### 1.12.2. Поиск делением пополам (двоичный поиск)

Совершенно очевидно, что других способов убыстрения поиска не существует, если, конечно, нет еще какой-либо информации о данных, среди которых идет поиск. Хорошо известно, что поиск можно сделать значительно более эффективным, если данные будут упорядочены. Вообразите себе телефонный справочник, в котором фамилии не будут расположены по порядку. Это нечто совершенно бесполезное Поэтому мы приводим алгоритм, основанный на знании того, что массив a упорядочен, т. е. удовлетворяет условию

$$Ak:\ 1 \leqslant k < N:\ a_{k-1} \leqslant a_k \tag{1.39}$$

Основная идея — выбрать случайно некоторый элемент, предположим $a_m$, и сравнить его с аргументом поиска x. Если он равен x, то поиск заканчивается, если он меньше x, то мы заключаем, что все элементы с индексами, меньшими или равными m, можно исключить из дальнейшего поиска; если же он больше x, то исключаются индексы больше и равные m. Это соображение приводит нас к следующему алгоритму (он называется *«поиском делением пополам»*). Здесь две индексные переменные L и R отмечают соответственно левый и правый конец секции массива a, где еще может быть обнаружен требуемый элемент.

```
L := 0; R := N-1; found := FALSE;                    (1.40)
WHILE (L ≤ R) & ~found DO
  m := любое значение между L и R;
  IF a[m] = x THEN found := TRUE
  ELSIF a[m] < x THEN L := m+1
  ELSE R := m-1
  END
END
```

Инвариант цикла, т. е. условие, выполняющееся перед каждым шагом, таков:

$$(L \leq R)\ \&\ (Ak: 0 \leq k < L: a_k < x)\ \&\ (Ak: R < k < N: a_k > x) \tag{1.41}$$

из чего выводится результат

$$\text{found OR } ((L > R) \,\&\, (\forall k : 0 \le k < L : a_k < x) \,\&\, (\forall k : R < k < N : a_k > x))$$

откуда следует

$$(a_m = x) \text{ OR } (\exists k : 0 \le k < N : a_k = x)$$

Выбор m совершенно произволен в том смысле, что корректность алгоритма от него не зависит. Однако на его эффективность выбор влияет. Ясно, что наша задача — исключить на каждом шагу из дальнейшего поиска, каким бы ни был результат сравнения, как можно больше элементов. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива. В результате максимальное число сравнений равно $\log N$, округленному до ближайшего целого. Таким образом, приведенный алгоритм существенно выигрывает по сравнению с линейным поиском, ведь там ожидаемое число сравнений — $N/2$.

Эффективность можно несколько улучшить, поменяв местами заголовки условных операторов. Проверку на равенство можно выполнять во вторую очередь, так как она встречается лишь единожды и приводит к окончанию работы. Но более существенно следующее соображение: нельзя ли, как и при линейном поиске, отыскать такое решение, которое опять бы упростило условие окончания. И мы действительно находим такой быстрый алгоритм, как только отказываемся от наивного желания кончить поиск при фиксации совпадения. На первый взгляд это кажется странным, однако при внимательном рассмотрении обнаруживается, что выигрыш в эффективности на каждом шаге превосходит потери от сравнения с несколькими дополнительными элементами. Напомним, что число шагов в худшем случае — $\log N$. Быстрый алгоритм основан на следующем инварианте:

$$(\forall k : 0 \le k < L : a_k < x) \,\&\, (\forall k : R \le k < N : a_k \ge x) \qquad (1.42)$$

причем поиск продолжается до тех пор, пока обе секции не «накроют» массив целиком.

```
L := 0; R := N;                                          (1.43)
WHILE L < R DO
  m := (L+R) DIV 2;
  IF a[k] < x THEN L := m+1 ELSE R := m END
END
```

Условие окончания — $L \geqslant R$, но достижимо ли оно? Для доказательства этого нам необходимо показать, что при всех обстоятельствах разность $R - L$ на каждом шаге убывает. В начале каждого шага $L < R$. Для среднего арифметического m справедливо условие $L \leqslant m < R$. Следовательно, разность действительно убывает, ведь либо L увеличивается при присваивании ему значения $m + 1$, либо R уменьшается при присваивании значения m. При $L = R$ повторение цикла заканчивается. Однако наш инвариант и условие $L = R$ еще не свидетельствуют о совпадении. Конечно, при $R = N$ никаких совпадений нет. В других же случаях мы должны учитывать, что элемент $a[R]$ в сравнениях никогда не участвует. Следовательно, необходима дополнительная проверка на равенство $a[R] = x$. В отличие от первого нашего решения (1.40) приведенный алгоритм, как и в случае линейного поиска, находит совпадающий элемент с наименьшим индексом.

### 1.12.3. Поиск в таблице

Поиск в массиве иногда называют *поиском в таблице*, особенно если ключ сам является составным объектом, таким, как массив чисел или символов. Часто встречается именно последний случай, когда массивы символов называют строками или словами. *Строковый* тип определяется так:

```
String = ARRAY[0 .. M - 1] OF CHAR                       (1.44)
```

соответственно определяется и отношение порядка для строк:

$$(x = y) = (\mathbf{A}j: 0 \leq j < M: x_j = y_j)$$
$$(x < y) = \mathbf{E}i: 0 \leq i < N: ((\mathbf{A}j: 0 \leq j < i: x_j = y_j) \,\&\, (x_i < y_i))$$

большие расстояния, будет связана лишь с одним-единственным элементом.

### 2.2.2. Сортировка с помощью прямого выбора

Этот прием основан на следующих принципах:

1. Выбирается элемент с наименьшим ключом.
2. Он меняется местами с первым элементом $a_1$.
3. Затем этот процесс повторяется с оставшимися n — 1 элементами, n — 2 элементами и т. д. до тех пор, пока не останется один, самый большой элемент.

Процесс работы этим методом с теми же восемью ключами, что и в табл. 2.1, приведен в табл. 2.2. Алгоритм формулируется так:

```
FOR i := 1 TO n-1 DO
  присвоить k индекс наименьшего из a[i]... a[n];
  поменять местами a[i] и a[k];
END
```

Такой метод — его называют *прямым выбором* — в некотором смысле противоположен прямому включению. При прямом включении на каждом шаге рассматриваются только *один* очередной элемент исходной последовательности и *все* элементы готовой последовательности, среди которых отыскивается точка включения; при прямом выборе для поиска *одного* элемента с наименьшим ключом просматриваются все элементы исходной последовательности и найденный помещается как очередной элемент в готовую последовательность. Полностью алгоритм прямого выбора приводится в прогр. 2.3.

Таблица 2.2. Пример сортировки с помощью прямого выбора

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Начальные ключи | 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 |
| | 06 | 55 | 12 | 42 | 94 | 18 | 44 | 67 |
| | 06 | 12 | 55 | 42 | 94 | 18 | 44 | 67 |
| | 06 | 12 | 18 | 42 | 94 | 55 | 44 | 67 |
| | 06 | 12 | 18 | 42 | 94 | 55 | 44 | 67 |
| | 06 | 12 | 18 | 42 | 44 | 55 | 94 | 67 |
| | 06 | 12 | 18 | 42 | 44 | 55 | 94 | 67 |
| | 06 | 12 | 18 | 42 | 44 | 55 | 67 | 94 |

```
PROCEDURE StraightSelection;
  VAR i, j, k: index; x: item;
BEGIN
  FOR i := 1 TO n-1 DO
    k := i; x := a[i];
    FOR j := i+1 TO n DO
      IF a[j] < x THEN k := j; x := a[k] END
    END;
    a[k] := a[i]; a[i] := x
  END
END StraightSelection
```

Прогр. 2.3. Сортировка с помощью прямого выбора.

*Анализ прямого выбора.* Число сравнений ключей (C), очевидно, не зависит от начального порядка ключей. Можно сказать, что в этом смысле поведение этого метода менее естественно, чем поведение прямого включения. Для C имеем

$$C = (n^2 - n)/2$$

Число перестановок минимально

$$M_{min} = 3 * (n - 1) \tag{2.6}$$

в случае изначально упорядоченных ключей и максимально

$$M_{max} = n^2/4 + 3 * (n - 1)$$

если первоначально ключи располагались в обратном порядке. Для того чтобы определить $M_{avg}$, мы должны рассуждать так. Алгоритм просматривает массив, сравнивая каждый элемент с только что обнаруженной минимальной величиной; если он меньше первого, то выполняется некоторое присваивание. Вероятность, что второй элемент окажется меньше первого, равна 1/2, с этой же вероятностью происходят присваивания минимуму. Вероятность, что третий элемент окажется меньше первых двух, равна 1/3, а вероятность для четвертого оказаться наименьшим — 1/4 и т. д. Поэтому полное ожидаемое число пересылок равно $H_n - 1$, где $H_n$ — n-е гармоническое число:

$$H_n = 1 + 1/2 + 1/3 + \ldots + 1/n \tag{2.7}$$

$H_n$ можно выразить и так:

$$H_n = \ln n + g + 1/2n - 1/12n^2 + \ldots \quad (2.8)$$

где $g = 0.577216 \ldots$ — константа Эйлера. Для достаточно больших n мы можем игнорировать дробные составляющие и поэтому аппроксимировать среднее число присваиваний на i-м просмотре выражением

$$F_i = \ln i + g + 1$$

Среднее число пересылок $M_{avg}$ в сортировке с выбором есть сумма $F_i$ с i от 1 до n:

$$M_{avg} = n * (g + 1) + (Si: 1 \leqslant i \leqslant n: \ln i)$$

Вновь аппроксимируя эту сумму дискретных членов интегралом

$$\text{Integral } (1 : n) \ln x \, dx = x * (\ln x - 1) = n * \ln (n) - n + 1$$

получаем, наконец, приблизительное значение

$$M_{avg} \doteq n * (\ln (n) + g)$$

Отсюда можно сделать заключение, что, как правило, алгоритм с прямым выбором предпочтительнее строгого включения. Однако, если ключи в начале упорядочены или почти упорядочены, прямое включение будет оставаться несколько более быстрым.

### 2.2.3. Сортировка с помощью прямого обмена

Классификация методов сортировки редко бывает осмысленной. Оба разбиравшихся до этого метода можно тоже рассматривать как «обменные» сортировки. В данном же, однако, разделе мы опишем метод, где обмен местами двух элементов представляет собой характернейшую особенность процесса. Изложенный ниже алгоритм прямого обмена основывается на сравнении и смене мест для пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы.

Как и в упоминавшемся методе прямого выбора, мы повторяем проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива. Если мы будем рас-

Таблица 3.1. Три возможных обхода конем

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 23 | 10 | 15 | 4 | 25 | 23 | 4 | 9 | 14 | 25 |
| 16 | 5 | 24 | 9 | 14 | 10 | 15 | 24 | 1 | 8 |
| 11 | 22 | 1 | 18 | 3 | 5 | 22 | 3 | 18 | 13 |
| 6 | 17 | 20 | 13 | 8 | 16 | 11 | 20 | 7 | 2 |
| 21 | 12 | 7 | 2 | 19 | 21 | 6 | 17 | 12 | 19 |

| | | | | | |
|---|---|---|---|---|---|
| 1 | 16 | 7 | 26 | 11 | 14 |
| 34 | 25 | 12 | 15 | 6 | 27 |
| 17 | 2 | 33 | 8 | 13 | 10 |
| 32 | 35 | 24 | 21 | 28 | 5 |
| 23 | 18 | 3 | 30 | 9 | 20 |
| 36 | 31 | 22 | 19 | 4 | 29 |

схему — (3.29). К ней надо обращаться с помощью оператора Try(1).

```
PROCEDURE Try(i: INTEGER);
  VAR k: INTEGER;
BEGIN k := 0;
  REPEAT k := k+1; выбор k-го кандидата;
    IF подходит THEN                                    (3.29)
      его запись;
      IF i < n THEN Try(i+1);
        IF неудача THEN стирание записи END
      END
    END
  UNTIL удача OR (k = m)
END Try
```

В оставшейся части этой главы мы разберем еще три примера. В них используются различные реализации абстрактной схемы (3.29), и мы хотим еще раз продемонстрировать уместное использование рекурсии.

## 3.5. ЗАДАЧА О ВОСЬМИ ФЕРЗЯХ

Задача о восьми ферзях — хорошо известный пример использования методов проб и ошибок и алгоритмов с возвратами. В 1850 г. эту задачу исследовал К. Ф. Гаусс, однако полностью он ее так и не решил. Это никого не должно удивлять. Для подобных задач характерно отсутствие аналитического решения. Они требуют огромного количества изнурительной

работы, терпения и аккуратности. Поэтому такие задачи стали почти исключительно прерогативой электронных вычислительных машин, ведь им эти свойства присущи в значительно большей степени, чем человеку, пусть и гениальному.

Задача о восьми ферзях формулируется следующим образом (см. также [3.4]): восемь ферзей нужно расставить на шахматной доске так, чтобы один ферзь не угрожал другому. Воспользовавшись схемой 3.29 как шаблоном, легко получаем грубый вариант решения:

```
PROCEDURE Try(i: INTEGER);
BEGIN
    инициация выбора положения i-го ферзя;
  REPEAT выбор очередного положения;
    IF безопасное THEN поставить ферзя;
      IF i < 8 THEN Try(i+1);
        IF неудача THEN убрать ферзя END
      END
    END
  UNTIL удача OR мест больше нет
END Try
```

Чтобы идти дальше, нужно остановиться на каком-либо представлении для данных. Поскольку из шахматных правил мы знаем, что ферзь бьет все фигуры, находящиеся на той же самой вертикали, горизонтали или диагонали, то заключаем, что на каждой вертикали может находиться один и только один ферзь, поэтому при поиске места для i-го ферзя можно ограничить себя лишь i-й вертикалью. Таким образом, параметр i становится индексом вертикали, а процесс выбора возможного местоположения ограничивается восемью допустимыми значениями для индекса горизонтали j.

Остается решить вопрос: как представлять на доске эти восемь ферзей? Очевидно, доску вновь можно было бы представить в виде квадратной матрицы, но после небольших размышлений мы обнаруживаем, что это значительно усложнило бы проверку безопасности поля. Конечно, подобное решение нежелательно, поскольку такая операция выполняется очень часто. Поэтому хотелось бы остановиться на таком пред-

ставлении данных, которое, насколько это возможно, упростило бы проверку. В этой ситуации лучше всего делать непосредственно доступной именно ту информацию, которая действительно важна и чаще всего используется. В нашем случае это не поля, занятые ферзями, а сведения о том, находится ли уже ферзь на данной горизонтали или диагонали. (Мы уже знаем, что на каждой k-й вертикали ($1 \leqslant k \leqslant i$) стоит ровно один ферзь.) Эти соображения приводят к таким описаниям переменных:

```
VAR x: ARRAY [1 .. 8] OF INTEGER;
    a: ARRAY [1 .. 8] OF BOOLEAN;
    b: ARRAY [b1 .. b2] OF BOOLEAN;
    c: ARRAY [c1 .. c2] OF BOOLEAN;
```
(3.31)

где

$x_i$ обозначает местоположение ферзя на i-й вертикали;

$a_j$ указывает, что на j-й горизонтали ферзя нет;

$b_k$ указывает, что на k-й /-диагонали ферзя нет;

$c_k$ указывает, что на k-й \-диагонали ферзя нет.

Выбор границ индексов b1, b2, c1, c2 определяется, исходя из способа вычисления индексов для b и c, на /-диагонали у всех полей постоянна сумма координат i и j, а на \-диагонали постоянна их разность. Соответствующие вычисления приведены в прогр. 3.4. Если мы уже определили так данные, то оператор «Поставить ферзя» превращается в такие операторы:

```
x[i] := j, a[j] := FALSE, b[i + j] := FALSE;
c[i - j] := FALSE
```
(3.32)

а оператор «Убрать ферзя» в такие:

```
a[j] := TRUE; b[i + j] := TRUE; c[i - j] := TRUE
```
(3.33)

Условие «безопасно» выполняется, если поле с координатами $\langle i, j \rangle$ лежит на горизонтали и вертикали, которые еще не заняты. Следовательно, ему соответствует логическое выражение

```
a[j] & b[i + j] & c[i - j]
```
(3.34)

```
MODULE Queens;
  FROM InOut IMPORT WriteInt, WriteLn;
VAR i: INTEGER; q: BOOLEAN;
  a: ARRAY [1 .. 8] OF BOOLEAN;
  b: ARRAY [2 .. 16] OF BOOLEAN;
  c: ARRAY [-7 .. 7] OF BOOLEAN;
  x: ARRAY [1 .. 8] OF INTEGER;

PROCEDURE Try(i: INTEGER; VAR q: BOOLEAN);
  VAR j: INTEGER;
BEGIN j := 0;
  REPEAT j := j+1; q := FALSE;
    IF a[j] & b[i+j] & c[i-j] THEN
      x[i] := j;
      a[j] := FALSE; b[i+j] := FALSE; c[i-j] := FALSE;
      IF i < 8 THEN
        Try(i+1, q);
        IF ~q THEN
          a[j] := TRUE; b[i+j] := TRUE; c[i-j] := TRUE
        END
      ELSE q := TRUE
      END
    END
  UNTIL q OR (j = 8)
END Try;

BEGIN
  FOR i := 1 TO 8 DO a[i] := TRUE END;
  FOR i := 2 TO 16 DO b[i] := TRUE END;
  FOR i := -7 TO 7 DO c[i] := TRUE END;
  Try(1,q);
  FOR i := 1 TO 8 DO WriteInt(x[i], 4) END;
  WriteLn
END Queens.
```

Прогр. 3.4. Расстановка восьми ферзей.

На этом создание алгоритма заканчивается; полностью он представлен в прогр. 3.4. На рис. 3.9 приведено полученное решение $x = (1, 5, 8, 6, 3, 7, 2, 4)$.

Прежде чем закончить разбор задач, «посвященных» шахматной доске, мы воспользуемся задачей о восьми ферзях и представим одно важное обобщение алгоритма проб и ошибок. В общих словах, речь идет о нахождении не одного, а *всех* решений поставленной задачи.

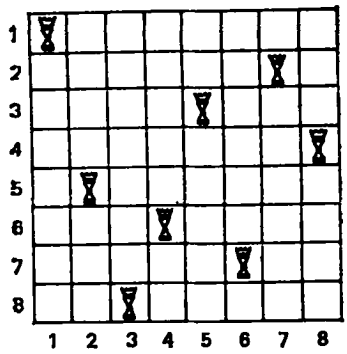


Рис. 3.9. Одно из решений задачи о восьми ферзях.

Такое обобщение получается довольно легко. Напомним, что формирование возможных кандидатов происходит регулярным образом, гарантирующим, что ни один кандидат не встретится более чем один раз. Такое свойство алгоритма обеспечивается тем, что поиск идет по дереву кандидатов так, что каждая из его вершин проходится точно один раз. Это позволяет, если найдено и должным образом зафиксировано одно решение, просто переходить к следующему кандидату, предлагаемому упомянутым процессом систематического перебора. Общую схему такого процесса (3.35) можно «вывести» из схемы (3.29).

```
PROCEDURE Try(i: INTEGER);
  VAR k: INTEGER;
BEGIN                                                          (3.35)
  FOR k := 1 TO m DO
    выбор k-го кандидата;
    IF подходит THEN его запись;
      IF i < n THEN Try(i+1) ELSE печатать решения END;
      стирание записи
    END
  END
END Try
```

Обратите внимание: из-за того, что условие окончания в процессе выбора свелось к одному отношению k = m, оператор повторения со словом REPEAT заменится на оператор цикла с FOR. Удивительно, что

```
MODULE AllQueens;
  FROM InOut IMPORT WriteInt, WriteLn;

VAR i: INTEGER;
   a: ARRAY [ 1 .. 8] OF BOOLEAN;
   b: ARRAY [ 2 .. 16] OF BOOLEAN;
   c: ARRAY [-7 .. 7] OF BOOLEAN;
   x: ARRAY [ 1 .. 8] OF INTEGER;

 PROCEDURE print;
   VAR k: INTEGER;
 BEGIN
   FOR k := 1 TO 8 DO WriteInt(x[k], 4) END ;
   WriteLn
 END print;

 PROCEDURE Try(i: INTEGER);
   VAR j: INTEGER;
 BEGIN
   FOR j := 1 TO 8 DO
    IF a[j] & b[i+j] & c[i-j] THEN
      x[i] := j;
      a[j] := FALSE; b[i+j] := FALSE; c[i-j] := FALSE;
      IF i < 8 THEN Try(i+1) ELSE print END ;
      a[j] := TRUE; b[i+j] := TRUE; c[i-j] := TRUE
    END
   END
 END Try;

BEGIN
  FOR i := 1 TO 8 DO a[i] := TRUE END ;
  FOR i := 2 TO 16 DO b[i] := TRUE END ;
  FOR i := -7 TO 7 DO c[i] := TRUE END ;
  Try(1)
END AllQueens.
```

Прогр. 3.5. Расстановка восьми ферзей (все решения).

поиск всех возможных решений выполняется более простой программой, чем в случае поиска одного-единственного решения *).

---

*) Если бы программа отыскивала лишь 12 принципиально различных решений, причем делала бы это максимально быстро, то и она сама, и данные, которыми она должна была бы манипулировать, оказались бы значительно более сложными. — *Прим. перев.*

Таблица 3.2. Двенадцать решений задачи восьми ферзей

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | n |
|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 8 | 6 | 3 | 7 | 2 | 4 | 876 |
| 1 | 6 | 8 | 3 | 7 | 4 | 2 | 5 | 264 |
| 1 | 7 | 4 | 6 | 8 | 2 | 5 | 3 | 200 |
| 1 | 7 | 5 | 8 | 2 | 4 | 6 | 3 | 136 |
| 2 | 4 | 6 | 8 | 3 | 1 | 7 | 5 | 504 |
| 2 | 5 | 7 | 1 | 3 | 8 | 6 | 4 | 400 |
| 2 | 5 | 7 | 4 | 1 | 8 | 6 | 3 | 072 |
| 2 | 6 | 1 | 7 | 4 | 8 | 3 | 5 | 280 |
| 2 | 6 | 8 | 3 | 1 | 4 | 7 | 5 | 240 |
| 2 | 7 | 3 | 6 | 8 | 5 | 1 | 4 | 264 |
| 2 | 7 | 5 | 8 | 1 | 4 | 6 | 3 | 160 |
| 2 | 8 | 6 | 1 | 3 | 5 | 7 | 4 | 336 |

Обобщенный алгоритм, приведенный в прогр. 3.5, отыскивает 82 решения задачи о восьми ферзях. Однако принципиально различных решений всего 12 — наша программа не учитывает симметрию. В табл. 3.2 приведены первые 12 решений. Приведенное справа число n указывает, сколько раз проводилась проверка на «безопасность» поля. Среднее число для всех 92 решений равно 161.

## 3.6. ЗАДАЧА О СТАБИЛЬНЫХ БРАКАХ

Предположим, есть два непересекающихся множества A и B одинакового размера n. Нужно найти множество из n пар $\langle a, b\rangle$, таких, что a принадлежит A, а b принадлежит B, и они удовлетворяют некоторым условиям. Для выбора таких пар существует много различных критериев; один из них называется «*правилом стабильных браков*».

Предположим, что A — множество мужчин, а B — женщин. У каждых мужчины и женщины есть различные правила предпочтения возможного партнера. Если среди n выбранных пар существуют мужчины и женщины, не состоящие между собой в браке, но предпочитающие друг друга, а не своих фактических супругов, то такое множество браков считается нестабильным. Если же таких пар нет, то множество считается стабильным. Такая ситуация характерна для многих похожих задач, где нужно проводить распределение в соответствии с некоторыми правилами