Чулюков В.А.

МЕТОДЫ РАЗРАБОТКИ ПРОГРАММ (АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ)

ЧАСТЬ 2

РЕКУРСИЯ

Воронеж - 2015

1. ВВЕДЕНИЕ

Есть тема, которую часто опускают во вводных курсах программирования, хотя она и играет важную концептуальную роль во многих алгоритмах, – это рекурсия. Поэтому эта лекция посвящена рекурсивным решениям. В ней показано, что рекурсия – обобщение понятия повторения (итерации), и как таковая она представляет собой важную и мощную концепцию программирования. К несчастью, во многих курсах программирования рекурсия используется в примерах, где достаточно простой итерации. Вместо этого хотелось бы обратить внимание на особенно интересный раздел программирования – это задачи из области «искусственного интеллекта». Здесь нужно строить алгоритмы, которые находят решение определенной задачи не по фиксированным правилам вычисления, а методом проб и ошибок (алгоритмы с возвратом).

Выясним, что такое рекурсия? Во всех языках программирования можно описать функцию, как подпрограмму с параметром, и вызывать ее при необходимости. Однако не во всех языках программирования функция может вызывать сама себя (например, в Фортране). Если же функция в процессе выполнения вызывает сама себя, то такой прием называется рекурсией. Итак, **рекурсия** — это такой способ организации вспомогательного алгоритма (подпрограммы), при котором эта подпрограмма (процедура или функция) в ходе выполнения ее операторов обращается сама к себе. Вообще, **рекурсивным** называется любой объект, который частично определяется через себя.

Тут нужно задуматься – что значит, функция вызывает сама себя? Обычно конструкции языков программирования можно более или менее представить и проиллюстрировать примерами из реальной жизни. А как же представить наглядно функцию, вызывающую самою себя? Лучшей моделью, на мой взгляд, служит английский стишок «Дом, который построил Джек», а еще лучшей – пародия на него, стишок одной команды КВН из книжки «КВН раскрывает секреты» вышедшей в 1967 году (М. Молодая гвардия).

Вот стенд, который построил студент,
А вот космическая частица,
Которая с бешеной скоростью мчится
В стенде, который построил студент.
А вот инженер молодой, бледнолицый,
Который клянет и судьбу, и частицу,

Которая с бешеной скоростью мчится
В стенде, который построил студент.
А вот кандидат, горделивый не в меру,
Который блистательно сделал карьеру,
Совместно работая с тем инженером,
Который клянет и судьбу и частицу,
Которая с бешеной скоростью мчится
В стенде, который построил студент.
А вот начальник, на вид простоватый,
Который был шефом того кандидата,
Который блистательно сделал карьеру,
Совместно работая с тем инженером,
Который клянет и судьбу и частицу,
Которая с бешеной скоростью мчится
В стенде, который построил студент.
А вот консультант от академии,
С которым встречался время от времени
Тот самый начальник, на вид простоватый,
Который был шефом того кандидата,
Который блистательно сделал карьеру,
Совместно работая с тем инженером,
Который клянет и судьбу и частицу,
Которая с бешеной скоростью мчится
В стенде, который построил студент.
А вот отчетов горы бумажные,
В которых копалась комиссия важная,
Которая выдала крупную премию,
Тому консультанту из академии,
Начальнику дали за вид простоватый,
Кусок уделили тому кандидату,
Который блистательно сделал карьеру,
Остатки вручили тому инженеру,
Который уже не ругает частицу,
Которая с бешеной скоростью мчится
В стенде, который построил
СТУДЕНТ

А кто не видел рекламной картинки, которая содержит свое собственное изображение?

Рекурсия является особенно мощным средством в математических определениях. Известны примеры рекурсивных определений натуральных чисел, некоторых функций:

Натуральные числа:

a) 1 есть натуральное число;

b) целое число, следующее за натуральным, есть натуральное число.

Функция факториал n! для неотрицательных целых чисел:

a) 0! = 1,

b) если n > 0, то n! = n* (n - 1)!

Вот еще одно рекурсивное определение.

a) 3 коровы – это стадо коров.

b) Стадо из *n* коров – это стадо из *n*–1 коровы и еще одна корова.

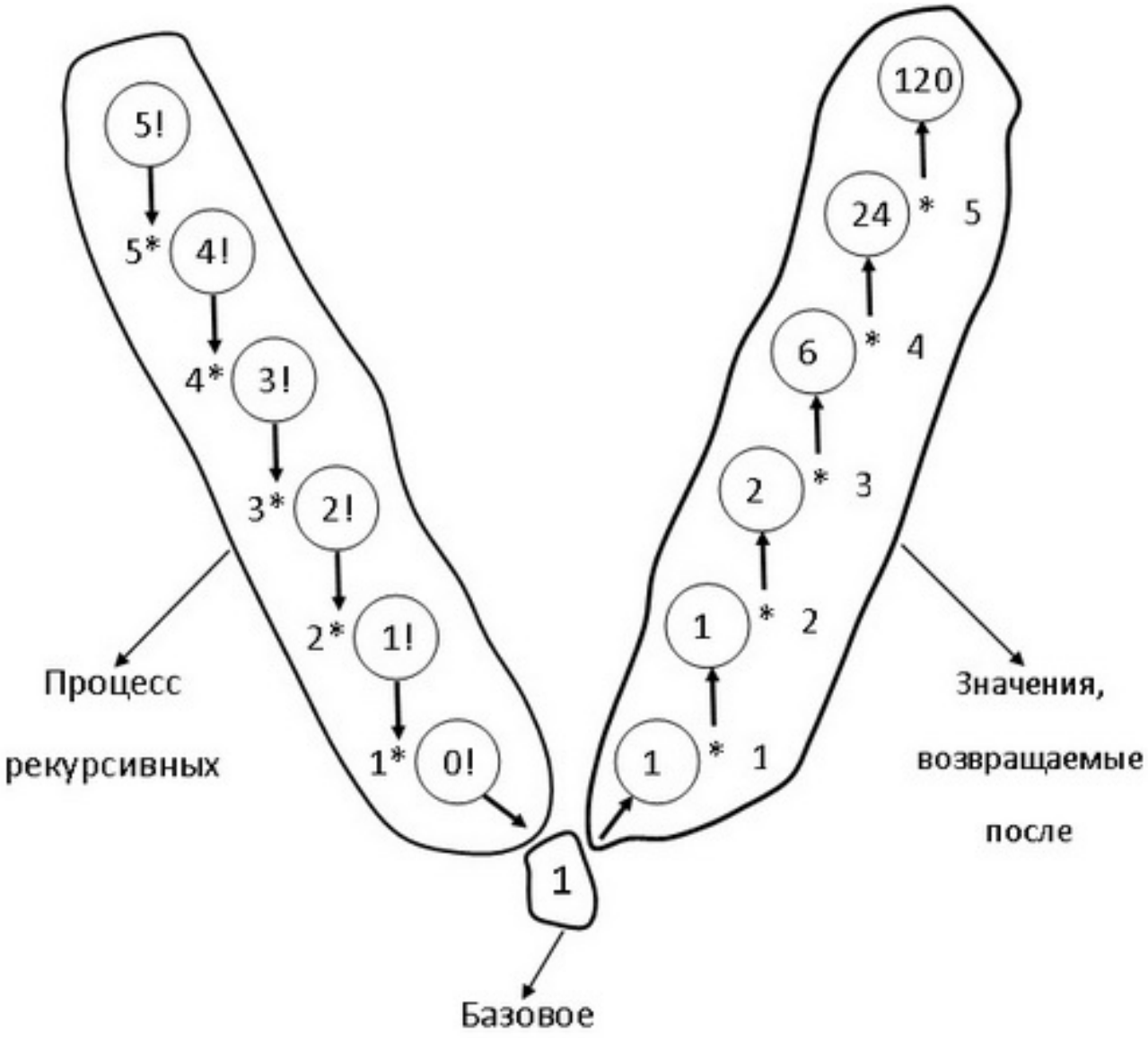
Попробуем применить это определение для проверки, является ли стадом группа из пяти коров (обозначим ее К5). Объект К5 не удовлетворяет первому пункту определения, поскольку пять коров – это не три коровы. Согласно второму пункту К5 – стадо, если там есть одна корова, а остальная часть К5, назовем ее К4, – тоже стадо коров. Решение относительно объекта К5 откладывается, пока не будет принято решение относительно К4. Объект К4 снова не подходит под первый пункт, а второй пункт гласит, что К4 – стадо, если объект К3, полученный из К4 путем отделения одной коровы, тоже стадо. Решение о К4 тоже откладывается. Наконец, объект К3 удовлетворяет первому пункту определения, и мы можем смело утверждать, что К3– стадо коров. Теперь и о К4 можно утверждать, что это стадо, а значит, и К5 является стадом коров.

Любое рекурсивное определение состоит из двух частей. Эти части принято называть *базовой* и *рекурсивной* частями. Базовая часть является нерекурсивной и задает определение для некоторой фиксированной части объектов. Рекурсивная часть определяет понятие через него же и записывается так, чтобы при цепочке повторных применений она редуцировалась бы к базе.

Очевидно, что мощность рекурсии связана с тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания. Точно так же бесконечные вычисления можно описать с помощью конечной рекурсивной программы, даже если эта программа не содержит явных циклов. Однако лучше всего использовать рекурсивные алгоритмы в тех случаях, когда решаемая задача, или вычисляемая функция, или обрабатываемая структура данных определены с помощью рекурсии. В общем виде рекурсивную программу P можно изобразить как композицию R базовых операторов S_i (не содержащих P) и самой P *:*

$$P \equiv R\ [S_i, P]. \qquad (1)$$

Необходимое и достаточное средство для рекурсивного представления программ – это описание процедур, или подпрограмм, так как оно позволяет присваивать какому-либо оператору имя, с помощью которого можно вызвать этот оператор. Если процедура P содержит явное обращение к самой себе, то она называется *прямо рекурсивной;* если P содержит обращение к процедуре Q, которая содержит (прямо или косвенно) обращение к P, то P называется *косвенно рекурсивной*. Поэтому использование рекурсии не всегда сразу видно из текста программы.



С процедурой возврата принято связывать некоторое множество локальных объектов, т.е. переменных, констант, типов и процедур, которые определены локально в этой процедуре, а вне ее не существуют или не имеют смысла. Каждый раз, когда такая процедура рекурсивно вызывается, для нее создается новое множество локальных переменных. Хотя они имеют те же имена, что и соответствующие элементы множества локальных переменных, созданного при предыдущем обращении к этой же процедуре, их значения различны. Следующее правило области действия идентификаторов позволяют исключить какой-либо конфликт при использовании имен: идентификаторы всегда ссылаются на множество переменных, созданное последним. То же правило относится к параметрам процедуры.

Подобно операторам цикла, рекурсивные процедуры могут привести к бесконечным вычислениям. Поэтому необходимо рассмотреть проблему

окончания работы процедур (определить граничное условие). Очевидно, что для того, чтобы работа когда-либо завершилась, необходимо, чтобы рекурсивное обращение к процедуре P подчинялось условию B, которое в какой-то момент перестает выполняться. Поэтому более точно схему рекурсивных алгоритмов можно представить так:

$$P \equiv \textbf{if } B \textbf{ then } R\ [S_i,\ P], \qquad (2)$$

или

$$P \equiv R\ [S_i, \textbf{ if } B \textbf{ then } P] \qquad (3)$$

Основной способ доказать, что выполнение операторов цикла когда-либо заканчивается, – определить функцию $f(x)$ (x – множество переменных программы), такую, что $f(x) \leq 0$ удовлетворяет условию окончания цикла (с предусловием или с постусловием), и доказать, что при каждом повторении $f(x)$ уменьшается. Точно так же можно доказать, что выполнение рекурсивной процедуры P когда-либо завершиться, показав, что каждое выполнение P уменьшает $f(x)$. Наиболее надежный способ обеспечить окончание процедуры – связать с P параметр (значение), скажем n, и рекурсивно вызвать P со значением этого параметра $n - 1$. Тогда замена условия B на $n > 0$ гарантирует окончание работы. Это можно изобразить следующими схемами программ:

$$P(n) \equiv \textbf{if } n > 0 \textbf{ then } R\,[S_i,\ P(n-1)], \qquad (4)$$

$$P(n) \equiv R\,[S_i,\ \textbf{if } n > 0 \textbf{ then } P\,(n-1)] \qquad (5)$$

Итак, обращение к рекурсивной подпрограмме ничем не отличается от вызова любой другой подпрограммы. При этом при каждом новом рекурсивном обращении в памяти создаётся новая копия подпрограммы со всеми локальными переменными. Такие копии будут порождаться до выхода на граничное условие. Очевидно, в случае отсутствия граничного условия, неограниченный рост числа таких копий приведёт к аварийному завершению программы за счёт переполнения стека.

Порождение все новых копий рекурсивной подпрограммы до выхода на граничное условие называется *рекурсивным спуском*. Максимальное количество копий рекурсивной подпрограммы, которое одновременно может находиться в памяти компьютера, называется *глубиной рекурсии*. Завершение работы рекурсивных подпрограмм, вплоть до самой первой, инициировавшей рекурсивные вызовы, называется *рекурсивным подъёмом*.

Выполнение действий в рекурсивной подпрограмме может быть организовано одним из вариантов:

Begin
P;
Операторы
End;
	Begin
	Операторы;
	P
	End;
		Begin
		Операторы;
		P;
		Операторы
		End;

Рекурсивный подъем	Рекурсивный спуск	И рекурсивный спуск, и рекурсивный подъем

Здесь P — рекурсивная подпрограмма. Как видно, действия могут выполняться либо на одном из этапов рекурсивного обращения, либо на обоих сразу.

1.1 Когда не следует использовать рекурсию

Рекурсивные алгоритмы наиболее пригодны в случаях, когда представленная задача или используемые данные определены рекурсивно. Но это не значит, что при наличии таких рекурсивных определений лучшим способом решения задачи непременно является рекурсивный алгоритм. В действительности из-за того, что обычно понятие рекурсивных алгоритмов объяснялось

на неподходящих примерах, в основном и возникло широко распространенное предубеждение против использования рекурсии в программировании и приравнивание ее к неэффективности. Повлиял на это и тот факт, что широко распространенный язык программирования Фортран запрещает рекурсивное использование программ и тем самым не допускает рекурсию, даже когда ее применение оправдано.

Программы, в которых следует избегать использования рекурсии, можно охарактеризовать следующей схемой, изображающей их строение. Эта схема (6) и эквивалентная ей (7):

$$P \equiv \textbf{if } B \textbf{ then } (S;P) \qquad (6)$$

$$P \equiv (S; \textbf{ if } B \textbf{ then } P) \qquad (7)$$

Эти схемы естественно применять в тех случаях, когда выполняемые значения определяются с помощью простых рекуррентных соотношений. Рассмотрим, например, широко известный пример вычислений факториалов $f_i = i!$:

$$\begin{aligned} i &= 0, \ 1, \ 2, \ 3, \ 4, \ 5, \ \ldots \\ f_i &= 1, \ 1, \ 2, \ 6, \ 24, \ 120, \ \ldots \end{aligned} \qquad (8)$$

«Нулевое» число определяется явным образом как $f_0 = 1$, а последующие числа обычно определяются рекурсивно - с помощью предшествующего значения:

$$f_{i+1} = (i+1) \cdot f_i \qquad (9)$$

Эта формула предполагает использование рекурсивного алгоритма для вычисления n-го факториального числа. Если мы введем две переменные I и F для значений i и f_i на i-м уровне рекурсии, то увидим, что для перехода к следующему числу в последовательности (8) необходимы следующие вычисления:

$$I := I + 1; \ \ F := I * F \qquad (10)$$

и, подставив (10) вместо S в (6), мы получаем рекурсивную программу

$$P \equiv if\ I < n\ then\ (I := I+1;\ F := I * F;P)$$

$$I := 0;\ F := 1;\ P \qquad (11)$$

Первую строку в (11) можно записать на языке программирования Turbo Pascal следующим образом:

```
procedure P;
begin if I < n then
        begin I := I+1; F:= I*F; P                    (12)
        end
end
```

Чаще употребляемая, но эквивалентная, по существу, форма дана в (13). Вместо процедуры здесь вводится функция. Функцию можно использовать непосредственно как элемент выражения. Тем самым переменная *F* становится излишней, а роль *I* выполняет явный параметр функции.

```
function F( I : integer):integer;
begin if I > 0 then F := I*F(I - 1)
                 else F := 1;                         (13)
end;
```

Полностью текст программы будет иметь следующий вид:

```
program factor;
var
I: integer;
function F(I: integer):integer;
          begin
                    if I=0 then F:=1
                    else F:=I*(F(I-1));
end;
Begin{of main}
writeln('Введите число ');
readln(I);
writeln(I,'! = ',F(I));
readln;
end.
```

Программа 1.

Результат работы программы 1:

```
Введите число
7
7!=5040
```

Совершенно ясно, что здесь рекурсию можно заменить обычной итерацией, а именно:

```
I := 0: F:= 1;
while I < n do
    begin
        I := I + 1;                    (14)
        F := I * F
    end
```

В общем виде программы, соответствующие схемам (6) или (7), нужно преобразовать так, чтобы они соответствовали схеме (15):

$$P \equiv (x{:=}x_0;\ \textbf{while}\ B\ \textbf{do}\ S) \qquad (15)$$

Есть и другие, более сложные рекурсивные схемы, которые можно и должно переводить в итеративную форму. Примером служит вычисление чисел Фибоначчи, определяемых с помощью рекуррентного соотношения

$$fib_{n+1} = fib_n + fib_{n-1} \quad \text{для} \quad n > 0$$

$$\text{и } fib_1 = 1,\ fib_0 = 0. \qquad (16)$$

Проще говоря, каждое число Фибоначчи является суммой двух предыдущих. Последовательность имеет вид

1, 1, 2, 3, 5, 8, 13, 21, 34, 55…

Числа Фибоначчи играют важную роль в технике и даже в биологии (в соответствии с законом Фибоначчи растет численность живых организмов).

При непосредственном, «лобовом» подходе мы получим программу

```
function Fib(n: integer): integer;
begin if n = 0 then Fib := 0  else
      if n = 1 then Fib := 1  else                    (17)
      Fib := Fib(n-1) + Fib(n-2)
End
```

Напишем текст программы 2, выводящей на экран номер и значение числа Фибоначчи, вычисляемого в функции Fib. Причем, сама функция Fib,

кроме оговоренных первого и второго чисел, обращается сама к себе, чтобы сложить два предыдущих значения ряда.

```
program fibon;
var     i,n,fact: integer;
function Fib(n:integer):integer;
        begin
                if n=0 then Fib:=0 else
                if n=1 then Fib:=1 else Fib:=Fib(n-1)+Fib(n-2);
        end;
Begin{of main}
for i:=1 to 10 do writeln(i,' ',Fib (I),' ',i+10,' ',Fib(i+10));   readln;
end.
```

Программа 2

Результат работы программы 2:

```
1     1     11    89
2     1     12    144
3     2     13    233
4     3     14    377
5     5     15    610
6     8     16    987
7     13    17    1597
8     21    18    2584
9     34    19    4181
10    55    20    6765
```

Проследим, например, как работает рекурсивная функция при вычислении пятого члена ряда Фибоначчи. Для пятого члена значение функции должно равняться сумме четвертого и третьего членов. При вычислении четвертого члена функция обратиться к третьему и второму, при вычислении третьего – ко второму и первому, которые равны единице. Подобное «погружение» произойдет и при вычислении третьего члена как слагаемого четвертого (рис.1).

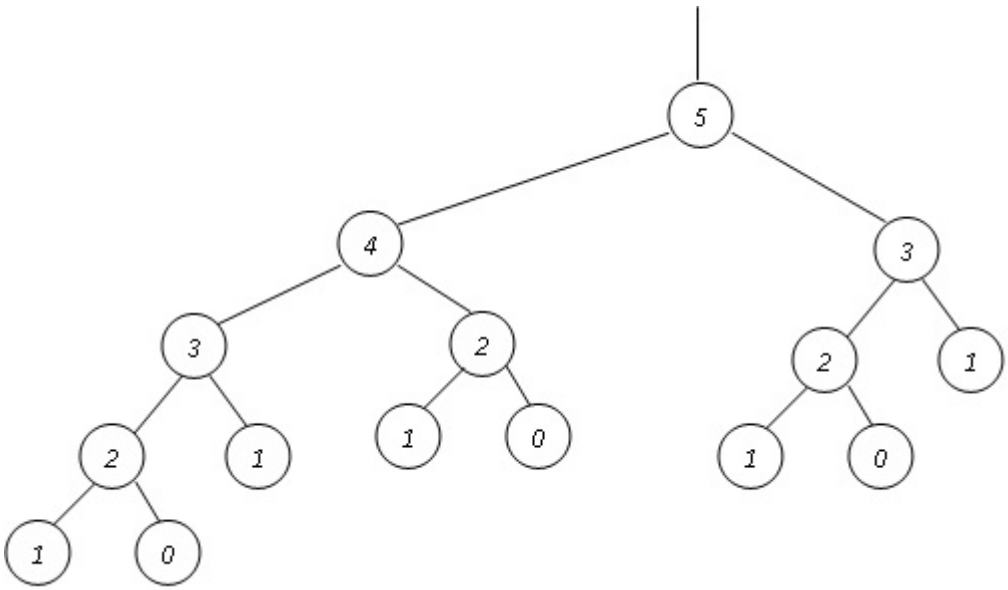


Рис.1. 15 вызовов *Fib*(*n*) при *n*=5

То есть мы видим, что в данном случае применение рекурсии на редкость неэффективно, мы заставляем процессор совершать множество «движений», причем их количество лавинообразно (экспоненциально) растет с ростом номера числа в ряду. Если первые десять чисел появляются на экране моментально, следующие с видимой и растущей задержкой, двадцатое на Pentium'е III (700МГц, 64МБ ОЗУ) «пережевывалось» около 5 секунд. (Сороковое – около минуты.)

Однако очевидно, что числа Фибоначчи можно вычислять по итеративной схеме, при которой использование вспомогательных переменных $x = \mathrm{fib}_i$ и $y = \mathrm{fib}_{i-1}$ позволяет избежать повторного вычисления одних и тех же значений:

```
{вычисление x = fib_n для n > 0}
i := 1; x := 1; y := 0;
while i < n do
    begin z := x; i := i + 1;                    (18)
        x := x + y; y := z;
    end
```

Заметим, что три присваивания *x*, *y* и *z* можно выразить всего лишь двумя присваиваниями без использования вспомогательных переменных *z*

$$x := x + y;\ y := x - y$$

Итак, вывод таков: следует избегать рекурсии, когда имеется *очевидное* итеративное решение поставленной задачи. (*Итерация* - способ организации

обработки данных, при котором определенные действия повторяются многократно, не приводя при этом к рекурсивным вызовам программ).

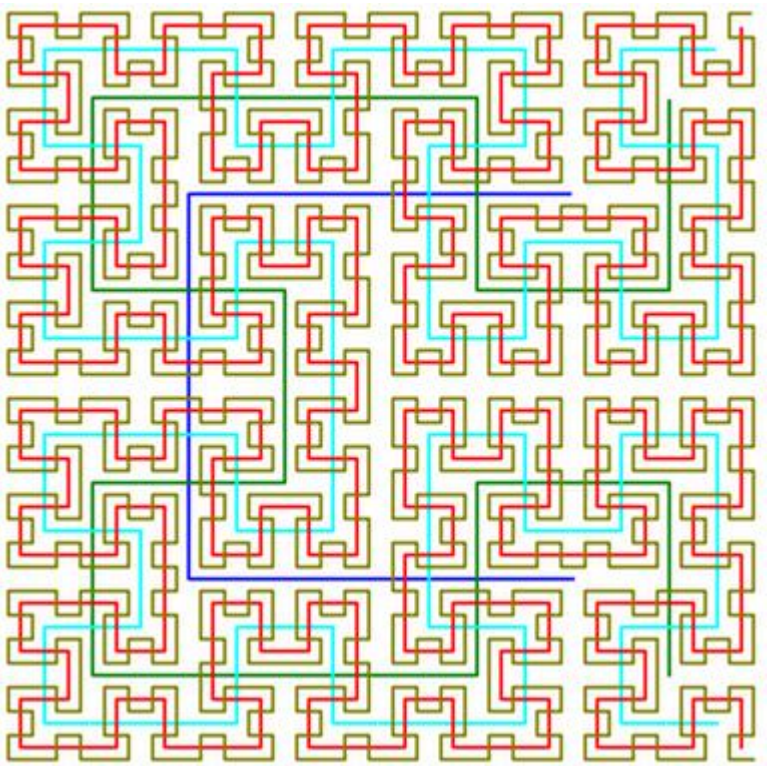
Но это не означает, что всегда нужно избавляться от рекурсии любой ценой. Во многих случаях она вполне применима. Тот факт, что рекурсивные процедуры можно реализовать на не рекурсивных по сути машинах, говорит о том, что для практических целей любую рекурсивную программу можно преобразовать в чисто итеративную.

Но это требует явного манипулирования со стеком рекурсий, и эти операции до такой степени заслоняют суть программы, что понять ее становится очень трудно. Следовательно, алгоритмы, которые по своей природе скорее рекурсивны, чем итеративны, нужно представлять в виде рекурсивных процедур.

2 Два примера рекурсивных программ

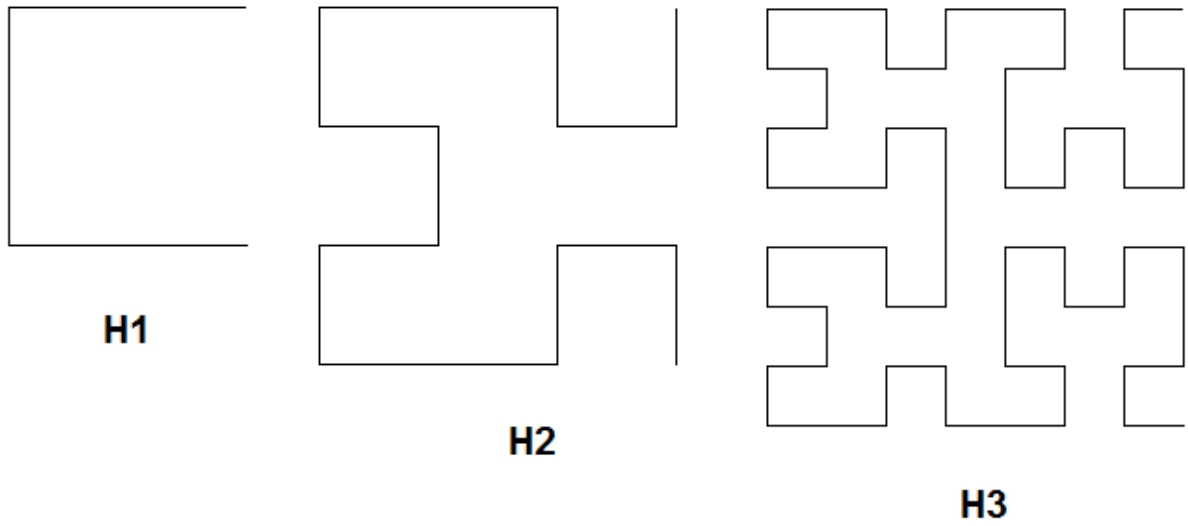
2.1 Кривые Гильберта

Симпатичный узор на следующем рисунке состоит из суперпозиции пяти кривых.



Эти кривые строятся на основе некоторого регулярного образца, и предполагается, что их можно нарисовать с помощью графопостроителя,

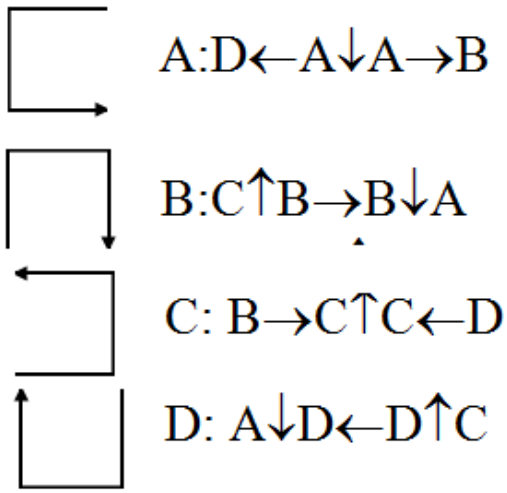
управляемого вычислительной машиной. Задача — найти рекурсивную схему, по которой можно написать программу, управляющую графопостроителем. Рассматривая рисунок, мы обнаруживаем, что три наложенные друг на друга кривые имеют форму, показанную на рис.



Мы обозначаем их через H_1, H_2 и H_3. На рисунках видно, что H_{i+1} получается соединением четырех Hi вдвое меньшего размера, соответствующим образом повернутых и связанных вместе тремя соединительными линиями. Отметим, что можно считать, что H_1 состоит из четырех пустых H_0, связанных тремя прямыми линиями. Кривая H_i называется кривой *Гильберта i-го порядка* в честь его первооткрывателя Д. Гильберта (1891).

Предположим, что у нас имеются следующие основные средства для построения графов: две координаты — переменные x и y, процедура *setplot* (устанавливающая перо в точку с координатами x и y) и процедура *plot* (передвигающая перо, которое при этом чертит прямую из текущей точки в точку, обозначенную x и y).

Поскольку каждая кривая H_i состоит из четырех вдвое меньших копий H_{i-1}, то естественно построить процедуру, рисующую H_i в виде композиции четырех частей, каждая из которых рисует H_{i-1} соответствующего размера и с нужным поворотом. Если мы обозначим эти четыре части A, B, C и D, а подпрограммы, рисующие соединительные линии, в виде стрелок, указывающих соответствующее направление, то получим следующую рекурсивную схему (19):



(19)

Если длину соединительной линии обозначить через h, то процедуру, соответствующую схеме A, можно легко выразить с помощью рекурсивных обращений к описанным аналогичным образом процедурам B и D и самой процедуры A:

```
Procedure A(i: integer);
begin if i>0 then
        begin D(i-1); x:=x - h; plot;
              A(i-l); y:=y - h;  plot;
              A(i-l); x:=x + h;  plot;
             B(i-l)
        end
end
```

Эта процедура инициируется один раз основной программой для каждой кривой Гильберта, которые накладываются одна на другую, образуя данный рисунок. Основная программа задает исходную точку для кривой, т. е. начальные значения x и y, и единичное приращение h. Величина h_0 соответствует ширине всей страницы и должна удовлетворять равенству $h_0 = 2^k$ для некоторого $k <= n$. Программа рисует всего n кривых Гильберта.

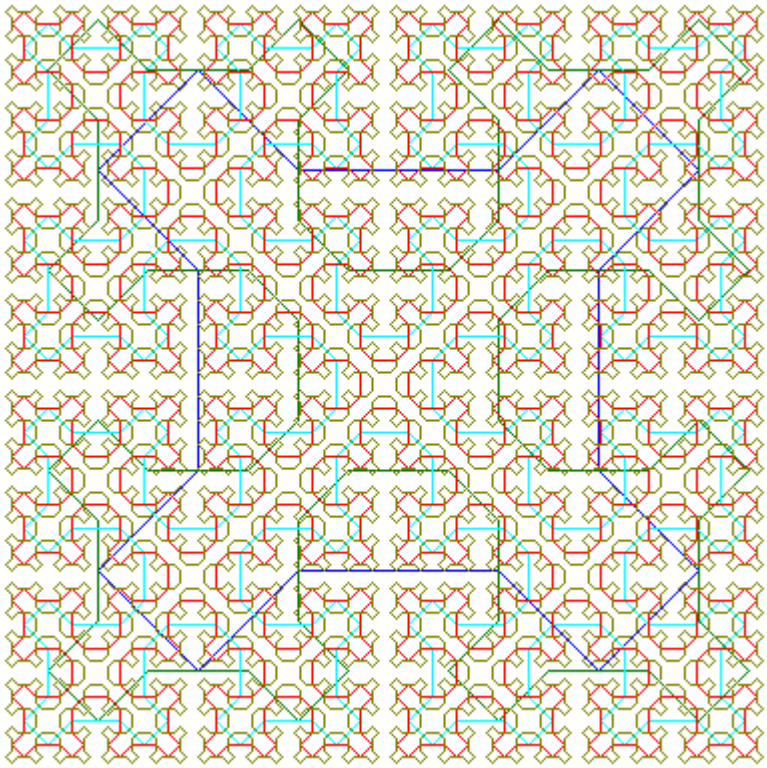
```
Program Hilbert(pf,output);
{изображение кривых Гильберта порядка от 1 до n}
Const n=4;h0=512;
Var i,h,x,y ,x0,y 0: integer;
```

```
Procedure A(i: integer);
Begin if  i >0 then
        begin D(i-l);x:=x-h; plot;
              A(i-l);y:=y-h ;  plot;
              A(i-1) ;x:=x+h; plot;
              B(i-l);
        end;
end;
Procedure B (i: integer);
begin if i>0 then
       Begin C(i-l)y:=y+h;  plot;
             B(i-l);x:=x+h;  plot;
             B(i-l);y:=y-h;  plot;
             A(i-l);
        end
end;
Procedure C(i: integer);
Begin if i>0 then
       begin B(i-l);x:=x+h;  plot;
             C(i-l);y:=y+h; plot;
             C(i-l);x:=x-h;  plot;
             D(i-l);
        end;
end;
Procedure D(i: integer);
Begin if i>0 then
```

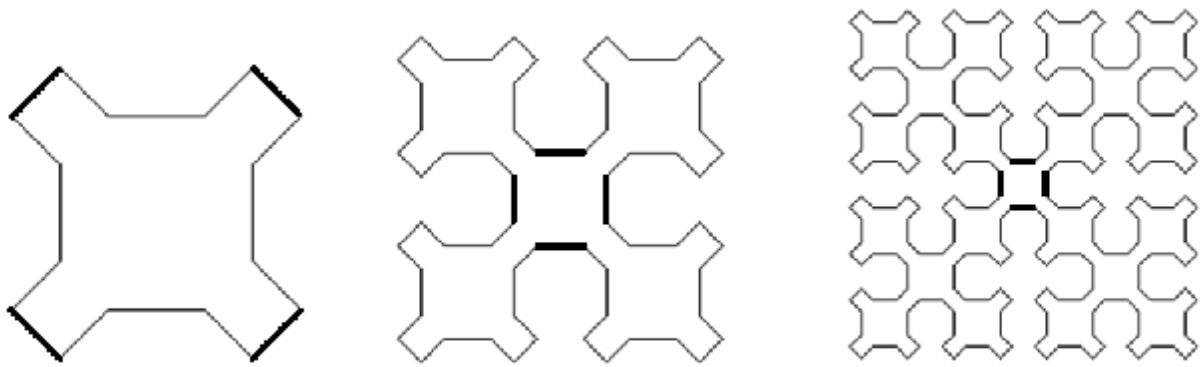
```
        begin A(i-l);y:=y-h; plot;
                D(i-l);x:=x-h; plot;
                D(i-l);y:=y+h; plot;
                C(i-l);
         end;
end;
begin
       i:=0;h:=h0;x0:=h div 2;y0:=x0;
        repeat {изображение кривой Гильберта порядка i}
                i:=i+l;  h:=h div 2;
                x0:=x0+(h div 2);y0:=y0+(h div 2);
                x:=x0;y:=y0;setplot;
               A(i)
        until i=n;
end.
```

2.2 Кривые Серпинского

Похожий, но несколько более сложный и эстетически утонченный рисунок приведен на на следующем рисунке.



Он также получен с помощью наложения нескольких кривых; три такие кривые показаны на следующем рисунке.



S_1 S_2 S_3

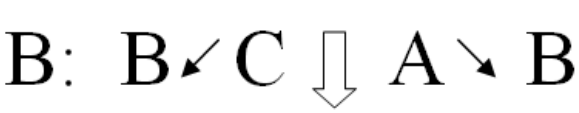
Кривая Si, называется кривой Серпинского i-го порядка. Какова рекурсивная схема для такой кривой? Попробуем в качестве основного строительного блока выделить лист S_1, возможно, без одного ребра. Но это не приводит нас к нужному решению. Принципиальное различие между кривыми Серпинского и Гильберта заключается в том, что кривые Серпинского являются замкнутыми (без соединительных линий). Это означает, что основная рекурсивная схема должна давать разомкнутую кривую, а четыре части соединяются линиями, не принадлежащими самому рекурсивному узору. Действительно, эти связи представляют собой четыре прямые в четырех внешних «углах» изображенных на рис. жирными линиями. Можно считать, что они принадлежат к непустой начальной кривой S_0, представляющей собой квадрат, стоящий на одном угле.

Теперь легко построить рекурсивную схему. Четыре составляющие фигуры вновь обозначаются A, B, C и D, а соединительные линии рисуются явно. Заметим, что четыре рекурсивные кривые действительно одинаковы с точностью до поворота на 90°.

Основной образ кривых Серпинского следующий:

(21)

а объединенные рекурсивные фигуры строятся по таким схемам:



(22)

(Двойные стрелки обозначают линии двойной длинны.)

Используя те же примитивы для операций построения, что и в случае кривых Гильберта, приведенную выше рекурсивную схему легко преобразовать в (прямо и косвенно) рекурсивный алгоритм:

```
procedure A(i: integer);
begin if i >0   then
        begin
                A(i-l); x:=x+h; y:=y-h; plot;
                B(i-l);x:=x+2*h;  plot;                                    (23)
                D(I-1);  x:=x+h; y:=y+h; plot;
                A(i-l)

        end
end;
```

Эта процедура соответствует первой строке рекурсивной схемы (22). Процедуры, соответствующие фигурам B, C и D, строятся аналогично. Основная программа строится по схеме (21). Она должна установить начальные значения для координат рисунка и задать длину единичной линии h в зависимости от формата бумаги, как показано в программе. Результат работы этой программы при n= 4 показан на рис. 7. Заметим, что S0 не рисуется.

```
program Serp;
{изображение кривых Серпинского порядка от 1 до n}
const n=4;h0=512;
var i,h,x,y,x0,y0:integer;
    pf: file of integer;
procedure A(i: integer);
begin if i >0 then
        begin A(i-l); x:=x+h; y:=y-h; plot;
                B(i-l); x:=x+2·h;  plot;
                D(i-l);x:=x+h; y:=y+h; plot;
                A(i-l)

         end
end;

procedure B(i: integer);
begin if i >0 then
        begin B(i-l); x:=x-h; y:=y-h; plot;
                C(i-l); x:=x-2*h;  plot;
                A(i-l);x:=x+h; y:=y-h; plot;
                B(i-l)

        end
end;
```

```
procedure C(i: integer);
begin if i >0 then
        begin c(i-l); x:=x-h; y:=y+h; plot;
                D(i-l); x:=x-2*h;  plot;
                B(i-l); x:=x-h; y:=y-h; plot;
                C(i-l)
        end
end;
procedure D (i: integer);
begin if i >0 then
        begin D(i-l); x:=x+h; y:=y+h; plot;
                A(i-l); x:=x+2*h;  plot;
                C(i-l); x:=x-h; y:=y+h; plot;
                D(i-l)
        end
end;
Begin
    I:=0; h:=h0 div 4; x0:=2*h; y0:=3*h;
     repeat i:=i+l; x0:=x0-h;
          h:=h div 2; y0:=y0+h;
          x:=x0 ; y:=y0; setplot;
          A(i); x:=x+h; y:=y-h; plot;
          B(i); x:=x-h; y:=y-h;  plot;
          C(i); x:=x-h; y:=y+h;  plot;
          D(i); x:=x+h; y:=y+h; plot;
    until i = n;
    end.
```

2. Искуственный интеллект

Особенно интересный раздел программирования — это задачи из области «искусственного интеллекта». Здесь нужно строить алгоритмы, которые находят решение определенной задачи не по фиксированным правилам вычисления, а методом проб и ошибок. Обычно процесс проб и ошибок разделяется на отдельные подзадачи. Часто эти подзадачи наиболее естественно описываются с помощью рекурсии. Процесс проб и ошибок можно рассматривать в общем виде как поисковый процесс, который постепенно строит и просматривает (а также обрезает) дерево подзадач. Во многих случаях такие деревья поиска растут очень быстро в зависимости от заданного параметра. Соответственно увеличивается стоимость поиска. Часто дерево поиска можно обрезать, используя только эвристические соображения, и тем самым сводить количество вычислений к разумным пределам.

В этой лекции рассмотрим общий принцип разбиения таких задач на подзадачи и использование в них рекурсии.

2.1. ЗАДАЧА О ХОДЕ КОНЯ

Дана доска n x n, содержащая n^2 полей. Конь, который ходит согласно шахматным правилам, помещается на поле с начальными координатами x_0, y_0. Нужно покрыть всю доску ходами коня, т. е. вычислить обход доски, если он существует, из n^2 — 1 ходов, такой, что каждое поле посещается ровно один раз.

Очевидно, что задачу покрытия n^2 полей можно свести к более простой: или выполнить очередной ход, или установить, что никакой ход невозможен. Поэтому будем строить алгоритм, который пытается сделать очередной ход.

Первая попытка выглядит так:

```
Begin инициация выборки ходов;
    Repeat выбор следующего возможного хода из списка очередных ходов;
        If он приемлем then
        Begin запись хода;
            If доска не заполнена then                    (5.1)
            Begin попытка следующего хода;
```

If *неудача* **then** *стирание предыдущего хода*

End

End

Until *(ход был удачным) или (нет других возможных ходов)*

End

Для того, чтобы более точно описать этот алгоритм, необходимо выбрать некоторое представление для данных. Очевидно, что доску можно представить в виде матрицы, скажем, *h*. Введем также тип индексирующих значений:

Type *index* =1..*n*;

Var *h*: **array** [*index, index*] **of integer**

Так как нужно сохранять историю последовательного «захвата» доски, то необходимо представлять каждое поле доски целым числом, а не булевским значением, которое отражало бы просто факт занятия поля. Очевидно, можно остановиться на таких соглашениях:

h[*x*,*y*]=0: поле [*x*,*y*]не посещалось,
h[*x*,*y*]=*i*: поле [*x*,*y*] посещалось на *i*-м ходу (*i* <= 1 <= n^2).

Теперь необходимо выбрать соответствующие параметры. Они должны определять начальные условия следующего хода, а также сообщать о его удаче или неудаче. Первая задача выполняется заданием координат поля *x*, *y*, с которого делается ход, а также номером хода *i* (для его фиксации). Для решения второй задачи нужен булевский параметр-результат: *q* = *true* означает удачу, *q* = *false* — неудачу.

Какие операторы можно теперь уточнить на основе этих решений? Разумеется, «доска не заполнена» можно выразить как «$i < n^2$». Кроме того, если ввести две локальные переменные *u* и *v* для обозначения координат возможного хода, определяемых по правилам хода коня, то предикат «приемлем» можно выразить как логическую конъюнкцию двух условий: чтобы новое поле находилось на доске, т. е. 1 <= *u* <= *n* и 1 <= *v* <= *n*, и чтобы оно ранее не посещалось, т. е. *h*[*u*, *h*]= 0. Фиксация допустимого хода выполняется с помощью присваивания *h*[*u*, *v*]: = *i*, а отмена хода (стирание)—как *h*[*u*, *v*]:=0. Если при рекурсивном вызове этого алгоритма в качестве параметра-результата передается локальная переменная *q*1, то вместо «ход был удачным» можно подставить *q*1. Таким образом, мы приходим к программе:

```
Procedure try (i: integer; x, y: index, var q: Boolean);
Var v, u: integer;
      ql: Boolean;
Begin инициация выбора ходов;
   Repeat пусть u, v — координаты следующего хода, определяемого
шахматными правилами;
     If (1<=u<=n) and (l<=v<=n) and (h [u, v]=0) then
      Begin h [u, v]: = i;
            If i < sqr (n) then
            Begin try (i+1,u, v, ql);
                 If not ql then h [u, v]: =0                    (5.2)
            End else q1: = true
        End
   Until ql or (нет других ходов);
  q: =ql
End
```

Еще один этап уточнения, и можно будет написать программу уже полностью на нашем языке программирования. Надо заметить, что до сих пор программа разрабатывалась совершенно независимо от правил хода коня. Теперь пора обратить на них внимание. Если задана начальная пара координат (x,y), то имеется восемь возможных координат $<u,v>$ следующего хода. На рис.5.1 они пронумерованы от 1 до 8.

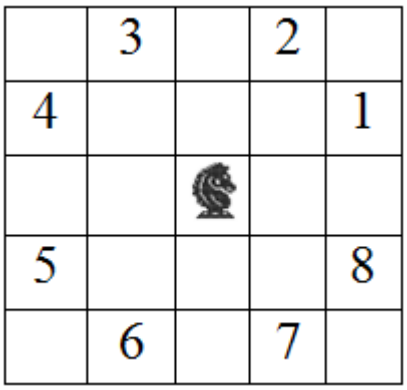


Рис. 5.1. Восемь возможных ходов коня

Получать u, v из x, y просто — будем прибавлять к ним разности координат, помещенные либо в массиве пар разностей, либо в двух массивах отдельных разностей. Пусть эти массивы обозначены через a и b и соответствующим образом инициированы. Для нумерации следующего возможного хода можно использовать индекс k. Подробности показаны в программе ниже. Рекурсивная процедура вызывается в первый раз с параметрами x_0, y_0 —

координатами поля, с которого начинается обход. Этому полю присваивается значение 1, остальные поля маркируются как свободные:

h [x_0, y_0]: = 1;
try (2, x_0, y_0, *q*)

Не следует упускать еще одну деталь. Переменная *h*[*u*,*v*] существует лишь в том случае, когда *u* и *v* находятся внутри границ массива 1 .. *n*. Следовательно, выражение в (5.2), подставленное вместо «он приемлем» в (5.1), осмысленно, только если его первые две составляющие истинны. В программе 5.1 это условие подходящим образом переформулировано, кроме того, двойное отношение 1 <= *u* <= *n* заменено выражением *u* **in** [1,2, ..., *n*], которое при достаточно малых *n* обычно бывает более эффективным. В таблице 5.1 приведены решения, полученные при исходных позициях <1,1>, <3,3> для *n* = 5 и <1,1> для *n* = 6.

Параметр-результат *q* и локальную переменную *q*1 можно заменить глобальной переменной и тем самым несколько упростить программу.

```
Program Knighstour;
Const n= 5; nsq = 25;
Type index =1.. n;
Var i, j: index;
    q: Boolean',
     s: set of index;
     a,b: array [1.. 8] of integer;
     h: array [index, index] of integer;
Procedure try (i: integer; x, y: index; Var ql: Boolean);
Var k,u,v. integer; ql: Boolean;
Begin
 k := 0;
Repeat
   k := k+l; ql := false;
   u:=x + a[k]; v:=y + b[k];
   If (u in s) and (v in s) then
   If h [u, v] =. 0 then
   Begin h [u, v]: = i;
            If i < nsq then
              Begin try (i+l, u, v, q1);
                 If not ql then h [u, v]: = 0
             End else ql: = true
    End
 Until ql or (k=8);
 q := ql
End {try} ;
```

```
Begin
 s := [1,2,3,4,5];
 a[l] := 2;          b[l] := 1;
 a[2]:=1;            b[2]:=2;
 a[3]:=-1;           b[3]:=2;
 a[4]:=-2;           b[4]:=1;
 a[5]:=-2;           b[5]:=-1;
 a[6]:=-1;           b[6]:=-2;
 a[7]:=1;            b[7]:=-2;
 a[8]:=2;            b[8]:=-1;
 For i: =1 to n do
   For j: = 1 to n do h [i, j]: = 0;
 h[1,1]: = 1; try(2,l,1,q);
 If q then
      For i: = 1 to n do
      Begin for j: = 1 to n do write (h [i, j]: 5);
        Writeln;
      End
 Else writeln('Нет решения ')
End.
```

Программа 5.1. Ход коня

Таблица 5.1. Три обхода конем

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

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

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

Каким образом теперь можно обобщить этот пример? Какой схеме, типичной для задач подобного рода, он следует? Какие выводы можно сделать, изучая его?

Характерная черта этого алгоритма состоит в том, что он предпринимает какие-то шаги по направлению к общему решению, эти шаги фиксируются (записываются), но можно возвращаться обратно и стирать записи, если оказывается, что шаг не приводит к решению, а заводит в «тупик». Такое действие называется **возвратом**. Из схемы (5.1) можно вывести общую схему (5.3), если предположить, что число возможных дальнейших путей на каждом шаге конечно:

Procedure try;
Begin инициировать выборку возможных шагов;
Repeat выбрать следующий шаг;
If приемлемо **then**
Begin записать его;
If решение неполно **then**
Begin попробовать очередной шаг;

```
            If неудачно then стереть запись                    (5.3)
      End
  End
Until удача или больше нет путей
End
```

Разумеется, в конкретных программах эта схема может воплощаться различными способами. Часто в них используется явный параметр уровня, обозначающий глубину рекурсии и допускающий простое условие окончания.

Если, кроме того, на каждом шаге число исследуемых дальнейших путей фиксировано, скажем равно *m*, то используется схема (5.4), вызываемая оператором «*try*(l)»:

```
Procedure try (i: integer);
Var k: integer;
Begin
 k: =0;
Repeat
      k: = k+1; выбрать k-й возможный путь;
     If приемлемо then
     Begin записать его;
        (5.4)
            If i < n then
            Begin try (i + 1);
                 If неудачно then стереть запись
             End
     End
 Until удачно или (k = m)
End
```

2.2. МОГУТ ЛИ ВОСЕМЬ ФЕРЗЕЙ НЕ БИТЬ ДРУГ ДРУГА?

Исходным пунктом обсуждения будет такая головоломка : могут ли восемь ферзей (на шахматной доске 8*8) не бить друг друга? Более точно – требуется найти все расстановки восьми ферзей, в которых никакие два ферзя не стоят на одной вертикали ,горизонтали или диагонали(если такие позиции вообще существуют).

Прежде чем обсуждать вопрос о написании программы, решающей эту задачу, расскажем об истории ее решения без машины. Задача поставлена в 1848 году немецким шахматистом М.Беццелем. В июне 1850г. Ф.Наук опубликовал 60 решений . Великому математику К.Гауссу удалось найти 72 решения. Однако вскоре его результат перекрыл тот же Наук,нашедший 92 решения. В 1874г. английский математик Д.Глешер доказал, что больше решений не существует.

Вернемся к нашей головоломке. Легко заметить, что на каждой вертикали должен стоять один ферзь, поскольку вертикалей столько же сколько ферзей,а никакие два ферзя не должны стоять на одной вертикали. Аналогичным образом на каждой горизонтали должен стоять один ферзь. Если бы у нас были не ферзи, а ладьи, то эти условия были бы достаточными, чтобы ладьи не били друг друга. Знакомые с комбинаторикой легко сообразят, что в случае с ладьей существует 8!=1*2*3*…*8 решений.

В нашем случае (когда расставляют не ладей, а ферзей) существенны и диагонали, так что задача, видимо, сложнее. Попытаемся перебрать все варианты и найти среди них искомые. Приняв такое решение, сталкиваемся со следующими вопросами: нам нужно быть уверенными, что в ходе перебора мы не пропустим ни одного решения головоломки. Кроме того, было бы желательно не рассматривать одну и ту же позицию многократно, чтобы по возможности избежать лишней работы. Таким образом, нам нужно составить алгоритм (план) перебора, позволяющий достигнуть этих целей. Чесно говоря это и смоставляет основную тему данной главы, азадача о восьми ферзях является лишь поводом. Поэтому, составив интересующий нас алгоритм, мы не будем пытаться его выполнить ни вручную, ни с помощью компьютера.

Итак, мы хотим перебрать и испробовать все варианты, чтобы быть уверенными, что ни одного решения головоломки не пропустили. Однако выражение “все варианты” не ясно: хотим ли мы рассматривать все расста-

новки ферзей на доске ? или все расстановки восьми ферзей? Или все расстановки, где на каждой горизонтали стоит по ферзю? или еще что-нибудь?

Нужно, таким образом, определить, какие конфигурации будут рассматриваться в качестве “кандидатов” в решении головоломки. Здесь есть две опасности. Первая состоит в том, что кандидатов окажется много. А ведь чем их больше, тем более трудоемким окажется перебор. Этой опасности можно было бы избежать, считая кандидатами расстановки восемь ферзей, в которых никакие два ферзя не бьют друг друга. Тогда количество кандидатов, разумеется, минимально, но не ясно, как их перебирать: уметь их перебирать значит уметь решать исходную головоломку.

Мы будем представлять себе позицию на доске как результат появления на доске ферзей друг за другом. Поскольку порядок появления ферзей безразличен, и на каждой горизонтали должно стоять по ферзю, можно представлять себе, что ферзей ставят снизу вверх – сначала ставится ферзь на первую горизонталь, затем на вторую и т.д. Может оказаться, что ферзь, поставленный последним, попадает под бой уже стоящих на доске. В этом случае наша позиция бесперспективна и ставить ферзей дальше смысла нет.

Возникает “дерево вариантов”. Внизу, в его корне, находится пустая позиция – позиция, в которой нет ни одного ферзя. Выше нее нарисованы все позиции, которые могут получиться, когда мы поставим ферзя на первую горизонталь. И с каждой такой позиции можно получить восемь позиций, поставив следующего ферзя на одну из клеток второй горизонтали.

Дерево вариантов строится следующим образом. Внизу рисуем пустую позицию. Далее применяется такое правило: над каждой уже имеющейся в дереве позицией, в которой ферзи не бьют друг друга и есть свободная горизонталь, рисуем восемь других, соответствующих восьми возможным положениям ферзя на нижней из свободных горизонталей, и соединяем их с ней линиями. Таким образом, “ листьями” нашего дерева будут, во-первых, бесперспективные позиции и, во-вторых, решения нашей головоломки, т.е. позиции, в которых все горизонтали заполнены ферзями и они друг друга не бьют.

2.2.1. ЗАДАЧА О ВОСЬМИ ФЕРЗЯХ

Задача о восьми ферзях — хорошо известный пример использования метода проб и ошибок и алгоритмов с возвратом. В 1850 г. ею занимался К. Ф. Гаусс, но полного ее решения он не дал. Это и не удивительно. Для подобных задач характерно отсутствие аналитического решения. Вместо этого они требуют большого объема изнурительных вычислений, терпения и аккуратности. Поэтому такие задачи стали почти исключительно прерогативой вычислительных машин, которые обладают этими свойствами в гораздо большей степени, чем люди, даже гениальные.

Задача о восьми ферзях поставлена следующим образом: нужно так расставить восемь ферзей на шахматной доске, чтобы ни один ферзь не угрожал другому.

Используя схему (5.4) в качестве образца, мы легко получаем следующую предварительную версию алгоритма:

```
procedure try(i: integer);
begin
инициировать выбор позиции для i-го ферзя;
 repeat выбрать позицию;
        if безопасно then
          begin поставить ферзя;                                    (5.5)
             if i < 8 then
                begin try(i+l);
                   if неудачно then убрать ферзя
                end
          end
until удачно or нет больше позиции
end
```

Чтобы идти дальше, нужно выбрать некоторое представление для данных. Поскольку мы знаем, что по шахматным правилам ферзь бьет все фигуры, расположенные на той же горизонтали, вертикали или диагонали доски, то мы заключаем, что каждая вертикаль может содержать одного и только одного ферзя, так что *j*-го ферзя можно сразу помещать на *i*-ю вертикаль. Итак, параметр *i* становится индексом вертикали, а выбор позиции ограничивается восемью возможными значениями индекса горизонтали *j*.

Осталось решить, как представить расположение восьми ферзей на доске. Очевидно, что доску можно было бы вновь изобразить в виде квадратной матрицы, но после некоторого размышления мы обнаруживаем, что та-

кое представление значительно усложнило бы проверку безопасности позиции. Это крайне нежелательно, поскольку такая операция выполняется наиболее часто. Поэтому нужно выбрать представление, которое насколько возможно упростит эту проверку. Лучше всего сделать наиболее доступной ту информацию, которая действительно важна и чаще всего используется. В нашем случае это не расположение ферзей, а информация о том, помещен ли ферзь на данной горизонтали или диагонали. (Мы уже знаем, что на каждой *k*-ой вертикали уже помещен один ферзь для $1 \le k \le$ i.) Это приводит к следующим описаниям переменных:

```
var x : array [1 .. 8] of integer;
    a : array [1.. 8] of  Boolean;
    b : array [b1 .. b2] of Boolean;                    (5.6)
    c : array [cl.. c2] of Boolean;
```

x[i] указывает позицию ферзя на *i-й* вертикали;

a[j] означает, что на j-й горизонтали нет ферзя;

b [k] означает отсутствие ферзя на ↙ *k-й* > - диагонали;

c [k] означает отсутствие ферзя на ↘*k-й* - диагонали.

Выбор индексных границ *b1, b2, c1, c2* определяется, исходя из способа, которым вычисляются индексы *b и c;* мы замечаем, что на ↙-диагонали все поля имеют одну и ту же сумму координат *i* и, а на↘-диагонали постоянна разность координат *i* - j. Соответствующее решение показано в программе 5.2.

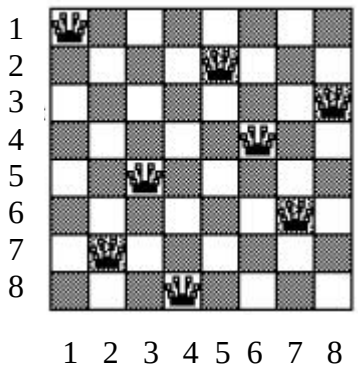


Рис. 5.2. Одно из решений задачи о восьми ферзях

При таких данных оператор «*поставить ферзя*» принимает следующую форму:

x [i]:=*j*; *a* [j]: = *false;* *b [i* + j] := *false;* *c [i — j]* := *false*

При уточнении оператора «убрать ферзя» мы получаем

a[j]:=true; b [i + j] := true; c[i — j]:=true

а условие *«безопасно»* выполняется, если поле <i,j> находится на горизонтали и диагонали, которые еще свободны (представлены как *true),* что можно описать логическим выражением

a[j]/\b[i + j]/\c[i-j]

Этим завершается разработка алгоритма, который полностью описан в программе 5.2. Она дает решение x = (1, 5, 8, 6, 3, 7, 2, 4), которое изображено на рис. 5.2.

```
program eightqueen1;
{поиск одного решения задачи о восьми ферзях}
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] and b[i+j] and c[i-j] then
                begin x[i]:= j;
                        a[j] := false; b[i+j] := false; c[i-j] := false;
                         if i< 8 then
                         begin try(i+ 1,q);
                         if not q then
                                 begin a[j] := true; b[i+j] :=true; c[i-j] := true
                                 end
                        end else q :=true
                 end
        until q or (j=8)
end {try} ;
begin
        for i:=1 to 8 do a[i] := true;
        for i:=2 to 16 do b[i] :=true;
        for i:= -7 to 7 do c[i] := true;
        try (1,q);
        if q then
                for i := 1 to 8 do write (x[i]:4);
        writeln
end.
```

Программа 5.2. Восемь ферзей (одно решение).

Прежде чем закончить разбор задач, связанных с шахматной доской, мы воспользуемся задачей о восьми ферзях для иллюстрации важного обобщения алгоритма проб и ошибок. В общих словах это обобщение заключается в том, чтобы находить не одно, а все решения поставленной задачи.

К этому обобщению легко перейти. Вспомним, что множество возможных путей строится по строго определенной системе, так чтобы никакой путь не предлагался более одного раза. Это свойство алгоритма соответствует поиску по дереву, при котором каждый узел посещается только один раз. Это позволяет — после того как решение найдено и должным образом зафиксировано — просто переходить на следующий возможный путь. Общая схема такого алгоритма (5.7) получена из (5.4):

```
procedure try(i: integer);
var k: integer;
begin
      for k := 1 to m do
            begin выбор k-го пути;
                  if приемлемо then                                    (5.7)
                        begin запись его
                         if i < n then try(i+l) else печать решения
                        стирание записи
                         end
             end
 end
```

Заметим, что благодаря тому, что условие окончания цикла упростилось до одной составляющей k — m, оператор цикла с постусловием естественно заменить на оператор цикла с параметром. К удивлению, поиск *всех* возможных решений описывается более простой программой, чем поиск одного решения.

Обобщенный алгоритм, который находит все 92 решения задачи о восьми ферзях, представлен в программе 5.3. На самом деле существует только 12 принципиально различных решений; наша программа не учитывает симметрию.

Таблица 5.2. Двенадцать решений задачи о восьми ферзях

X1	X2	X3	X4	X5	X6	X7	X8	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	72
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

В табл. 1.2 приведены первые 12 решений. Число N указывает частоту проверок безопасности полей. Ее среднее значение для всех 92 решений равно 161.

```
program eightqueens;
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 write(x[k]: 4);
writeln
end {print} ;
procedure try(i: integer);
var j: integer; begin
        for j:=1 to 8 do
                if a[j] and b[i+j] and c[i-j] then
                begin           x[i] :=j;
                        a[j] := false; b[i+j] := false; c[i-j] := false;
                         if i < 8 then try(i+1) else print;
                         a[j] := true; b[i+j] :=true; c[i-j]:=true
                end
```

```
end {try} ;
begin
        for i:=1 to 8 do a[i]:=true ;
        for i:=2 to 16 do b[i]:=true;
        for i:= -7 to 7 do c[i]:=true;
        try(1)
end.
```

Программа 5.3. Восемь ферзей (все решения).

2.3. Задача об устойчивых браках

Пусть даны два непересекающихся множества *A* и *B* с одинаковыми кардинальными числами, равными *n*. Нужно найти некоторое множество пар <*a, b* > из *n*, такое, чтобы *a* **in** *A* и *b* **in** *B* удовлетворяли некоторым ограничениям. Для выбора таких пар существует много разных критериев; один из них называется «правилом устойчивых браков».

Предположим, что *A* – множество мужчин, а *B* – множество женщин. Каждый мужчина и каждая женщина устанавливают определенный порядок предпочтения для своих возможных партнеров по браку. Если *n* пар выбрано таким образом, что существуют какие-то мужчина и женщина, не состоящие в браке друг с другом, но предпочитающие друг друга своим действительным супругам, то такое множество считается *неустойчивым*. Если же такой пары не существует, то множество называется *устойчивым*.

Эта ситуация типична для многих похожих задач, в которых распределение зависит от каких-то предпочтений, как, например, выбор школы учащимися, выбор рекрутами различных родов войск и т. д. Пример с браками отчасти выбран интуитивно; заметим, однако, что установленный порядок предпочтений инвариантен и не изменяется по мере образования пар. Это упрощает задачу, но и несколько искажает действительность (как любая абстракция).

Решение можно искать следующим образом: пробовать последовательно объединять в пары члены двух множеств, пока оба множества не будут исчерпаны. Пусть *try (m)* – алгоритм поиска партнерши для мужчины *m*, и пусть поиск происходит согласно списку предпочтений этого мужчины.

Приведем первую версию, основанную на этих допущениях:

```
procedure  try(m: man);
```

```
    var r : rank;
begin
    for r := 1 to n do
    begin выбрать r-ю женщину из списка предпочтений
        мужчины m;
    if приемлемо then
        begin записать брак;                                    (4.20)
            if m – не последний мужчина
              then try(succ(m))
              else записать устойчивое множество;
            отменить брак
        end
    end
end
```

Вновь мы не можем двигаться дальше, пока не решим, как представлять данные. Определим три скалярных типа; для простоты пусть их значения будут целыми числами от 1 до *n*. Хотя формально эти три типа одинаковы, присваивание им различных имен значительно проясняет программу. В частности, сразу понятно, что означает какая-либо переменная:

```
type man = 1 . . n;
     woman = 1 . . n;                                           (4.21)
     rank = 1 . .n ;
```

Исходные данные представляются двумя матрицами, в которых указан порядок предпочтений мужчин и женщин их партнерами:

```
var wmr: array [man, rank] of woman
    mwr: array [woman, rank] of man                             (4.22)
```

wmr [*m*] обозначает список предпочтений, установленный мужчиной *m*, т.е. *wmr* [*m*][*r*]=*wmr* [*m*, *r*] – женщина, которая занимает *r*-е место в списке предпочтений мужчины *m*. Точно так же *mwr* [*w*] – список предпочтений женщины *w*, а *mwr* [*w*, *r*] – мужчина, занимающий *r*-е место в ее списке.

Результат предоставляется в виде массива женщин *x*, такого, что *x*[*m*] обозначает партнершу мужчины *m*. Для сохранения симметрии (называемой

также «равноправием») между мужчинами и женщинами, вводится дополнительный массив *y*, такой, что *y*[*w*] обозначает партнера женщины *w:*

var *x:* **array** [*man*] **of** *woman;*

y: **array** [*woman*] **of** *man;* *(4.23)*

Ясно, что в массиве *y* нет особой нужды, поскольку он содержит информацию, которая уже представлена в массиве *x*. В самом деле, для любых *m* и *w*, состоящих между собой в браке, выполняются равенства

$$x[y[w]] = w, \quad y[x[m]] = m \qquad (4.24)$$

Следовательно, значение y[w] можно установить просто поиском по x; но использование массива y явно повышает эффективность. Информация, представленная в массивах x и y, нужна для определения устойчивости предлагаемого множества браков. Поскольку это множество строиться постепенно, путем выбора отдельных пар и проверки устойчивости множества после каждого такого предлагаемого брака, x и y используются еще до того, как будут заполнены. Для того чтобы знать, какие компоненты уже определены, можно ввести булевские массивы

*singlem***: array**[*man*] **of** *boolean*

*singlew***: array**[*woman*] **of** *boolean* *(4.25)*

с такими значениями:

- *singlem* [*m*] *предполагает, что x*[*m*] *определено*
- *singlew* [*w*] *предполагает, что y*[*w*] *определено.*

Но, рассматривая предлагаемый алгоритм, легко обнаружить, что семейное положение мужчины можно определить просто по значению *m* следующим образом:

$$singlem[k] \equiv k < m. \qquad (4.26)$$

Поэтому массив s*inglem* можно удалить; соответственно мы упростим имя s*inglew* до *single.*

После соответствующих соглашений алгоритм принимает вид (4.27). Предикат *«приемлемо»* можно изобразить в виде конъюнкции *single* и *stable («устойчивый»),* где *stable* – функция, которую еще надо будет уточнить.

procedure *try(m: man);*

```
var r : rank; w: woman;
begin
    for r:= 1 to n do
begin w := wmr[m, r];
        if  single[w] Λ  stable then                                    (4.27)
        begin x[m]:=w; y[w]:=m; single[w] := false;
            if m <n  then try(succ(m))
               else  запись устойчивого множества;
           single[w]:=true
         end
     end
end
```

Теперь основная задача – уточнить алгоритм определения устойчивости. Прежде всего нужно иметь ввиду, что устойчивость по определению следует из сравнения предпочтений, или рангов. Но ранги мужчин и женщин нигде в наших описанных до сих пор данных явно не представлены. Конечно, ранг женщины *w* с точки зрения мужчины m можно вычислить, но лишь с помощью трудоемкого поиска *w* в *wmr* [*m*].

Так как вычисление устойчивости – очень частое действие, желательно, чтобы эта информация была очень доступна. Для этого мы будем использовать две матрицы

```
rmw: array[man, woman] of rank;
rwm: array[woman, man] of  rank;                                        (4.28)
```

такие, что *rmw[m, w]* означает ранг *w*-й женщины в списке предпочтений *m*-го мужчины, а *rwm[w,m]* – ранг *m*-го мужчины в списке *w*-й женщины. Ясно, что значения этих вспомогательных массивов постоянны и могут быть получены с самого начала из значений *wmr* и *mwr*.

Значение предиката *stable* ("устойчивый") теперь вычисляется в строгом соответствии с его исходным определением. Вспомним, что мы исследуем возможность брака между *m* и *w*, где *w= wmr[m, r]*, т.е. *w* имеет ранг r в *m*-м списке предпочтений. Будучи оптимистами, мы вначале предполагаем, что устойчивость сохранилась, а затем пытаемся найти возможные источники

неприятностей. Где они могут таиться? Есть две симметричные возможности:

1. Может существовать женщина *pw*, предпочтительная для *m* *по* сравнению с *w*, которая, сама предпочитает *m* своему мужу.

2. Может существовать мужчина *pm*, предпочтительный для w по сравнению с *m*, который сам предпочитает *w* своей жене.

Исследуя источник неприятностей 1, мы сравниваем ранги *rwm[pw, m]* и *rwm[pw, y[pw]]* для всех женщин, которых m предпочитает своей невесте *w*, т. е. для всех pw = wmr[m, i], таких, что $i < r$. Мы знаем, что все эти женщины уже выданы замуж. Так как если бы какая-то из них была еще одинока, *m* выбрал бы ее раньше, чем *w*. Этот процесс поверки можно сформулировать в виде простого линейного поиска; *s* означает устойчивость:

```
s := true; i:=1;
while (i < r) Λ s do
begin pw: = wmr[m, i]; i:=i+1;
      if ¬ single[pw] then                                  (4.29)
      s := rwm[pw, m] > rwm[rw, y[pw]]
end
```

Исследуя источник неприятностей 2, мы должны рассмотреть всех мужчин *pm* , которых w предпочитает своему предполагаемому партнеру m, т. е. всех $pm = mwr[w, i]$, таких, что $i < rwm[w, m]$. Как и при исследовании источника 1, нужно сравнивать ранги *rmw[pm, w]* и *rmw[pm, x[pm]]*. Но здесь нужно быть внимательными, чтобы не производить сравнений с участием *x[pm]*, где *pm* еще женат. Необходимой предосторожностью будет проверка $pm < m$, поскольку мы знаем, что все мужчины, предшествующие *m*, уже женаты.

По своей сути этот алгоритм основан на простой схеме поиска с возвратом. Его эффективность зависит в основном от удачного построения схемы усекания дерева решения. Несколько более быстрый, но более сложный и менее понятный алгоритм предложен Мак-Вити и Уилсоном, которые, кроме того, распространили его на случай множеств (мужчин и женщин) неодинакового размера.

Текст программы

```
program marriage;
uses crt;
const n=8;
type  man=1..n;
      woman=1..n;
      rank=1..n;
var m: man;  r: rank;  w: woman;
    wmr: array [man, rank]of woman;
    mwr: array [woman, rank]of man;
    rmw: array [man, woman]of rank;
    rwm: array [woman, man]of rank;
      x: array [man]of woman;
      y: array [woman]of man;
    single: array [woman]of boolean;
    bra: text;
    i, l: integer;
    rez: byte;
procedure print;
var m: man; rm, rw: integer;
begin
 rez:=rez+1;rm:=0;rw:=0;
 write(rez:2 ,':');
 for m:=1 to n do
    begin
      write(x[m]:4);
      rm:=rm+rmw[m,x[m]];
      rw:=rw+rwm[x[m],m];
    end;
writeln(rm:8,rw:4,l:4);l:=0;
end;
```

```
procedure try(m:man);
var r: rank; w: woman;

   function stable: boolean;
   var pm: man; pw: woman;
       i, lim: rank;
       s: boolean;
  begin
   s:=true; i:=1;l:=l+1;
   while (i<r) and s do
     begin
       pw:=wmr[m, i];i:=i+1;
       if not (single[pw]) then
       s:=rwm[pw, m]>rwm[pw, y[pw]];
     end;
   i:=1;lim:=rwm[w, m];
   while (i<lim) and s do
     begin
       pm:=mwr[w, i];i:=i+1;
       if pm<m then s:=rmw[pm, w]>rmw[pm, x[pm]];
     end;
stable:=s;
end;{of stable}

begin {of try}
 for r:=1 to n do
    begin
      w:=wmr[m,r];
      if single[w] then
        if stable then
```

```
          begin
            x[m]:=w; y[w]:=m;
           single[w]:=false;
              if m<n then try(succ(m)) else
              print;
              single[w]:=true;
          end;
      end;
end;{of try}

begin{of main}
 clrscr;
wmr[1,1]:=7;wmr[1,2]:=2;wmr[1,3]:=6;wmr[1,4]:=5;
wmr[1,5]:=1;wmr[1,6]:=3;wmr[1,7]:=8; wmr[1,8]:=4;
wmr[2,1]:=4;wmr[2,2]:=3;wmr[2,3]:=2;wmr[2,4]:=6;
wmr[2,5]:=8;wmr[2,6]:=1;wmr[2,7]:=7; wmr[2,8]:=5;
wmr[3,1]:=3;wmr[3,2]:=2;wmr[3,3]:=4;wmr[3,4]:=1;
wmr[3,5]:=8;wmr[3,6]:=5;wmr[3,7]:=7; wmr[3,8]:=6;
wmr[4,1]:=3;wmr[4,2]:=8;wmr[4,3]:=4;wmr[4,4]:=2;
wmr[4,5]:=5;wmr[4,6]:=6;wmr[4,7]:=7; wmr[4,8]:=1;
wmr[5,1]:=8;wmr[5,2]:=3;wmr[5,3]:=4;wmr[5,4]:=5;
wmr[5,5]:=6;wmr[5,6]:=1;wmr[5,7]:=7; wmr[5,8]:=2;
wmr[6,1]:=8;wmr[6,2]:=7;wmr[6,3]:=5;wmr[6,4]:=2;
wmr[6,5]:=4;wmr[6,6]:=3;wmr[6,7]:=1; wmr[6,8]:=6;
wmr[7,1]:=2;wmr[7,2]:=4;wmr[7,3]:=6;wmr[7,4]:=3;
wmr[7,5]:=1;wmr[7,6]:=7;wmr[7,7]:=5; wmr[7,8]:=8;
wmr[8,1]:=6;wmr[8,2]:=1;wmr[8,3]:=4;wmr[8,4]:=2;
wmr[8,5]:=7;wmr[8,6]:=5;wmr[8,7]:=3; wmr[8,8]:=8;

 writeln(' Исходные данные: ');
```

```
 writeln('         Партнерши для мужчин');
 for m:=1 to n do
     begin
      for r:=1 to n do
       begin
        write(wmr[m,r]:4);
        rmw[m, wmr[m,r]]:=r;
       end;
       assign(bra,'bra.txt');
       reset(bra);
      writeln;
   end;

writeln('         Партнеры для женщин');
 for w:=1 to n do
     begin
      for r:=1 to n do
       begin
         read(bra, mwr[w, r]);
         write(mwr[w, r]:4);
         rwm[w, mwr[w, r]]:=r;
       end;
     readln(bra);
     writeln;
    end;
close(bra);
readln;
writeln('   Решение :');
for w:=1 to n do single[w]:=true;
rez:=0;
```

```
try(1);
readkey;
end.
```

Таблица 1 содержит множество входных данных, соответствующих массивам *wmr* и *mwr*. И наконец, в таблице 2 приведены девять полученных решений.

Таблица 1. **Пример входных данных для задачи об устойчивых браках**

Ранг	1	2	3	4	5	6	7	8
Мужчина 1 выбирает женщину	7	2	6	5	1	3	8	4
Мужчина 2 выбирает женщину	4	3	2	6	8	1	7	5
Мужчина 3 выбирает женщину	3	2	4	1	8	5	7	6
Мужчина 4 выбирает женщину	3	8	4	2	5	6	7	1
Мужчина 5 выбирает женщину	8	3	4	5	6	1	7	2
Мужчина 6 выбирает женщину	8	7	5	2	4	3	1	6
Мужчина 7 выбирает женщину	2	4	6	3	1	7	5	8
Мужчина 8 выбирает женщину	6	1	4	2	7	5	3	8

Женщина 1 выбирает мужчину	4	6	2	5	8	1	3	7
Женщина 2 выбирает мужчину	8	5	3	1	6	7	4	2
Женщина 3 выбирает мужчину	6	8	1	2	3	4	7	5
Женщина 4 выбирает мужчину	3	2	4	7	6	8	5	1
Женщина 5 выбирает мужчину	6	3	1	4	5	7	2	8
Женщина 6 выбирает мужчину	2	1	3	8	7	4	6	5
Женщина 7 выбирает мужчину	3	5	7	2	4	1	8	6
Женщина 8 выбирает мужчину	7	2	8	4	5	6	3	1

Таблица 2. Решения задачи об устойчивых браках

		X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	rm	rw	c^*
Решение	1	7	4	3	8	1	5	2	6	16	32	21
	2	2	4	3	8	1	5	7	6	22	27	449
	3	2	4	3	1	7	5	8	6	31	20	59
	4	6	4	3	8	1	5	7	2	26	22	62
	5	6	4	3	1	7	5	8	2	35	15	47
	6	6	3	4	8	1	5	7	2	29	20	143
	7	6	3	4	1	7	5	8	2	38	13	47
	8	3	6	4	8	1	5	7	2	34	18	758
	9	3	6	4	1	7	5	8	2	43	11	34

(c* - количество проверок устойчивости
Решение 1- решение, оптимальное для мужчин.
Решение 9- решение, оптимальное для женщин)

Алгоритм такого вида, дающий все возможные решения задачи (при определенных ограничениях), часто используется для выбора одного или нескольких решений, которые в каком-то смысле являются оптимальными. Таким образом, можно было бы, допустим, интересоваться решением, которое в среднем больше удовлетворяет мужчин, или женщин, или всех. Отметим, что в табл. 2 указаны суммы рангов всех женщин с точки зрения их мужей и суммы рангов всех мужчин с точки зрения женщин. Это значения

Решение с наименьшим значением rm называется устойчивым решени-

$$rm = \sum_{m=1}^{n} rmw\,[m, x[m]], \qquad rw = \sum_{m=1}^{n} rwm[x[m], m].$$

ем, оптимальным для мужчин, а с наименьшим rw – устойчивым решением, оптимальным для женщин. При избранной стратегии поиска первыми вычисляются решения, хорошие с точки зрения мужчин, а благоприятные для женщин решения даются в конце работы. В этом смысле алгоритм ориентирован на сильный пол. Но его можно быстро изменить, систематически поменяв ролями мужчин и женщин, т.е. заменив mwr на wmr и rmw на rwm.

2.4. ЗАДАЧА ОПТИМАЛЬНОГО ВЫБОРА

Наш последний пример алгоритма с возвратом — это логическое обобщение двух предыдущих алгоритмов, которые строятся по общей схеме (3.35). Вначале мы использовали принцип возврата для нахождения *одного* решения данной задачи. Это было показано на примерах хода коня и задачи о восьми ферзях. Затем мы задались целью найти все решения задачи; примерами были восемь ферзей и устойчивые браки. Теперь мы хотим найти *оптимальное решение*.

Для этого нужно получить все возможные решения и в процессе их получения оставлять только то, которое в некотором смысле является оптимальным. Если предположить, что оптимальность определена с помощью некоторой функции f(s), принимающей положительные значения, то алгоритм можно получить из схемы (3.35) заменой оператора «печать решения» на оператор

```
if f(solution) > f(optimum) then optimum := solution   (3.47)
```

В переменной optimum хранится лучшее из полученных до сих пор решений. Разумеется, ее нужно должным образом инициировать; кроме того, значение f(optimum) принято хранить в другой переменной, чтобы избежать ее частого перевычисления.

В качестве примера общей задачи поиска оптимального решения мы выбираем следующую важную и часто встречающуюся задачу: найти оптимальную выборку из заданного множества объектов, подчиненную некоторым ограничениям. Выборки, составляющие приемлемые «решения, строятся постепенно, с помощью исследования отдельных объектов базового множества. Процедура *try* описывает процесс исследования пригодности объекта для включения в выборку; она вызывается рекурсивно при переходе к следующему объекту, пока все объекты не будут рассмотрены.

Мы видим, что из рассмотрения каждого объекта можно сделать два возможных вывода: либо включить объект в текущую выборку, либо не включать его. Поэтому здесь не удастся использовать операторы цикла; вместо этого нужно явно описать оба случая. Это показано в (3.48); предположим, что объекты пронумерованы

```
1, 2, ..., n:
```

```
procedure try(i: integer);
begin
1: if включение приемлемо then
        begin включить i-й объект;
            if i<n then try(i + 1) else проверить оптимальность;
          удалить i-й объект                                    (3.48)
        end;
2:  if  невключение  приемлемо  then  if i < n then try(i + 1)
                else проверить оптимальность
end
```

Из этой схемы видно, что всего имеются 2^n возможные выборки; поэтому ясно, что критерии приемлемости должны значительно ограничить количество рассматриваемых возможностей. Для пояснения возьмем конкретный пример: пусть каждый из n объектов $a_1,\ldots, a_n$ обладает весом w и ценностью v. Оптимальным пусть считается множество с наибольшей суммарной ценностью компонент, а ограничением пусть служит их предельный общий вес. Эта задача хорошо знакома всем отправляющимся в путешествие, которые упаковывают чемоданы, стараясь так выбрать n предметов, чтобы их общая ценность была максимальной, а общий вес не превышал какого-то допустимого предела.

Теперь мы можем решить, как представить известные факты в виде данных. Из вышесказанного легко получаются такие описания:

```
Type  index = 1 .. n;
Object = record w,v: integer
End;
Var a: array [index] of object;                                 (3.49)
limw, totv, maxv: integer;
s, opts: set of index
```

Переменные *limw* и *totv* обозначают предельный вес и общую ценность всех n объектов. Фактически эти два значения в течение всего процесса отбора постоянны; через *s* обозначаются текущая выборка объектов, где каждый объект представлен своим именем (индексом);*opts* есть оптимальная выборка, полученная до сих пор, а *maxv*—её ценность.

Теперь посмотрим, каковы критерии приемлемости объекта для текущей выборки. Когда речь идет о *включении,* объект можно включить в выборку, если он удовлетворяет допустимому весу. Если же не удовлетворяет, то можно прекратить попытки добавлять новые объекты в текущей выборке. Если рассматривать *невключение*, то критерий приемлемости, т.е. возможности продолжать построение текущей выборки, будет возможность получить без этого объекта такую общую ценность выборки, которая была бы не меньше полученного до сих пор оптимума. Ведь иначе продолжение поиска, хотя и будет давать какие-то решения, никогда не приведет к оптимальному, следовательно, на этом пути бесполезен какой-либо дальнейший поиск. С учетом этих двух условий мы определяем, какие существенные величины нужно вычислять для каждого шага в процессе отбора:

1. Общий вес *tw* выборки *s*, полученной до сих пор.

2. Общую ценность *αv* текущей выборки *s*, которой еще можно достичь.

Эти два значения удобно представить в виде параметров процедуры *try*.

Условие «*включение приемлемо*» в (3.48) теперь можно сформулировать как

$$tw + a[i].w \le limw \qquad (3.50)$$

а последующую проверку оптимальности — как

```
if av > maxv then
begin {запись нового оптимума}
    opts:=s, maxv :=av
end
```

Последнее присваивание связано с тем соображением, что достижимое значение будет получено после просмотра всех n объектов.

Условие «*невключения приемлемо*» в (3.48) выражается как

$$av - a[i].v > maxv \qquad (3.52)$$

Так как позднее оно снова используется, значение $av - a[i].v$ присваивается переменной *avl*, чтобы избежать повторного вычисления.

Всю программу полностью теперь можно получить из (3.48) с помощью (3.52), добавив соответствующие операторы инициации глобальных переменных. Следует отметить, что здесь удобно используются операции над множествами.

```
Program seliction(input, output);
{Поиск оптимальной выборки объектов при ограничениях}
const n = 10;
type index = 1..n;
     object = record v,w: integer  end;
var i: index;
    a:array[index] of object;
    limw, totv, maxv: integer;
    w1, w2, w3: integer;
    s, opts  : set of index;
    z : array[boolean] of char;
procedure try(i: index; tw, av: integer);
   var av1: integer;
begin {попытка включения объекта i}
      if tw+ a[i].w≤ limw then
      begin
            s:= s + [i];
            if  i < n then try( i+1, tw + a[i].w, av) else
                if  av > maxv then
                begin maxv:= av; opts:= s
                end;
             s:= s – [i] ;
      end;
end {try};
begin totv:= 0;
for i:=1 to n do
    with a[i] do
```

```
        begin read(w, v); totv:= totv + v;
        end;
 read(w1, w2, w3);
z[true]:= '*'; z[false]:= ' ';
write('weight');
for i:= 1 to n do write(a[i].w: 4 );
writeln; write('value');
for i:= 1 to n do  write(a[i].v: 4);
          writeln;
repeat limw:= w1; maxv:= 0; s:= [ ]; opts:= [ ];
try(1, 0, totv);
write(limw);
for i:= 1 to n do write('      ', z[i  in  oprs]);
writeln; w1:=w1 + w2;
until w1 > w3
end.
```

Программа 3.7. Оптимальная выборка.

Результат выполнения программы 3 7 с заданными предельными весами от 10 до 120 показан в табл. 3.5.

Таблица 3.5. Пример результата работы программы оптимальной выборки

Вес	10	11	12	13	14	15	16	17	18	19
Значение	18	20	17	19	25	21	27	23	25	24
10	*									
20							*			
30					*		*			
40	*				*		*			
50	*	*		*			*			
60	*	*	*	*	*					
70	*	*			*		*		*	
80	*	*	*		*		*	*		
90	*	*			*		*		*	*
100	*	*		*	*		*	*	*	
110	*	*	*	*	*	*	*		*	
120	*	*			*	*	*	*	*	*