последний штрих к КМП-алгоритму: для построения таблицы значений префикс-функции можно воспользоваться самим же КМП-алгоритмом!

Допустим, что для всех позиций k шаблона, предшествующих и включая i, $d[k]$ уже вычислены и $d[i] = j+1$. Это означает, что $pref(q, j) = suff(pref(q, i), j)$. Сравним $q[i+1]$ и $q[j+1]$: если они равны, то $pref(q, j+1) = suff(pref(q, i+1), j+1)$, т. е. $d[i+1] = j+2$; если они не равны, то выберем для испытаний следующий по длине префикс q, являющийся суффиксом $pref(q, i)$, т. е. $d[j]$.

Окончательно приходим к следующей программе (замечание: поскольку в Си массивы индексуются с нуля, все рассмотренные выше индексы и значения d уменьшены на 1):

```
int seek_substring_KMP (char s[], char q[]) {
   int i, j, N, M;
   N = strlen(s);
   M = strlen(q);
   int *d = (int *)malloc(M*sizeof(int)); /* динамический
                                             массив длины M+1 */
   /* Вычисление префикс-функции */
   i=0; j=-1; d[0]=-1;
   while (i < M-1) {
      while ((j>=0) && (q[j] != q[i]))
         j = d[j];
      i++; j++;
      if (q[i]==q[j])
        d[i] = d[j];
      else
        d[i] = j;
   }
   /* поиск */
   for (i=0,j=0; (i<=N-1) && (j<=M-1); i++,j++)
      while ((j>=0) && (q[j]!=s[i]))
         j = d[j];
   free (d);        /* освобождение памяти массива d */
   if (j==M)
      return i-j;
```

```
   else /* i==N */
      return -1;
}
```

2.6. Алгоритм Рабина—Карпа поиска подстроки

Другой вариант поиска с линейной сложностью основан на «арифметическом» принципе. Пусть строка и шаблон состоят из символов алфавита A. Каждый символ этого алфавита будем считать d-ичной цифрой, где $d = |A|$. Строку из k символов можно рассматривать как запись d-ичного k-значного числа. Тогда поиск шаблона в строке сводится к серии из $N - M$ сравнений числа, представляющего шаблон, с числами, представляющими подстроки s длины M. Мы расчитываем, что сравнение чисел может быть выполнено за время, пропорциональное M, и тогда эффективность поиска будет $O(N + M)$.

Для начала предположим, что $A = \{0, 1, ..., 9\}$. Число, десятичной записью которого является шаблон q, обозначим через t_q. Аналогично, обозначим через t_k число, десятичной записью которого является подстрока $s[k \ldots k + M - 1]$. Очевидно, что подстрока $s[k \ldots k + M - 1]$ совпадает с шаблоном q тогда и только тогда, когда $t_q = t_k$.

По схеме Горнера, описанной в первой части этого учебного пособия, значения t_q и t_1 можно вычислить за время, пропорциональное M. Временно забудем о том, что вычисления могут привести к очень большим числам. Из t_k $(1 < k \leq N - M)$ за константное время можно вычислить t_{k+1}. В самом деле, по схеме Горнера:

$$t_k = s[k+M-1]+10\cdot(s[k+M-2]+10\cdot(s[k+M-3]+\ldots+10\cdot(s[k])\ldots). \quad (10)$$

Чтобы получить t_{k+1} из t_k, надо удалить последнее слагаемое из формулы (10) (т. е. вычесть $10^{M-1} \cdot (s[k])$), результат умножить на 10 и добавить к нему $s[k+M]$. В результате получим следующее рекуррентное соотношение:

$$t_{k+1} = 10 \cdot (t_k - 10^{M-1} \cdot s[k]) + s[k + M]. \quad (11)$$

Таким образом, вычисление t_{k+1} через t_k выполняется за константное время.

Вычислив все t_k, мы можем по очереди сравнить их с t_q, определив тем самым совпадение или несовпадение шаблона q с подстроками $s[k \ldots k+M-1]$. Время работы этого алгоритма пропорционально $N+M$.

До сих пор мы не учитывали того, что числа могут быть слишком велики. С этой трудностью можно справиться следующим образом. Надо проводить вычисления чисел t_q и t_k и вычисления по формуле (11) по модулю фиксированного числа p. Тогда все числа не превосходят p и действительно могут быть вычислены за время порядка M. Обычно в качестве p выбирают простое число, для которого $d \cdot p$ помещается в машинное слово, все вычисления в этом случае упрощаются. Рекуррентная формула (11) приобретает вид

$$t_{k+1} = (d \cdot (t_k - h \cdot s[k]) + s[k+M]) mod\ p, \quad \text{где } h \equiv (d^{M-1})\ mod\ p \tag{12}$$

Из равенства $t_q \equiv t_k (mod\ p)$ еще не следует, что $t_q = t_k$ и, стало быть, что $q = s[k\ ...\ k+M-1]$. В этом случае надо для надежности проверить совпадение шаблона и подстроки.

Алгоритм и функция, описанные ниже, определяют первое вхождение шаблона q в строку s. Считаем, что код любого символа не превышает значения d.

Алгоритм А5:

- вход: q – шаблон, s – строка, M – длина шаблона, N – длина строки, $M < N$, d – число символов в алфавите.

  По схеме Горнера вычислить числа t_q и t_1 по модулю p;

  цикл по k от 1 до $N - M + 1$

  если $t_q = t_k$ то

  сравнить шаблон q с подстрокой $s[k\ ...\ k+M-1]$;

  если они совпадают, то

  выдать k – результат сравнения;

  выход

  по формуле (12) вычислить t_{k+1};

  конец цикла

- выход: k – позиция начала вхождения шаблона в строку.

```
/* d - число символов в алфавите */
/* p - число, по модулю которого производятся вычисления */
/* возвращает смещение вхождения q в s относительно начала */
                                                    /* строки */
int Robin_Carp_Matcher(char s[], char q[], int d, int p)
{  int i,h,k,M,N,t_q, t_k;
   N = strlen(s);
```

```
   M = strlen(q);
   /* вычисление h=(d^(M-1)) mod p */
   h=1;
   for (i=1; i<M; i++)
        h=(h*d) % p;
   /* вычисление t_1 и t_q по схеме Горнера  */
   t_q = 0;
   t_k = 0;
   for (i=0; i<M; i++) {
        t_q = (d*t_q + q[i]) % p;
        t_k = (d*t_k + s[i]) % p;
   }
   /* сравнение шаблона  с */
   /* подстроками и вычисления по формуле (12) */
   for (k=0; k<=N-M; k++) {
        if (t_q==t_k) {
            for (i=0; (i<M) && (q[i]==s[k+i]); i++);
            if (i==M) return k;
        }
        if (k<N-M) {   /* вычисления по формуле (8) */
            t_k = (d * (t_k - s[k] * h) + s[k+M]) % p;
            if (t_k < 0)
                t_k + = p;
   }    }
   return -1;
}
```

2.7. Алгоритм Бойера—Мура поиска подстроки

Данный алгоритм ведет сравнение символов из строки и шаблона начиная с $q[M]$ и с $s[i+M-1]$ элементов в обратном порядке. В нем используется дополнительная таблица d. Для каждого символа x из алфавита (кроме последнего в шаблоне) $d[x]$ есть расстояние от самого правого вхождения x в шаблоне до последнего символа шаблона. Для последнего символа в шаблоне $d[x]$ равно расстоянию от предпоследнего вхождения x до последнего или M, если предпоследнего вхождения нет. Например, для шаблона *abcabeabce* ($M = 10$) $d['a'] = 3$, $d['b'] = 2$; $d['c'] = 1$, $d['e'] = 4$ и для всех символов x алфавита, не входящих в шаблон, $d[x] = 10$.

Будем последовательно сравнивать шаблон q с подстроками $s[i-M+1..i]$ (в начале $i = M$). Введем два рабочих индекса: $j = M, M-1, ..., 1$ — пробегающий символы шаблона, и $k = i, ..., i-M+1$ — пробегающий подстроку. Оба индекса синхронно уменьшаются на каждом шаге. Если все символы q совпадают с подстрокой (т. е. j доходит до 0), то шаблон q считается найденным в s с позиции k ($k = i-M+1$). Если $q[j] \neq s[k]$ и $k = i$, т. е. расхождение случилось сразу же, в последних позициях, то q можно сдвинуть вправо так, чтобы предпоследнее вхождение символа $s[i]$ в q совместилось с $s[i]$. Если $q[j] \neq s[k]$ и $k < i$, т. е. последние символы совпали, то q сдвинется так, чтобы последнее вхождение $s[i]$ в q совместилось с $s[i]$ (рис. 1). В обоих случаях величина сдвига равна $d[s[i]]$, по построению. В частности, если $s[i]$ вообще не встречается в q, то смещение происходит сразу на полную длину шаблона M.

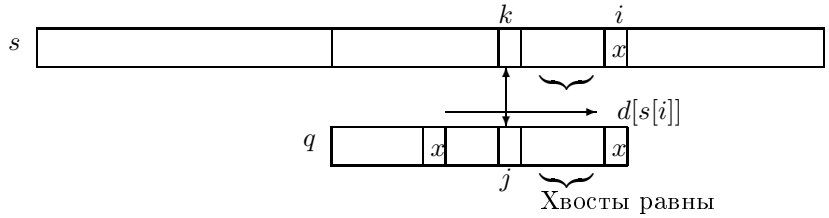


Рис. 1: Сравнение шаблона q с подстрокой s

Приведенная ниже программа реализует алгоритм Бойера–Мура [5].

```
int seek_substring_BM(unsigned char s[], unsigned char q[])
{  int d[256];
   int i, j, k, N, M;
   N = strlen(s);
   M = strlen(q);
   /* построение d */
   for (i=0; i<256; i++)
        d[i]=M;          /* изначально M во всех позициях */
   for (i=0; i<M-1; i++)/* M-i-1 -- расстояние до конца d */
       d[(unsigned char)q[i]]=M-i-1;  /* кроме последнего */
                                                 /* символа */
   /* поиск */
   i=M-1;
```

```
   do {
       j=M-1;
       k=i;            /* сравнение строки и шаблона */
       while ((j >= 0) && (q[j] == s[k])) {
          /* j - по шаблону, k - по строке */
           k--;
           j--;
       }
       if (j < 0)      /* шаблон просмотрен полностью */
          return k+1;
       i+=d[(unsigned)s[i]];  /* сдвиг на */
                        /* расстояние d[s[i]] вправо */
   } while (i < N);
   return -1;
}
```

Определение длин исходных строк выполняется в Си поиском заключительного нулевого символа и требует, таким образом, времени $N + M$. Для построения таблицы d необходимо занести значение M во все позиции таблицы и выполнить один проход по всем элементам шаблона q, т. е. таблица строится за время $(256 + M)$. Считаем, что M намного меньше N. Как правило, данный алгоритм требует значительно меньше N сравнений. В благоприятных обстоятельствах, а именно если последний символ шаблона всегда попадает на несовпадающий символ текста, максимальное число сравнений символов есть $\frac{N}{M}$.

3. Сортировка

3.1. Задача сортировки

Под сортировкой понимают процесс перестановки объектов заданной совокупности в определенном порядке (возрастающем или убывающем). Целью сортировки часто является облегчение последующего поиска элементов в отсортированном множестве (например, возможность применения быстрого бинарного поиска). Трудно представить себе телефонную книгу, словарь, тематический каталог или любой другой обширный информационный справочник, составленные без какого-либо упорядочения, вразброс. Эффективность методов сортировки и поиска важна при обработке больших объемов данных, зачастую именно она определяет эффективность и работоспособность всей системы. С сортировкой связаны многие фундаментальные приемы построения алгоритмов. Сортировка сама является идеальным примером для демонстрации разнообразия алгоритмов, решающих одну и ту же задачу, когда каждый алгоритм имеет определенные преимущества и недостатки, которые нужно оценивать с точки зрения конкретной ситуации.

На примере сортировки мы увидим, как влияет выбор структур данных на выбор метода их обработки. Зависимость метода от структуры данных настолько сильна для задачи сортировки, что методы сортировки обычно разделяют на две категории: *внутреннюю* сортировку массивов и *внешнюю* — сортировку файлов.

Итак, задача сортировки состоит в том, чтобы упорядочить N объектов $a_1, ..., a_N$, т. е. переставить их в такой последовательности $a_{p_1}, ..., a_{p_N}$, чтобы их ключи расположились в неубывающем порядке $k_{p_1} \leq k_{p_2} \leq ... \leq k_{p_N}$. (Как и прежде, мы будем отождествлять сортируемые записи a_i с их ключами k_i.)

Сортировка называется *устойчивой*, если она удовлетворяет дополнительному условию, согласно которому записи с одинаковыми ключами остаются в прежнем порядке, т. е. если $k_{p_i} = k_{p_j}$ и $i < j$, то $p_i < p_j$. При устойчивой сортировке относительный порядок элементов с одинаковыми ключами не меняется.

Достаточно хороший общий алгоритм затрачивает на сортировку N записей время $\sim N \cdot log_2 N$, при этом требуется около $log_2 N$ проходов по данным. Подробные обсуждения алгоритмов сортировки можно найти в классическом исследовании Д. Кнута [2].

3.2. Сортировка массивов

Напомним, что массивы являются структурами прямого доступа, т. е. для них возможно непосредственное обращение к произвольному элементу. Это выделяет методы сортировки массивов в отдельную группу, отличную от сортировки файлов и списков.

Нас будут интересовать методы с экономным использованием памяти, где не требуются вспомогательные массивы для промежуточной пересылки элементов, а переупорядочение элементов сортируемого массива тем или иным образом происходит внутри него самого. (Поэтому эту группу методов сортировки называют *внутренней сортировкой*.)

Мерой эффективности будет служить количество необходимых сравнений ключей C и количество необходимых пересылок элементов M. C и M будут определяться некоторыми функциями, зависящими от числа сортируемых элементов N.

Методы сортировки можно разбить на несколько основных классов в зависимости от лежащего в их основе приема сортировки:

— включения;
— выбор;
— обмен;
— подсчет;
— разделение;
— слияние.

3.3. Сортировка простыми включениями

Разделим условно все элементы массива на две последовательности: *входную*, еще неупорядоченную последовательность $a_i, ..., a_N$ и *готовую* последовательность $a_1, a_2, ..., a_{i-1}$, элементы которой уже отсортированы. В алгоритмах, основанных на методе включения, на каждом i-м шаге i-й элемент входной последовательности вставляется (включается) в подходящее место готовой последовательности.

Сортировка простыми включениями наиболее очевидна. Пусть $2 \le i \le N$, $a_1, ..., a_{i-1}$ уже отсортированы, т. е. $a_1 \le a_2 \le ... \le a_{i-1}$. Будем сравнивать по очереди a_i с $a_{i-1}, a_{i-2}, ...$ до тех пор, пока не обнаружим, что элемент a_i следует вставить между a_j и a_{j+1} $(0 \le j \le i-1)$ элементами. После этого подвинем записи $a_{j+1}, ..., a_{i-1}$ на одно место вправо и переместим запись a_i в позицию $j+1$.

Процесс сортировки включениями покажем на примере последовательности, состоящей из восьми ключей:

$$40 \mid 51 \;\; 8 \;\; 38 \;\; 90 \;\; 14 \;\; 2 \;\; 63. \tag{13}$$

Готовая последовательность в ней отделена от входной чертой «|». Изначально готовая последовательность состоит из одного первого элемента 40. На шаге $i = 2$ второй элемент 51 будет сравнен с 40 и займет второе место в готовой последовательности:

$$40 \;\; 51 \mid 8 \;\; 38 \;\; 90 \;\; 14 \;\; 2 \;\; 63.$$

На шаге $i = 3$ третий элемент 8 будет поочередно сравнен с 51 и 40 и продвинется на первое место в готовой последовательности:

$$8 \;\; 40 \;\; 51 \mid 38 \;\; 90 \;\; 14 \;\; 2 \;\; 63.$$

Продолжая этот процесс, получаем

```
i = 4:          8  38  40  51 | 90   14   2    63
i = 5:          8  38  40  51   90 | 14   2    63
i = 6:          8  14  38  40   51   90 | 2    63
i = 7:          2   8  14  38   40   51   90 | 63
i = 8:          2   8  14  38   40   51   63   90 |
```

где i — номер шага.

В программах, реализующих алгоритмы внутренней сортировки, последовательность элементов, как и прежде, будем представлять массивом ключей.

```
void sort_by_insertion(key a[], int N)
{  key x;
   int i,j;
   for (i=1; i < N; i++) {
        x = a[i];
        for (j=i-1; (j>=0)&&(x < a[j]); j--)
             a[j+1] = a[j];
        a[j+1] = x;
   }
}
```

Процесс поиска с одновременным сдвигом реализуется внутренним индексом по j. Он заканчивается, когда либо найден элемент $a[j]$, меньший, чем ключ x, либо когда достигнут левый конец готовой последовательности (начало массива).

Анализ. На i-м шаге максимально возможное число сравнений C_i во внутреннем цикле равно $i-1$; если предположить, что все перестановки N ключей равновероятны, число сравнений в среднем равно $\frac{i}{2}$. Для M_i, количества пересылок на i-м шаге, максимальное $\mathrm{M}_i = C_i + 2$. Всего шагов $N-1$. Следовательно, количество сравнений и пересылок в худшем и лучшем случаях:

$C_{max} = 1 + 2 + 3 + ... + N - 1 = \frac{N \cdot (N-1)}{2}$,
$C_{min} = N - 1$,
$M_{min} = 2 \cdot (N-1)$,
$M_{max} = \frac{N \cdot (N-1)}{2} + 2 \cdot (N-1) \approx \frac{N \cdot (N+3)}{2}$.

Наилучшие оценки реализуются в случае, когда элементы массива уже упорядочены, наихудшие — если элементы расположены в обратном порядке. Данный алгоритм описывает устойчивую сортировку, так как он оставляет неизменным порядок элементов с одинаковыми ключами.

3.4. Сортировка бинарными включениями

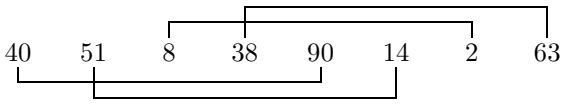
Алгоритм сортировки простыми включениями вставляет i-й элемент в готовую последовательность, которая уже отсортирована. Следовательно, для нахождения места для i-го элемента можно использовать описанный ранее метод бинарного поиска элемента в отсортированном массиве (см. п. 2.3.), в котором на i-м шаге выполняется $\sim log_2 i$ сравнений. Поэтому всего будет произведено $\sim N \cdot log_2 N$ сравнений. Но количество пересылок в этом методе не изменится.

В программе сортировки методом простых включений внутренний цикл поиска с одновременным сдвигом следует разделить: отдельно бинарным поиском быстро находится позиция вставки, затем все элементы готовой последовательности, находящиеся правее этой позиции, сдвигаются вправо. Детали реализации оставляем читателю в качестве упражнения.

3.5. Сортировка включениями с убывающим шагом. Метод Шелла

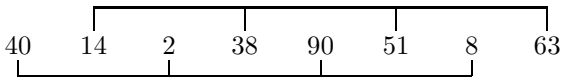
Одновременно несколькими исследователями: Шеллом, Хоаром, Флойдом – было замечено, что для алгоритмов сортировки, перемещающих в последовательности запись вправо или влево только на одну позицию, среднее время работы будет в лучшем случае пропорционально N^2. Поэтому метод, существенно превосходящий по скорости простые вставки, должен использовать некоторый механизм, с помощью которого записи могли бы перемещаться «большими скачками, а не короткими шажками». Д. Шелл предложил в 1959 г. метод, названный сортировкой *с убывающим шагом*.

Продемонстрируем его на последовательности (13). На первом проходе выделим в отдельные подпоследовательности элементы, отстоящие друг от друга на четыре позиции:



Полученные 4 последовательности отсортируем на месте независимо друг от друга методом простых включений. Этот процесс называется 4-*сортировкой*.

В результате 4-сортировки получим последовательность



На следующем шаге элементы, отстоящие друг от друга на две позиции, объединяются в подпоследовательности, которые сортируются методом простых включений. Этот процесс называется 2-*сортировкой*. Две новые подпоследовательности таковы: $40, 2, 90, 8$ и $14, 38, 51, 63$. После 2-сортировки получим последовательность:

2 14 8 38 40 51 90 63

Наконец, на третьем шаге все элементы сортируются методом простых включений. К последнему шагу элементы довольно хорошо упорядочены, поэтому требуется мало перемещений. Данный процесс называется 1-сортировкой.

В сортировке методом Шелла можно использовать любую убывающую последовательность шагов $h_t, h_{t-1}, ..., h_1$, в которой последний шаг

$h_1 = 1$. Каждая h-сортировка программируется как сортировка простыми включениями.

Анализ метода Шелла. Чтобы выбрать некоторую хорошую последовательность шагов сортировки, нужно проанализировать время работы как функцию от этих шагов. Это приводит к очень красивым, но еще не до конца решенным математическим задачам: никому до сих пор не удалось найти наилучшую возможную последовательность шагов $h_t, h_{t-1}, ..., h_1$ для больших N. Тем не менее известно много довольно интересных фактов. Кнут указывает, что разумным выбором может быть следующая последовательность приращений: $..., 121, 40, 13, 4, 1$, где $h_{k+1} = 3 \cdot h_k + 1, h_1 = 1$. Он рекомендует также последовательность шагов $..., 31, 15, 7, 3, 1$, где $h_{k+1} = 2 \cdot h_k + 1, h_1 = 1$. Выявлен примечательный факт, что элементы последовательностей приращений не должны быть кратны друг другу. Это позволяет на каждом проходе сортировки перемешивать цепочки, которые ранее никак не взаимодействовали. Желательно, чтобы взаимодействие между разными цепочками происходило как можно чаще.

Утверждение. *Если k-отсортированная последовательность i-сортируется ($k > i$), то она остается k-отсортированной.*

Из этого утверждения следует, что с каждым следующим шагом сортировки с убывающим приращением количество отсортированных элементов в последовательности возрастает.

Оценка эффективности этого метода зависит от последовательности h_t. Ниже приведены примеры из работы [2], показывающие оценку для разных последовательностей h_t, при $100 \leq N \leq 60000$. Для последовательности шагов $2^k + 1, ..., 9, 5, 3, 1$ количество пересылок пропорционально $N^{1.27}$, для последовательности $2^k - 1, ..., 15, 7, 3, 1$ — $N^{1.26}$, а для последовательности $(3^k - 1)/2, ..., 40, 13, 4, 1$ — $N^{1.25}$.

3.6. Сортировка простым выбором

Методы сортировки посредством выбора основаны на идее многократного выбора. На i-м шаге выбирается наименьший элемент из входной последовательности $a_i, ..., a_N$ и меняется местами с a_i-м. Таким образом, после шага i на первом месте во входной последовательности будет находиться наименьший элемент. Затем этот элемент перемещается из входной в готовую последовательность. Процесс выбора наименьшего элемента из входной последовательности повторяется до тех пор, пока в ней останется только один элемент.

Проиллюстрируем этот метод на той же последовательности (13):

$$| \; 40 \;\; 51 \;\; 8 \;\; 38 \;\; 90 \;\; 14 \;\; \underline{2} \;\; 63.$$

Готовая последовательность по-прежнему отделяется от входной чертой «|». На первом шаге находим наименьший элемент 2, обмениваем его с первым элементом 40 и перемещаем в готовую последовательность:

$$2 \; | \;\; 51 \;\; \underline{8} \;\; 38 \;\; 90 \;\; 14 \;\; 40 \;\; 63.$$

На втором шаге минимальный элемент из оставшихся, 8, обменивается с 51 и отделяется:

$$2 \;\; 8 \; | \;\; 51 \;\; 38 \;\; 90 \;\; \underline{14} \;\; 40 \;\; 63.$$

Продолжая этот процесс, получаем следующие промежуточные последовательности, в которых текущий минимальный элемент подчеркнут:

$$\begin{array}{cccccccc}
2 & 8 & 14 & |\underline{38} & 90 & 51 & 40 & 63 \\
2 & 8 & 14 & 38 & |90 & 51 & \underline{40} & 63 \\
2 & 8 & 14 & 38 & 40 & |\underline{51} & 90 & 63 \\
2 & 8 & 14 & 38 & 40 & 51 & |\underline{63} & 90 \\
2 & 8 & 14 & 38 & 40 & 51 & 63 & |\underline{90}
\end{array}$$

Данный метод в некотором смысле противоположен сортировке простыми включениями, при которой на каждом шаге рассматривается только один очередной элемент входной последовательности, и для этого элемента подыскивается место в готовой последовательности. При сортировке простым выбором, наоборот, рассматриваются все элементы входной последовательности и для фиксированного места из нее выбирается наименьший элемент. При этом не возникает необходимости “сдвига” участка массива, поскольку выбранный элемент вставляется всегда в конец готовой последовательности. Вытесняемый же элемент достаточно переставить на освободившееся место в неотсортированной входной части.

Ниже приведен алгоритм сортировки простым выбором:

```
void sort_by_selection(key a[], int N)
{  int i,j,k;
   key x;
   for (i=0; i<N-1; i++) {
        k=i;
        for (j=i+1; j<N; j++)
```

```
            if (a[k]>a[j])   /* k хранит индекс текущего */
                k=j;         /* минимального элемента */
        if (i!=k) {
           x=a[i]; a[i]=a[k]; a[k]=x;
        }
   }
}
```

Анализ. Ясно, что число C_i сравнений на i-м шаге не зависит от начального порядка элементов. На первом шаге первый элемент сравнивается с остальными $N-1$ элементами, на втором шаге число сравнений будет — $N-2$ и т. д. Поэтому число сравнений есть $C = (N-1)+(N-2)+ \ldots + 1 = \frac{N\cdot(N-1)}{2}$. Максимальное число пересылок $M_{max} = N - 1$, так как на каждом проходе выполняется обмен найденного минимального элемента с i-м. Вероятность того, что i-й элемент уже стоит на месте, невелика, поэтому средняя оценка M близка к максимальной.

Мы видим, что число сравнений в методе выбора всегда равно максимальному числу сравнений в методе простых включений, в то время как число перемещений, наоборот, минимально. Если вспомнить, что сравниваются ключи позиций, а перемещаются записи целиком (а они могут быть довольно большими), то метод выбора, экономящий число перемещений (хотя бы за счет сравнений) может на практике оказаться предпочтительней.

3.7. Пирамидальная сортировка

При сортировке методом простого выбора на каждом шаге выполняется линейный поиск минимального элемента в неупорядоченной входной части, основанный на поочередном сравнении всех элементов с текущим минимальным. Линейная сложность этого поиска неизбежно делает сложность всей сортировки квадратичной.

Возможно ли найти минимальный элемент за время лучшее линейного? Это означает, что при поиске должны рассматриваться не все элементы входной последовательности (и даже не каждый k-й для любого заданного k). Оказывается, что это возможно, если использовать на каждом следующем шаге информацию о взаимных отношениях элементов, накопленную на предыдущих шагах. Линейный поиск в этом смысле является самым «беспамятным»: например, в сортировке выбором никак не используется то, что после выборки минимального элемента на очередном шаге последовательность остается практически той же

самой: вместо этого все элементы сравниваются заново. Уже известный нам метод бинарного поиска с логарифмической сложностью, наоборот, максимально пользуется фактом линейной упорядоченности всех элементов, однако здесь он не применим напрямую из-за отсутствия упорядоченности.

Тем не менее идея бинарного выбора может быть эффективно применена, если организовать входные данные в виде так называемой *пирамиды* (или *сбалансированного бинарного дерева поиска*) и поддерживать их в этом виде в процессе сортировки. Метод сортировки с использованием такой пирамиды был предложен Р. У. Флойдом в 1964 г. под названием *«Heap sort» — пирамидальной сортировки* или *сортировки кучей*.

Пусть дана последовательность $h_1, ..., h_n$. Элемент h_i *образует пирамиду* в этой последовательности, если выполнены следующие условия:

— если $2i \leq n$, то $h_i \geq h_{2i}$ и h_{2i} образует пирамиду;

— если $2i + 1 \leq n$, то $h_i \geq h_{2i+1}$ и h_{2i+1} образует пирамиду.

Элементы $h_{\frac{n}{2}+1}, ..., h_n$ всегда образуют тривиальные пирамиды, поскольку для них приведенные условия имеют ложные посылки и, стало быть, истинны.

Если элемент h_1 образует пирамиду, то и каждый элемент последовательности образует пирамиду. В этом случае будем говорить, что вся последовательность является *полной пирамидой*.

Полная пирамида может быть изображена в виде корневого бинарного дерева, в котором элементы h_{2i} и h_{2i+1} являются сыновьями элемента h_i. Элемент в любом узле численно не меньше всех своих потомков, а вершина полной пирамиды h_1 содержит максимальный элемент всей последовательности. На рис. 2 показана полная пирамида при $n = 15$.

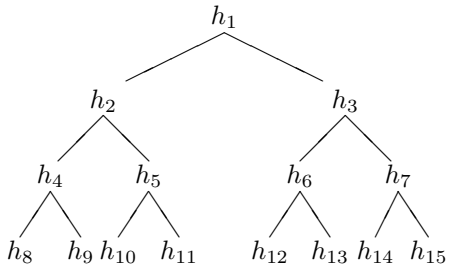


Рис. 2: Полная пирамида при $n = 15$

Если число элементов в полной пирамиде не равно $2^k - 1$, самый нижний уровень дерева будет неполным: недостающих сыновей можно достроить, добавив в пирамиду несколько заключительных «минимальных» элементов «o», не нарушающих условия пирамиды.

Например, последовательность из 12 элементов $12, 11, 7, 8, 10, 6, 3, 2, 1, 5, 9, 4$ является полной пирамидой с вершиной 12 (рис. 3).

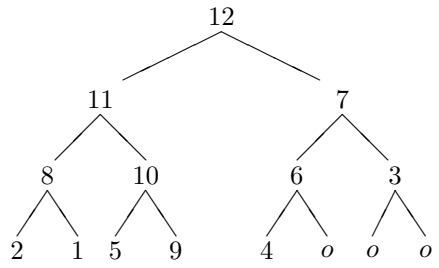


Рис. 3: Полная пирамида при $n = 12$

Очевидно, что последовательность, упорядоченная по убыванию, является полной пирамидой.

Теперь идея метода пирамидальной сортировки такова:

1. Подготовка к сортировке: входная неупорядоченная последовательность перестраивается в пирамиду.

2. Сортировка: входная и готовая последовательности хранятся в одном массиве, причем готовая последовательность формируется *в хвосте* массива, а входная остается в начале массива.

На каждом шаге сортировки первый элемент массива (а это вершина, т. е. максимальный элемент пирамиды, образуемой входной последовательностью) переносится в начало готовой последовательности путем обмена с последним элементом пирамиды, занимающим его место. Затем остаток входной последовательности вновь перестраивается в пирамиду, обеспечивая корректность следующего шага.

Основой реализации метода является следующая процедура *просеивания*. Пусть в последовательности $h_1, ...h_n$ элементы $h_{i+1}, ..., h_n$ уже образуют пирамиды. Требуется перестроить последовательность так, чтобы пирамиду образовывал элемент h_i.

Процедура Просеять (i,n):

пока $2 \cdot i \leq n$ цикл /* цикл просеивания h_i в поддеревья*/

$h_l := h_{2i}$;

если $2 \cdot i + 1 \leq n$ то $h_r := h_{2i+1}$ иначе $h_r := o$;

если $h_i \geq h_l$ и $h_i \geq h_r$ то
выход; /* условие пирамиды в h_i выполнено*/
если $h_l \geq h_r$ то
обменять h_i и h_{2i}; /*просеять в левое поддерево*/
$i := 2 \cdot i$;
иначе если $h_r > o$ то /*просеять в правое, если оно есть*/
обменять h_i и h_{2i+1};
$i := 2 \cdot i + 1$;
конец цикла
конец процедуры

На каждой итерации цикла наибольший из трех элементов h_i, h_{2i} и h_{2i+1} путем обмена оказывается в корне текущего поддерева, что обеспечивает истинность условий пирамиды в этом корне. Если при этом изменяется корень левого или правого поддерева, то просеивание продолжается для него.

Перейдем теперь к детализации пирамидальной сортировки для массива $a_1, ..., a_N$.

Подготовка. Как отмечено выше, элементы $a_{\frac{N}{2}+1}, ..., a_N$ уже изначально образуют пирамиды. Применяя процедуру просеивания поочередно к элементам $a_{[\frac{N}{2}]}, ..., a_1$, мы получаем полную пирамиду. Этот процесс продемонстрирован на примере массива из 10 элементов (на каждом шаге показана цепочка пар сравниваемых и просеиваемых элементов) (рис. 4). Последняя строка является полной пирамидой.

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
Шаг 1, $i = 5$:	52	81	42	23	11	76	91	63	37	20
Шаг 2, $i = 4$:	52	81	42	23	20	76	91	63	37	11
Шаг 3, $i = 3$:	52	81	42	63	20	76	91	23	37	11
Шаг 4, $i = 2$:	52	81	91	63	20	76	42	23	37	11
Шаг 5, $i = 1$:	52	81	91	63	20	76	42	23	37	11
Выход :	91	81	76	63	20	52	42	23	37	11

Рис. 4: Построение пирамиды

Сортировка. Удобно нумеровать шаги в порядке убывания начиная с N. В начале i-го шага элементы $a_1, .., a_i$, по предположению, хранят входную последовательность как пирамиду, а $a_{i+1}, ..., a_N$ – упорядоченную по возрастанию готовую последовательность (изначально пустую).

На i-м шаге текущий максимальный элемент пирамиды a_1 обменивается

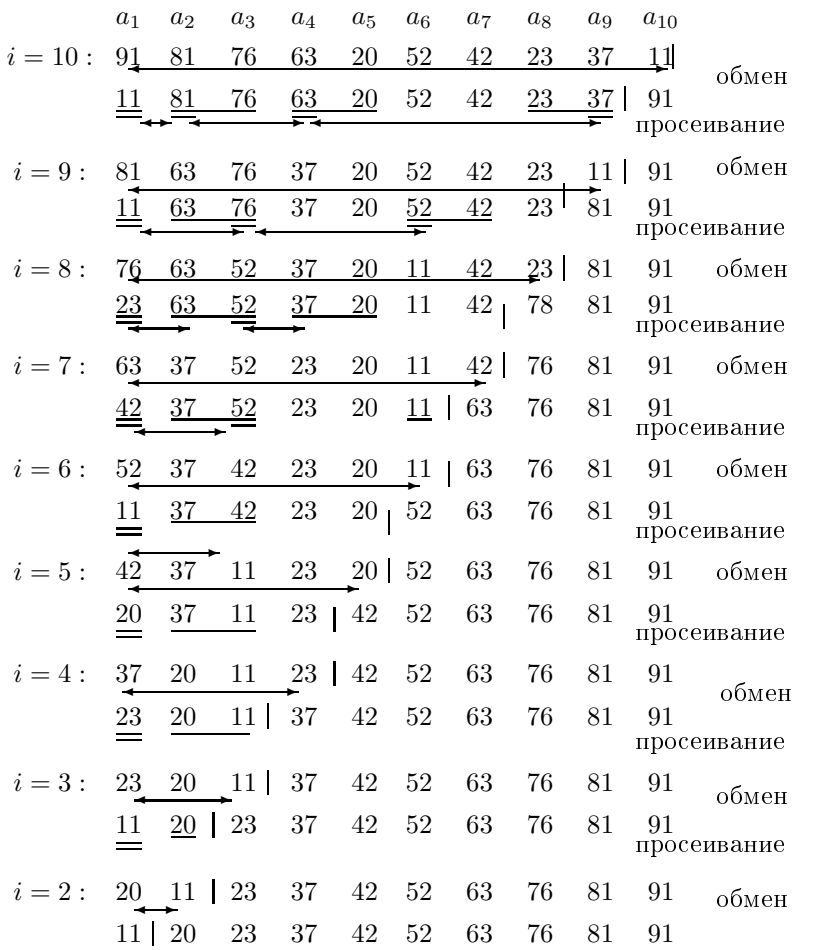


Рис. 5: Сортировка пирамиды

с a_i, становясь началом новой готовой последовательности, где он будет новым минимальным элементом. Входная последовательность (пирамида) при этом претерпевает два изменения:

— она теряет последний элемент (можно считать, что он заменяется на "o"), что не нарушает условий пирамиды ни в одном узле;

— ее первый элемент становится произвольным, что может нарушать условие пирамиды только в первом узле.

Таким образом, для новой входной последовательности $a_1, ..., a_{i-1}$ условия пирамиды выполнены для всех элементов, кроме первого. Применение процедуры просеивания к a_1 восстанавливает полную пирамиду в $a_1, ...a_{i-1}$, что обеспечивает условия осуществимости следующего шага.

Проиллюстрируем процесс сортировки для пирамиды, полученной на подготовительном этапе (рис. 5) (как и прежде, входная последовательность отделяется от готовой вертикальной чертой). Как видно, после обмена на шаге $i = 2$ входная последовательность сокращается до одного элемента: перестраивать ее в пирамиду уже нет смысла, хотя процедура просеивания допускает это. Алгоритм завершает работу после шага $i = 2$. Полностью алгоритм пирамидальной сортировки реализуется следующими функциями на C:

```
void Sift (key a[], int i, int n)
{  int l, r, k;
   key t;
   i++;
   while ((l=2*i) <= n) {
      r = (l+1 <= n)? l+1 : i;
      if ((a[i-1] >= a[l-1]) && (a[i-1] >= a[r-1]))
         return;
      k = (a[l-1] >= a[r-1]) ? l : r;
      t = a[i-1]; a[i-1] = a[k-1]; a[k-1] = t;
      i = k;
   }
}
void heap_sort(key a[], int N)
{  int i; key t;
   for (i = N/2; i >= 0; i--) /* подготовка */
      Sift (a,i, N);
   for(i = N-1; i > 0; i--) { /* сортировка */
      t = a[0];
      a[0] = a[i];
      a[i] = t;
      Sift (a,0, i);         /* просеивание */
   }
}
```

Анализ сложности метода. Число итераций цикла в процедуре просеивания не превосходит высоты пирамиды, т. е. числа уровней в ней. Высота полного бинарного дерева из N узлов, каковым является пирамида, равна $[log_2 N]$. Таким образом, просеивание имеет логарифмическую сложность, как и бинарный поиск.

Оба этапа сортировки реализуются арифметическими циклами, выполняющими просеивание. Стало быть, итоговая сложность пирамидальной сортировки $\sim N \cdot log_2 N$ — даже в худшем случае! Это наиболее быстрая сортировка из рассматриваемых в этом пособии.

Наилучшим расположением элементов для пирамидальной сортировки является обратное упорядочение входной последовательности: в этом случае цикл просеивания работает только один раз и не обменивает элементов.

3.8. Сортировка простым обменом

Классификация методов сортировки достаточно условна. Обмен элементов происходит во всех методах. Но в приведенных выше методах обмен элементов не является основной характеристикой процесса. Теперь рассмотрим метод, основанный на принципе сравнения и обмена пар соседних элементов.

На первом шаге сравним последний и предпоследний элементы, если они не упорядочены, поменяем их местами. Далее проделаем то же со вторым и третьим элементами от конца массива, третьим и четвертым и т. д. до первого и второго с начала массива. При выполнении этой последовательности операций меньшие элементы в каждой паре продвинутся влево, а поскольку две соседние пары имеют общую позицию, наименьший займет первое место в массиве. Повторим этот же процесс от N-го до 2-го элемента, потом от N-го до 3-го и т. д. На последнем проходе сравнивается только одна пара последних элементов; i-й проход по массиву приводит к «всплыванию» наименьшего элемента из входной последовательности на i-е место в готовую последовательность.

Этот метод широко известен под названием сортировки *методом пузырька*. Он реализуется программой:

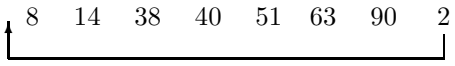
```
void bubble_sort(key a[], int N)
{  int i,j;
   key x;
   for (i=0; i<N-1; i++)
      for (j=N-1; j>i; j--)
```

```
         if (a[j-1]>a[j]) {
            x=a[j]; a[j]=a[j-1]; a[j-1]=x;
         }
}
```

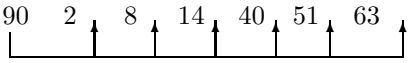
Анализ. Количество сравнений C_i на i-м проходе равно $N-i$, что приводит к уже известному выражению для C: $C = (N-1)+(N-2)+ \ldots + 1 = \frac{N\cdot(N-1)}{2}$. Минимальное количество пересылок $M_{min} = 0$, если массив уже упорядочен, максимальное $M_{max} =$ C, если массив упорядочен по убыванию; в этом случае при каждом сравнении происходит обмен.

3.9. Улучшенная сортировка обменом (шейкер-сортировка)

Нередко случается, что последние проходы сортировки простым обменом работают «вхолостую», так как элементы уже упорядочены. Один из способов улучшения алгоритма сортировки пузырьком состоит в том, чтобы запомнить, производился ли на очередном проходе какой-либо обмен. Если ни одного обмена не было, то алгоритм может закончить работу. Д. Кнут отмечает странную асимметрию метода: один неправильно расположенный «пузырек» на «тяжелом» конце почти отсортированного массива «всплывет» на место за один проход:



Неправильно расположенный «камень» на «легком» конце будет «опускаться» на правильное место только только по одному шажку на каждом проходе:



Поэтому еще одно улучшение алгоритма состоит в том, чтобы чередовать направление проходов. На нечетном проходе будем сравнивать пары от конца массива к началу, в результате «легкие» элементы будут продвигаться к началу массива. На четных проходах будем сравнивать пары от начала массива к концу, в результате «тяжелые» элементы будут продвигаться к концу массива. Левая и правая границы продвижения на каждом проходе будут также поочередно сдвигаться на 1.

Полученный в результате улучшений алгоритм называется *шейкер-сортировкой* и реализуется следующей процедурой:

```
void shaker_sort (key a[], int N)
{  int j,k,l,r;
   int fl;
   key x;
   l = 0;
   r = k = N-1;
   do {
      fl = 1;
      for (j = r; j > l; j--) {
         if (a[j-1] > a[j]) {
            fl = 0;
            x = a[j-1];
            a[j-1] = a[j];
            a[j] = x;
            k = j-1;   /* позиция пузырька */
         }
      }
      l = k+1;
      if (!fl)
         for (j = l+1; j<=r; j++) {
            if (a[j-1] > a[j]) {
               x = a[j-1];
               a[j-1] = a[j];
               a[j] = x;
               k = j;   /* позиция камня */
            }
         }
           r = k-1;
   } while ((l<r) && !fl);
}
```

Переменная fl указывает, были ли сделаны перестановки в цикле. Если перестановок не было, то процесс сортировки можно закончить. Еще одно сделанное улучшение — запоминание на каждом проходе позиции последнего обмена и сдвиг границы сразу до этой позиции.

Анализ. Анализ улучшенного метода довольно сложен. Ясно, что $C_{min} = N - 1$. Кнут показал, что среднее число сравнений пропорцио-

нально $N^2 - N$. Но все предложенные улучшения не влияют на число обменов. В самом деле, каждый обмен уменьшает число инверсий в массиве на 1, следовательно, при любом алгоритме, основанном на обмене пар соседних элементов, число необходимых перестановок одинаково и равно числу инверсий в массиве.

Итак, сортировка обменом и ее улучшенная сортировка хуже, чем сортировка включениями и выбором. Шейкер-сортировку выгодно использовать тогда, когда массив уже почти упорядочен.

3.10. Сортировка с разделением (быстрая сортировка)

Рассмотрим теперь метод сортировки, при котором обмениваются местами пары несоседних элементов, а задача сортировки последовательности в целом изящно рекурсивно сводится к задачам сортировки ее меньших частей.

Допустим сначала, что мы уже переупорядочили некоторым образом элементы входной последовательности, после чего оказалось возможным разделить ее на две непустые части по границе некоторого индекса m: левую (индексы $1...m$) и правую (индексы $m+1...N$); причем все элементы левой части не превосходят всех элементов правой части, т. е.:

$$\forall\, i,j:\ 1 \le i \le m \text{ и } m < j \le N \text{ выполнено}:\ a_i \le a_j. \qquad (14)$$

Индекс m назовем *медианой*. Теперь отсортируем любым методом обмена отдельно левую часть, не затрагивая элементов правой части, а затем отдельно правую, не трогая левой. При этом обмениваться могут только пары элементов, находящиеся в одной части, поэтому никакой обмен не нарушает свойство (14). Значит, оно будет верно и для результирующей последовательности, которая в силу этого оказывается упорядоченной в целом.

Таким образом, после разбиения входной последовательности на две части задача ее сортировки в целом сводится к задаче сортировки двух ее частей. Это естественный повод применить рекурсию: каждую из выделенных частей также разделить на две части, рекурсивно отсортировать их и т. д.

Схема рекурсивного решения описывается следующей процедурой:

процедура СортировкаРазделением (l, r)
  /* l, r — границы сортируемой подпоследовательности */

/* Разделение */
привести подпоследовательность $a_l..a_r$ к условию (14)
и определить медиану m;
/* Рекурсивный спуск */
если $l < m$ то /* части длины 0 и 1 не сортируем */
СортировкаРазделением (l, m);
если $m + 1 < r$ то /* части длины 0 и 1 не сортируем */
СортировкаРазделением $(m + 1, r)$;
конец процедуры

Во избежание «зацикливания» на рекурсивном спуске, следует обеспечить, чтобы разделяемые последовательности строго укорачивались. Это выполняется, если $l \leq m < r$. Но уже нет смысла сортировать и тривиальные последовательности длины 1. Соответствующие условия обрыва рекурсивного спуска отражены в процедуре.

Получается, что в процессе дробления исходной задачи на подзадачи мы приходим к тривиальным подзадачам (сортировке последовательностей длины 0 и 1), которые решать и не требуется. Не приходится ничего делать и для слияния решений подзадач в решение исходной задачи во время возврата из рекурсии: упорядоченные последовательности образуются сами собой по мере упорядочения их частей.

Где же тогда фактически выполняется сортировка, т. е. сравнение и обмен элементов? На фазе разделения, иллюстрируя, как хорошая подготовка условий для решения зачастую уже и дает решение! К детализации фазы разделения сейчас и переходим.

В качестве критерия разделения нам понадобится так называемый *пилотируемый* элемент x. В классической версии алгоритма в качестве x выбирается произвольный элемент сортируемой последовательности: первый, последний, расположенный в середине или иначе. Влияние его выбора на эффективность алгоритма мы обсудим ниже.

В процессе разделения мы соберем в левой части последовательности все элементы $a_i \leq x$, а в правой — все элементы $a_j \geq x$. Условие (14) при этом будет выполнено даже при возможном наличии одинаковых элементов x в обеих частях.

Введем два бегущих индекса-указателя i и j, которые делят разделяемую подпоследовательность на три участка: левый $(a_l...a_{i-1})$, правый $(a_{j+1}...a_r)$ и средний $(a_i...a_j)$. В левом и правом участках будут накапливаться элементы левой и правой частей, подлежащих затем рекурсивной сортировке, а в среднем находятся остальные, еще не распределенные элементы.

Процесс разделения теперь выглядит так:
$i := l; j := r;$
цикл
  пока $a_i < x$ цикл /* проверка $i \le r$ не нужна:
                x есть где-то в конце */
    $i := i + 1$; /* в конце $a_i \ge x$*/
  конец цикла;
  пока $x < a_j$ цикл /* проверка $j \ge l$ не нужна:
                x есть где-то в начале */
    $j := j - 1$; /* в конце $a_j \le x$ */
  конец цикла;
  если $i \le j$ то /* если $i = j, a[i = j] = x$ и
                нужен сдвиг индексов для выхода из цикла */
    обменять a_i и a_j; /* теперь $a_i \le x \le a_j$ */
    $i := i + 1$; /* на случай $a_i = x$: добавить в левую часть */
    $j := j - 1$; /* на случай $a_j = x$: добавить в правую часть */
пока $i < j$

Циклы по встречным индексам переносят из средней части в левую или правую элементы, строго меньшие или большие x, которые могут быть добавлены в эти части без перестановки. После их выполнения процесс разделения либо заканчивается (если $i \ge j$), либо пара a_i и a_j образует инверсию. В последнем случае их следует обменять (вот где происходят упорядочивающие обмены с уменьшением числа инверсий в последовательности!) и включить в левую и правую части. Это включение можно было бы оставить до следующей итерации внешнего цикла, но элементы (любой или даже оба) могут быть равны x.

Проверка того, что бегущие индексы не выходят за границы $l...r$, строго говоря, необходима, но фактически не нужна: на первом проходе выход за границы немозможен, так как в массиве есть сам элемент x и оба цикла остановятся на нем. В конце же первого прохода происходит обмен элементов и обе части становятся не пусты, что гарантирует остановку циклов по встречным индексам в пределах интервала $l...r$ и на следующих проходах.

Цикл оканчивается при $i \ge j$, однако нам еще надо определить медиану. Строго говоря, определенные границы левой части – $l...i-1$, правой – $j+1...r$, однако интервал $j+1...i-1$ может быть не вырожден и заполнен элементами x (почему?). Эти элементы останутся на своих местах в процессе сортировки (почему?), поэтому их можно исключить

из левой и правой частей. Окончательно границами левой части можно считать $l...j$, а правой — $i...r$.

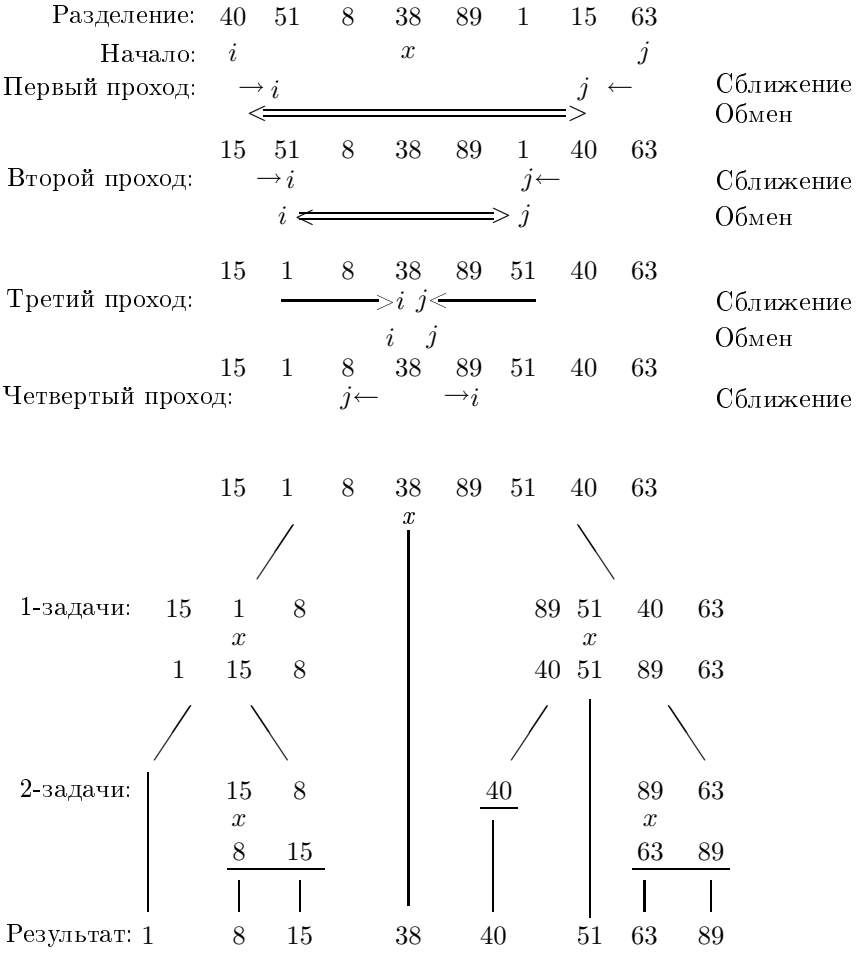


Рис. 6: Разделение

Сводя вместе обе части алгоритма: разделение и рекурсивный спуск, получаем следующую программу, где в качестве пилотируемого элемента выбирается срединный элемент:

```
void quicksort (key a[], int l, int r)
{  key x, w;
   int i, j;
   i = l; j = r;
   x = a[(l+r)/2];
   do {
      while (a[i] < x) i++;
      while (x < a[j]) j--;
      if (i <= j) {
         w = a[i];
         a[i] = a[j];
         a[j] = w;
         i++;
         j--;
      }
   } while (i<j);
   if (l<j) quicksort (a, l, j);
   if (i<r) quicksort (a, i, r);
}
```

Описанный метод обменной сортировки разделением ее изобретатель Ч. Э. Р. Хоар назвал *быстрой сортировкой*. Ее работа показана на рис. 6.

Анализ. Оказалось, что усовершенствование обменной сортировки дает лучший из известных до сего времени метод сортировки массивов. Проанализируем свойства быстрой сортировки. После того как выбран x, процессу разбиения подвергается весь массив, таким образом, выполняется ровно N сравнений. Число обменов можно оценить следующим образом: пусть массив имеет N элементов, x выбран в качестве пилотируемого элемента. Пусть после разделения x будет занимать в массиве позицию k. Число требующихся обменов равно числу элементов в левой части массива $(k-1)$, умноженному на вероятность того, что элемент нужно обменять. Элемент обменивается, если он не меньше, чем x. Вероятность этого равна $\frac{N-(k-1)}{N}$. Просуммируем всевозможные варианты выбора медианы и разделим эту сумму на N, в результате получим ожидаемое число обменов:

$$\mathrm{M} = \frac{1}{N} \cdot [\sum_{i=1}^{N} (x-1) \cdot \frac{(N-x+1)}{N}] = \frac{N}{6} - \frac{1}{6 \cdot N}.$$

Следовательно, ожидаемое число обменов равно приблизительно $\frac{N}{6}$. В лучшем случае, когда в качестве медианы всегда выбирается элемент, который меньше половины элементов или равен ей и больше другой половины или равен ей. Каждое разделение разбивает массив на две равные части, а число проходов, необходимых для сортировки, равно $\log_2 N$. Тогда общее число сравнений равно $N \log_2 N$, а общее число обменов – порядка $\frac{N}{6} \log_2 N$.

Однако в худшем случае сортировка становится «медленной», например, когда в качестве пилотируемого элемента всегда выбирается наибольшее значение. Тогда в результате разбиения в левой части оказывается $N-1$ элемент, т. е. массив разбивается на подмассивы из одного элемента и из $N-1$ элемента. В этом случае вместо $\log_2 N$ разбиений необходимо сделать $\sim N$ разбиений. В результате в худшем случае оценка оказывается $\sim N^2$, что гораздо хуже пирамидальной сортировки.

3.11. Сортировка методом подсчета

При сортировке подсчетом каждый элемент поочередно сравнивается со всеми остальными и подсчитывается количество элементов, которые меньше его. Это число (плюс единица) определяет позицию элемента в отсортированной последовательности при условии, что все элементы различны.

Простейшая реализация этого метода требует дополнительного массива, в котором накапливаются отсортированные элементы по мере определения их позиций. Это связано с тем, что в данном методе входная последовательность не сокращается по мере обработки элементов, а память под готовую последовательность должна быть зарезервирована в самом начале. Поэтому не удается объединить их в пределах памяти, занимаемой одной последовательностью, как мы это делали раньше.

Таким образом, требуемая этим методом память — $2N$ элементов, количество выполненных сравнений равно N^2, а количество перемещений элементов (при расстановке) равно N.

По этим характеристикам метод подсчета оказывается наихудшим из всех рассмотренных, если только мы не оптимизируем число перемещений.

4. Сортировка файлов

В этом разделе мы вспомним о структурах последовательного доступа (файлах и списках), для которых невозможен прямой доступ к элементам без предварительного перебора предшествующих им. Это свойство делает практически неприменимыми ранее рассмотренные методы сортировки и требует иных подходов.

4.1. Слияние последовательностей

Под *слиянием* будем понимать объединение двух или более упорядоченных последовательностей (например, по возрастанию) в одну упорядоченную. Это можно сделать следующим образом: сравнить наименьшие элементы из упорядоченных последовательностей и наименьший из них перенести в готовую последовательность (с продвижением позиции считывания той последовательности, из которой выбран элемент). Далее снова сравнить начала последовательностей и наименьший из этих элементов добавить в готовую последовательность и т. д. Как только одна из последовательностей закончится, она исключается из рассмотрения; когда остается только одна последовательность, ее «хвост» можно просто переместить в готовую.

Например, требуется объединить два следующих файла в третий (позиция считывания отмечена чертой):

```
|8  38  40  51 75
|1  15  63  89 101  107
```

Сравним первые элементы отсортированных файлов 8 и 1, наименьший из них запишем в выходной файл:

```
 |8  38  40  51
 1|15  63  89  101  107
 1|
```

На следующем шаге сравниваются 8 и 15:

```
8| 38  40  51
1|15  63  89  101  107
1  8|
```

Этот процесс продолжится до тех пор, пока все элементы первого и второго файлов не будут переписаны в третий в заданном порядке.

В результате получим отсортированный по возрастанию файл:

1 8 15 38 40 51 63 89 101 107

Метод слияния — один из самых первых методов, который естественным образом можно применить к сортировке файлов, а именно два отсортированных (например, по возрастанию) файла слить в третий отсортированный (по возрастанию). Данный метод слияния был предложен фон Нейманом в 1945 г. и предназначался именно для сортировки файлов.

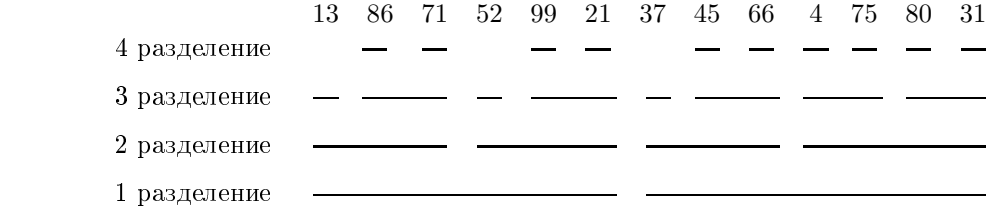
4.2. Сортировка массива простым двухпутевым слиянием

Идея метода сортировки слиянием такова: разделим входную последовательность на две части, отсортируем каждую из них по отдельности, а результаты сольем, как описано выше. Подобно сортировке разделением, исходная задача сводится к двум аналогичным задачам с меньшим объемом данных, и тут уместно применить рекурсию:

— на фазе рекурсивного спуска каждая из образующихся последовательностей делится на две части до тех пор, пока не образуются последовательности длины 0 или 1, которые сортировать не надо;

— на фазе возврата из рекурсии пары уже отсортированных подпоследовательностей сливаются.

Рассмотрим сначала сортировку слиянием для массива на примере последовательности:

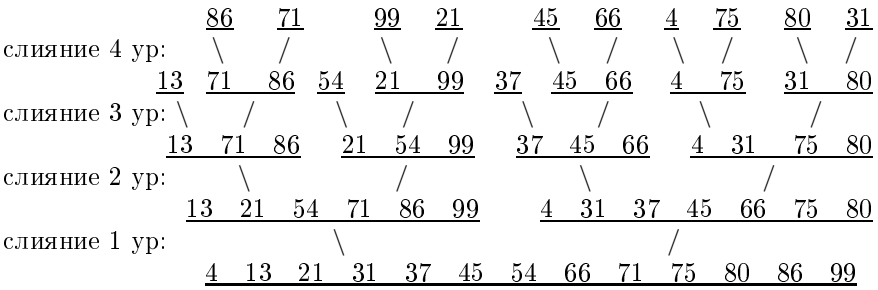


где также видно дерево разделений.

Мы можем делить каждую подпоследовательность на части разной длины: алгоритм слияния это допускает. Но из анализа быстрой сортировки мы уже знаем, что наименьшая высота дерева разделений — $log_2 N$, определяющая глубину рекурсивного спуска, достигается при делении строго пополам. В отличие от быстрой сортировки, где из-за

пилотируемого элемента разделение не всегда происходит по середине последовательности, здесь нам ничто не мешает выбрать индекс разделения наиболее близко к середине.

Слияние элементарных последовательностей начинается из глубины рекурсии и дает следующие подпоследовательности:



Следующая пара процедур реализует сортировку слиянием для массивов:

```
void merge (key a1[], int len1, key a2[], int len2, key ar[])
/* Слияние отсортированных массивов a1 длины len1 и a2
/* длины ken2 в массив ar */
{  int i=0, j=0, k=0;
   key x;
   while ((i<len1) || (j<len2)) {    /* пока есть элементы */
      if (i==len1)          /* 1-я кончилась: берем из 2-й */
         x = a2[j++];
      else if (j==len2)     /* 2-я кончилась: берем из 1-й */
         x = a1[i++];
      else if (a1[i]<a2[j]) /* выбираем из той, */
         x = a1[i++];       /* где наименьший */
      else
         x = a2[j++];
      ar[k++] = x;
   }
}
```

```
static key aw[N];  /* вспомогательный глобальный */
                   /* массив для слияния */

void sort_merging (key a[], int L, int R)
/* L,R - границы сортируемой части массива a */
{  int i,M;
   M = (L+R)/2;
   if (L < M)
      sort_merging (a, L, M);
   if (M+1 < R)
      sort_merging (a, M+1, R);
         /* слияние частей в aw */
   merge (&a[L], M-L+1, &a[M+1], R-M, &aw[L]);
         /* копирование в исход.фрагмент */
   for (i=L; i<=R; i++)
      a[i] = aw[i];
}
```

Анализ. Для слияния двух отсортированых частей необходим третий массив-результат aw: как и при сортировке подсчетом, здесь нет возможности высвобождать место под готовую последовательность в пределах двух входных. Однако, поскольку упорядоченные данные должны накапливаться для последующего слияния в исходном массиве, приходится дополнительно переписывать результат на место исходной подпоследовательности.

Нам было бы достаточно иметь в качестве aw локальный рабочий массив длины $R - L + 1$, но в Си невозможно описать массив переменной длины (без обращения к более медленным средствам динамической памяти). Введение же локального массива максимальной длины N (используемого лишь частично) привело бы к затратам памяти до $Nlog_2N$ записей, так как на каждом из log_2N уровней рекурсии в памяти хранился бы отдельный рабочий массив длины N. Использование глобального массива приводит к оценке затрат памяти в данном методе $\sim 2N$ записей и в данном случае организовано корректно: ни один элемент массива aw, записанный в процедуре слияния, не может быть изменен до переписи его в массив a в процедуре сортировки, а после переписи он становится не нужен. (Вообще, использование глобальных переменных в рекурсивных процедурах требует чрезвычайной осторожности и точного понимания того, как происходит обращение к ним из возможно нескольких активных версий одной и той же процедуры.)

Количество сравнений и перемещений элементов оценивается так же, как мы это делали для быстрой сортировки: на каждом уровне рекурсии происходит два перемещения каждого элемента $aw[i]$ — запись в него при слиянии подпоследовательностей и запись из него в исходный массив; каждой записи в массив предшествует не более одного сравнения с другим элементом при слиянии (некоторые элементы переписываются без сравнения). Таким образом, $C_{max} = Nlog_2N$, $M_{max} = 2Nlog_2N$, что ставит данный метод в ряд наиболее быстрых методов сортировки.

4.3. Сортировка файла простым двухпутевым слиянием

Пусть теперь вместо массива a дан файл f, который нужно отсортировать. Заметим, что в процедуре слияния *merge* доступ к элементам частей массива и к массиву-результату исключительно последовательный: индексы-указатели текущего доступа сдвигаются только на единицу вперед, без возвратов и скачков. Поэтому операции вида $a[i++]$ для массива можно заменить на типовые операции чтения и записи элемента файла с продвижением к позиции следующего элемента.

При разделении массива нам не приходилось явно отводить память под образуемые части и переписывать в них элементы. Вместо этого мы устанавливали и перемещали два указателя. Однако файл читать можно только по одному указателю, поэтому разделяемые части придется явно переписывать в отдельные файлы. Таким образом, нужна процедура *split*, выполняющая физическое разделение.

Наконец, для разделения массива пополам мы пользовались знанием его длины. Для файла число его записей не всегда известно и определение длины требует дополнительного холостого считывания, которое нежелательно. Это препятствие мы устраним так: поскольку разделяются еще неотсортированные файлы, разделение можно организовать подобно тому, как сдается колода карт на двух игроков: элементы разделяемого файла по мере (однократного) считывания переписываются в два новых файла поочередно. Концом «раздачи» является достижение конца входного файла, при этом количество элементов в новых файлах отличается максимум на единицу, что и требуется.

По новым правилам фаза разделения приводит к образованию следующих сортируемых файлов на разных стадиях (обратите внимание на несущественное перемешивание входной последовательности):

	13	86	71	52	99	21	37	45	66	4	75	80	31
1 разделение	13	71	99	37	66	75	31	86	52	21	45	4	80
2 разделение	13	99	66	31	71	37	75	86	21	4	52	45	80
3 разделение	13	66	99	31	71	75	37	86	4	21	52	80	45
4 разделение	13	66	99	31	71	75	37	86	4	21	52	80	45

Следующие процедуры реализуют все описанные модификации. Мы пользуемся стандартными файловыми функциями библиотеки Си, в том числе средствами создания промежуточных рабочих файлов, для которых не нужно беспокоиться о выборе уникальных имен (они устроены так, что могут даже не появляться на диске). Обратите внимание на необходимость явного открытия (или создания), закрытия и перемотки всех файлов.

```
/* упрощенные вызовы файловых функций C */
#define fget(f,x) fread (&x, sizeof(x), 1, f)
#define fput(f,x) fwrite(&x, sizeof(x), 1, f)

bool split (FILE *f, FILE *f1, FILE * f2)
/* Разделение f: перепись элементов нечетных позиций в f1, */
/* четных - в f2 */
{  key x;
   int n=0;       /* счетчик длины файла */
   rewind (f);    /* возврат к началу разделяемого файла */
   fget (f, x);
   while (!feof(f)) {
      /* (feof срабатывает ПОСЛЕ попытки чтения!) */
      fput (f1, x);
      fget (f, x);
      if (!feof(f)) {
         fput (f2, x);
         fget (f, x);
      }
      n++;
   }
   return n>1;   /* false (длина 0 или 1) сигнализирует */
}                /* о прекращении разделения */

void merge (FILE *f1, FILE *f2, FILE *fr)
/* Слияние f1 и f2 в fr */
```

```
{  key x1, x2;
   rewind (f1);  /* перемотка к началу всех файлов */
   rewind (f2);
   rewind (fr);
   fget (f1, x1);
   fget (f2, x2);
   while (!feof(f1) || !feof(f2)) {
      if (feof(f1)) {
         fput (fr, x2);
         fget (f2, x2);
      } else if (feof(f2)) {
         fput (fr, x1);
         fget (f1, x1);
      } else if (x1<x2) {
         fput (fr, x1);
         fget (f1, x1);
      } else {
         fput (fr, x2);
         fget (f2, x2);
      }
   }
}
void sort_merge (FILE *f)
/* Главная процедура сортировки, */
/* входной файл должен быть открыт */
{  /* создание временных файлов для частей */
   FILE *f1 = tmpfile(),
        *f2 = tmpfile();
       /* разделение на фазе спуска в рекурсию */
   if (split (f, f1, f2)) {
      sort_merge (f1);
      sort_merge (f2);
   }
       /* слияние на фазе возврата из рекурсии */
   merge (f1, f2, f);
       /* закрытие и удаление рабочих файлов */
   fclose (f1);
   fclose (f2);
}
```

```
void main ()
{  key x;
   FILE * f = fopen("inputfile","r+b"); /* открытие файла */
   sort_merge (f);  /* сортировка открытого файла */
   fclose (f);      /* закрытие выходного файла */
}
```

Анализ. Все оценки числа сравнений и перемещений (при чтении и записи) элементов для сортировки файлов остаются теми же, что и для массивов.

Количество используемой внешней памяти, равное суммарной длине всех одновременно существующих файлов, подсчитывается следующим образом. Исходный файл существует всегда, в нем N элементов. Оба файла 1-го уровня после разделения тоже существуют всегда, вплоть до слияния — в них тоже N элементов. Файлы 2-го уровня существуют не одновременно: те, которые получаются разделением 1-го файла 1-го уровня, уничтожаются после слияния, и при переходе ко 2-му файлу 1-го уровня занимаемая ими память может быть переиспользована. Таким образом, на 2-м уровне всегда хранится около $N/2$ элементов. Аналогично, на 3-м уровне хранится $N/4$ элемента, на k-м — $N/2^{k-1}$, на последнем $[log_2 N]$ уровне — 1 элемент. Окончательно на самом глубоком уровне рекурсии в памяти может находиться $1+2+4+...+N/2+N+N \sim 3N$ элементов.

Перемещение (возможно громоздких) записей во внешней памяти может занять значительное время, намного большее, чем выполнение гораздо большего числа перемещений во внутренней, поэтому для внешней сортировки оптимизация именно этот фактора представляет особый интерес. Поэтому еще во времена, когда файлы располагались на магнитофонных лентах, а «перемотка» означала действительную перемотку катушки, был придуман двухуровневый способ организации файлов: в виде файла записей, содержащего собственно данные, и *индекс-файла*, содержащего короткие записи-пары «ключ + ссылка». Ключи, как мы помним, это только та информация, сравнением которой определяется относительный порядок записей, а ссылки — это позиции в основном файле записей, соответствующих ключам. При всех операциях поиска и сортировки обрабатываются более короткие индекс-файлы, а доступ к основному файлу выполняется один раз, когда позиция требуемой записи становится определена. Добавление записи в файл всегда означало дописывание ее в конец основного файла. При удалении же записи обычно удалялся только индекс из индекс-файла, а запись в файле

только помечалась как удаленная («дырка»). Специально проводимая операция уплотнения файла (копирование с удалением «дырок») выполнялась во время техобслуживания устройств, когда работы на машине не производились.

Современные машины, конечно, намного производительнее, но возросли и объемы обрабатываемых данных. Поэтому технологии индексированных файлов применяются в системах обработки данных до сих пор.

4.4. Заключение

Из представленного материала ясно, что все простые методы сортировки имеют быстродействие $\sim N^2$, а усовершенствованные методы — $Nlog_2N$. Сортировка методом *пузырька* является наихудшей среди всех представленных в данном пособии. Ее улучшеный вариант — *шейкер*-сортировка хуже сортировок простыми включениями и выбором, кроме случая, когда входная последовательность уже «почти» отсортирована. *Быстрая* сортировка в общем случае является наилучшей. Кнут показал, что на огромном количестве представленных примеров она превосходит *пирамидальную* в $1,5$ раза. Она сортирует последовательность с элементами, расположенными в обратном порядке, так же хорошо, как и в остальных случаях. Тем не менее остается проблема наихудшего случая, когда *быстрая* сортировка становится «медленной». *Пирамидальная* сортировка имеет стабильную оценку. Для нее более всего подходят случаи, когда элементы расположены «почти» в обратном порядке.

Имеет место следующая теорема. **В любом алгоритме, упорядочивающем с помощью сравнений пар, на упорядочение последовательности из N элементов тратится не меньше $c \cdot N \cdot \log_2 N$ сравнений при $c > 0, N \longrightarrow \infty$.**

Обоснование. Для заданной последовательности из N элементов может быть построено $N!$ перестановок. Алгоритм сортировки, устанавливающий путем сравнения пар, какая из этих перестановок является единственно правильным решением, фактически осуществляет спуск по так называемому *дереву решений* — двоичному дереву, листьями которого являются решения, а узлами — условия, позволяющие сузить выбор.

Для последовательности из трех элементов a, b, c алгоритм, упорядочивающий с помощью сравнений, можно представить в виде дерева решений, как это показано на рис. 7.

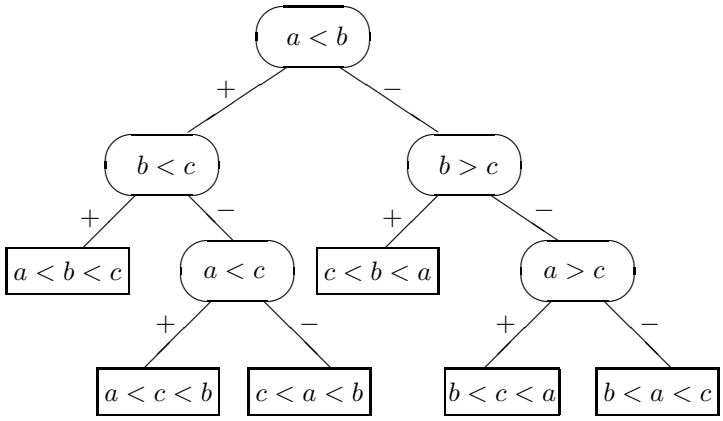


Рис. 7: Дерево решений для последовательности из трех элементов

Для этой последовательности можно построить шесть различных перестановок, которые являются листьями в представленном дереве решений.

Для задачи сортировки в дереве решений должно быть $N!$ листьев. Значит, высота дерева решений $\geq log_2 N!$ Из неравенства

$$N! \geq N \cdot (N-1) \cdot (N-2) \cdot ... \cdot [\frac{N}{2}] \geq (\frac{N}{2})^{\frac{N}{2}}$$

следует заключение теоремы, так как

$$log_2 N! \geq (\frac{N}{2}) \cdot log_2 \frac{N}{2} \geq (\frac{N}{4}) \cdot log_2 N \quad (\text{при } N \geq 4).$$

А количество сравнений, которые необходимо сделать, чтобы получить любую из перестановок примера, не меньше 2 ($log_2 6 \approx 2.5$), что и видно на рис. 7.

В данном учебном пособии рассмотрены алгоритмы поиска и сортировки для массивов и файлов. Не рассмотрены алгоритмы для таких структур данных, как списки и деревья: по мнению авторов, читатель пока недостаточно знаком с понятием *указателя*. Предполагается, что эти вопросы будут рассмотрены в следующем учебном пособии, ориентированном на изучение динамической организации структур данных и алгоритмов обхода, поиска, вставки элементов в такие объекты.

Список литературы

[1] *Кнут Д.* Искусство программирования для ЭВМ. М., 1978. Т. 1: Основные алгоритмы.

[2] *Кнут Д.* Искусство программирования для ЭВМ. М., 1978. Т. 3: Сортировка и Поиск.

[3] *Дейкстра Э.* Дисциплина программирования. М., 1978.

[4] *Кормен Т. и др.* Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. М., 2001.

[5] *Вирт Н.* Алгоритмы + структуры данных = программы. М., 1985.