Два подхода в построении решений в задаче о раскладывании колечек по штырькам

Условие задачи:

В каждую клетку шахматной доски воткнут штырек. В нашем распоряжении есть некоторое количество колечек, которые можно нанизывать на штырьки, причем на один штырек можно нанизать несколько колечек. Требуется подсчитать, сколькими способами можно распределить все эти колечки по штырькам. Два распределения считаются разными, если на двух соответствующих штырьках находится разное количество колечек.

Начнем процесс решения

Прежде всего, обратим внимание на одно важное обстоятельство. Два разбиения считаются различными, если в них хотя бы один штырек содержит различное количество колечек и ничего не сказано о взаимном расположении штырьков. Из этого следует, что способ расположения штырьков роли не играет и слова о квадратной доске — это не более чем способ ввести нас в заблуждение. Вытянем доску в линию, линию представим в виде массива, а кольца пусть будут единицами. Тогда нанизывание колечка на штырек сведется к добавлению еще одной единицы к элементу массива. С учетом сказанного условие можно переформулировать так:

Второе условие. Дан одномерный массив длины N и число L. Найти все возможные различные разложения числа L по элементам массива. Массив считается разложением числа L, если сумма его элементов равна L.

А можно сформулировать это же условие вообще без применения программистской терминологии.

Третье условие. Дано число L. Найти все возможные разбиения этого числа на суммы положительных чисел, при условии, что количество слагаемых не превышает N, и перемена места слагаемого дает другое разбиение.

Это хороший пример формализации условия, то есть записи условия на специальном языке, слова которого имеют четко определенный смысл. Сформулировать задачу несколькими способами всегда полезно. Это дает дополнительную информацию к размышлению, а иногда удачная переформулировка даже может подсказать и очень изящное решение. В нашем случае как минимум удалось избавиться от лишнего понятия доски и привести все к понятиям, легко переводимым на язык программирования.

Третье условие удобно для понимания, но оно, пожалуй, более математическое, чем программистское, поэтому мы далее воспользуемся вторым. По типу определения требуется построить объект с установленными свойствами (массив, сумма элементов которого равна заданному числу). Такие задачи решаются двумя способами. Во-первых, можно придумать метод получения всех возможных объектов и из них выбрать те, которые удовлетворяют поставленному условию. Во-вторых, можно придумать способ получения объектов, уже удовлетворяющих поставленному условию. Обычно это делается так: строится первый объект, уже удовлетворяющий заданному условию, затем последующий строится из имеющегося. Это сложнее и кроме того, при таком подходе надо еще доказать, что в построенной последовательности все объекты будут удовлетворять условию.

Мы используем в решении задачи оба подхода, получим два решения и посмотрим на их плюсы и минусы.

Первое решение. Задача по формулировке похожа на следующую: найти среди всех N-значных чисел такие, что сумма их цифр равна заданному числу. Действительно, массив, по элементам которого раскладывается число L, можно представить себе как число по основанию L, а его элементы как цифры. Тогда идея решения выглядит так: требуется получить все N-значные числа по основанию L, а на экран вывести только те, сумма цифр которых равна L.

Для того чтобы получить, например, все десятичные числа, не превышающие заданное число, необходимо взять ноль и прибавлять к нему единицу, пока не получится самое большое число. При этом гарантированно будут получены все десятичные числа от единицы до большого числа. Очевидно, что эта процедура не зависит от основания системы счисления. Опишем ее для числа с произвольным основанием, записанным в виде массива.

Возьмем число по основанию L, запишем его в массив. Прибавим к первому разряду единицу. Возможны две ситуации. Первый разряд меньше величины основания, тогда все нормально и никаких дополнительных действий выполнять не надо. Но возможно величина первого разряда превысит величину основания, тогда необходимо этот разряд обнулить и единицу перенести в следующий, но возможно в следующем разряде также возникнет переполнение, тогда перенос нужно выполнять до тех пор, пока не будет найден разряд, такой что при добавлении к нему единицы ситуация переполнения не возникнет. Заметим, что один раз возникнет неустранимая ситуация переполнения со старшим разрядом, но это как раз и будет означать, что все числа уже получены. Запишем алгоритм прибавления единицы:

```
Прибавим единицу к первому разряду.
От первого разряда до предпоследнего делать
  Если значение текущего разряда больше основания, то
    Текущий разряд обнулить.
    К следующему разряду прибавить единицу.
Если Значение старшего больше основания, ТО его Значение равно
значению основания.
```

Замечание

Наш алгоритм несколько отличается от алгоритма сложения двух чисел. При сложении столбиком двух разрядов, если возникает переполнение, то, выполняя перенос, мы не обнуляем текущий разряд числа результата. В текущем разряде может остаться число, меньшее основания. Различие это происходит от того, что в нашем алгоритме прибавляется не произвольное число, а единица.

Построенный алгоритм верен, но это наиболее очевидное решение, и оно, как все очевидные решения, обладает недостатком. Если в какой-то момент окажется, что текущий разряд имеет значение, меньшее или равное основанию, то последующие разряды также не могут иметь значения, больше основания, и проверка от первого до последнего разряда теряет смысл. Это небольшой недостаток. С учетом того, что длина массива невелика, выигрыш в раннем завершении проверки несущественен. Но если вам нравится бороться за эффективность, то можно предложить другой алгоритм. Кстати желание придумать более быстрый алгоритм всегда похвально. Вот как может он выглядеть:

```
К первому разряду прибавить единицу.
Начиная со второго и до тех пор, пока значение разряда больше
основания на единицу, делать
  Если разряд не последний, то
    Обнулить разряд
    Следующий увеличить на единицу
  Иначе значение разряда равно значению основания.
```

Небольшая техническая проблема с увеличением текущего числа на единицу решена. Теперь можно приступить к разработке общего алгоритма. Ядро его составит цикл, внутри которого прибавлением единицы формируется новое N-значное число по основанию L. Затем алгоритм просчитает сумму цифр этого числа, и если она равна L, то массив-число распечатывается:

```
Инициализировать массив нулями (исходное число)
Пока не получено последнее число, делать
  Выполнить алгоритм прибавления единицы
  Найти сумму элементов массива
  Если сумма равна L, то распечатать массив на экран.
```

Несколько последних технических замечаний.

- Функция final выясняет, достигнуто последнее число или нет. Последнее число — это число, в котором все разряды содержат значение, равное основанию, то есть L.

- В исходной формулировке задачи говорилось о двумерной доске. Поэтому вводится сторона этой доски, затем она возводится в квадрат, таким образом, получается длина соответствующего одномерного массива.

- Процедура print используется для распечатки массива и в ней же вычисляется сумма цифр.

Листинг

```
program example;
 uses crt;
 var
  i,n,l,k:word;
  a:array[1..100] of word;
function final:boolean;
var
 i:word;
 q:boolean;
begin
 q:=false;
 for i:=1 to n do
  if a[i]<l then q:=true;
 final:=q;
end;
procedure print;
var
 i,s:integer;
begin
 s:=0;
 for i:=1 to n do s:=s+a[i];
 if s=l then
  begin
```

```
   for i:=1 to n do
    write(a[i],' ');
   writeln;
  end;
end;
begin
 clrscr;
 read(n,l);
 n:=n*n;
for i:=1 to n do a[i]:=0;
 while final do
  begin
   a[1]:=a[1]+1;
   i:=1;
   while a[i]=l+1 do
    begin
     if i<n then
      begin
       a[i]:=0;
       a[i+1]:=a[i+1]+1;
      end
     else a[i]:=l;
     i:=i+1;
    end;
   print;
  end;
end.
```

Решение, полученное ранее, имеет одно серьезное достоинство и один серьезный недостаток. Достоинство его в простоте логики, а недостаток в том, что всех возможных чисел много больше, чем нужных. Если доска велика, а колечек немного, то даже на мощном компьютере программа будет заметно "подвисать". И этот недостаток настолько серьезен, что выигранное преимущество, пожалуй, теряется. Поэтому сейчас подумаем о втором подходе. Его сущность заключается в том, чтобы из одной подходящей комбинации строить следующую подходящую, игнорируя массу ненужного. Это, конечно, существенно сложнее логически, но выигрыш в скорости будет огромный.

При таком подходе задача выглядит достаточно сложной, поэтому подумаем, как ее можно упростить, или подумаем о том, что ее усложняет. Ясно, что это задача на построение различных комбинаций из значимых чисел (то есть больших нуля), нули нас не интересуют, так как они не вносят никакого вклада в общую сумму. Но нули могут в массиве располагаться различными способами. Для наших поисков более удобно вообще-то говорить не о расположении нулей, а о расположении значимых чисел. Следовательно, комбинации будут строиться в два этапа: во-первых, необходимо построить комбинацию позиций, в которые возможно ставить значимые числа, а затем для заданной комбинации позиций необходимо построить все возможные разложения числа L. То есть задача явно разбивается на две.

Как построить комбинацию позиций? Если допустимую позицию помечать единицей, а недопустимую нулем, то комбинация позиций, это двоичное число длины N. Задача перебора всех двоичных чисел уже решалась в главе 3, в задаче о выборках, поэтому будем считать эту проблему решенной.

Как строить разложение? Исходные данные — это массив, содержащий числа, и массив-двоичное число, определяющий, в какие позиции можно записывать числа. Как именно строить очередное разложение? Можно попытаться придумать формулу, которая, исходя из уже полученного разложения, вычисляла бы числа для каждой позиции нового, а можно записать в допустимые позиции единицы, а затем добавлять по какому-либо

правилу единицы, пока не будет получена нужная сумма. Мы пойдем вторым путем, а пока сформулируем важное правило.

Правило малых изменений

Если от вас требуется построить набор чисел с заданной количественной характеристикой (например, определенная общая сумма, как в нашей задаче), то можно построить набор, в котором данная характеристика меньше, чем требуемая, и затем небольшими изменениями подгонять ее к требуемому значению.

Наиболее очевидный способ построения минимальной суммы — это простое заполнение единицами массива числа. К примеру, вот так:

B	0	1	1	0	1	1
A	0	1	1	0	1	1

Здесь массив B — это массив двоичного числа, а массив A — массив, сумму элементов которого требуется довести до L. Значение этой исходной суммы может быть больше чем L, может быть равно L, а может быть меньше L. Рассмотрим все три случая.

- Если значение суммы равно L, то мы уже получили одно из допустимых разложений, и прибавление еще одной единицы ничего не даст.

- Если значение суммы меньше L, то значит решение еще не получено, но добавлением некоторого количества единиц мы получим решение, и, возможно, вариантов решений будет много.

- Если значение суммы больше L, то решений нет, прибавление даже одной единицы только ухудшит ситуацию, эта комбинация допустимых позиций тупиковая, ее можно не рассматривать.

Примерную структуру алгоритма можно представить в виде цикла, работающего до тех пор, пока не получено последнее двоичное число, то есть число состоящее из одних единиц. Для каждого двоичного числа необходимо построить минимальное разбиение, состоящее из одних единиц, и если окажется, что это минимальное разбиение уже дает нужную сумму или нужная сумма может быть из нее получена, то выполняется работа по дополнению минимальной суммы до значения L. А теперь самый сложный вопрос. Как дополнять минимальную сумму?

Предположим, очередная комбинация построена и заполнена единицами. Предположим, что количество единиц равно H. Тогда дополнительно не хватает L–H единиц. Мы можем добавить в первый разряд 1, 2, 3, ..., L–H единиц. Предположим, мы добавили в первый разряд t1 единиц, тогда нам не хватает еще L–H–t1 единиц, которые можно восполнить во втором, третьем, ... разрядах.

Наша задача перебрать все возможные разложения. Следовательно, необходимо для первого разряда рассмотреть все возможные дополнительные суммы от 1 до L–H, а для второго — все возможные дополнительные суммы от 1 до L–H–t1, для третьего — все возможные дополнительные суммы от 1 до L–H–t1–t2.

То есть если длина стороны квадрата равна, например, 2, и соответственно длина одномерного массива представляющего этот квадрат равна 4, то полный перебор всех разложений для очередной допустимой комбинации позиций будет состоять из четырех вложенных циклов. Внешний из их имеет границы изменения параметра от 1 до 4, следующий от 1 до 3 и т. д. Это для длины стороны, равной 4. Длина стороны может быть различной, следовательно, количество вложенных циклов также должно быть различным. Построить переменное количество циклов можно с помощью механизма рекурсии. Например, так:

```
Процедура Много_Циклов(параметр)
Если параметр меньше длины массива, то
  Для всех значений от параметра до длины массива делать
    Вызывать процедуру Много_Циклов(параметр+1)
```

Это примерная конструкция, наверное, она не вполне отвечает нашим целям. Поэтому следующим шагом мы проанализируем ее, посмотрим, что в ней не отвечает поставленным целям и как ее усовершенствовать.

Во-первых, заметим, что данная конструкция просто показывает, как организовать произвольное количество вложенных циклов, здесь нет ни одной содержательной команды, ради которых собственно эти циклы и создаются. Наша же главная содержательная команда — это прибавление единицы к разряду. Добавим ее и перепишем алгоритм следующим образом:

```
Процедура Много_Циклов(параметр)
Если параметр меньше длины массива, то
  Для всех значений от параметра до длины массива делать
    К текущему разряду прибавить еденицу.
    Вызывать процедуру Много_Циклов(параметр+1)
```

Этот алгоритм выглядит уже довольно содержательно, но если вы не поленитесь и напишите по нему программу, то результат будет отрицательным. Поэтому попробуем найти расхождение между алгоритмом и изложенной ранее идеей.

В идее «сказано», что к каждому разряду может быть прибавлено много единиц. Но построенный алгоритм к каждому разряду прибавит не более одной. Действительно, в текущем вызове процедуры ее цикл пробегает массив один раз, следовательно, в текущем цикле к конкретному разряду будет прибавлено не более одной единицы, а следующий вызов начинает свою работу уже со следующего разряда. Можно решить эту проблему, передавая в следующий вызов тот же параметр, который данный вызов и получил, но тогда рекурсивные вызовы никогда не закончатся, чтобы в этом убедиться посмотрите на условие завершения рекурсии. Чтобы все-таки завершить рекурсию, мы доопределим условие выхода, указав границу увеличения для разряда. Очевидно, что рекурсивную процедуру нет смысла вызывать, если сумма чисел массива достигла L. Обозначив, для краткости длину массива через N запишем новый вариант алгоритма:

```
Процедура Много_Циклов(параметр)
Если параметр меньше N и Сумма меньше L, то
  Для всех значений от параметра до длины массива делать
    К текущему разряду прибавить еденицу
    Вызывать процедуру Много_Циклов(параметр)
```

Алгоритм стал еще более содержательным, но программа, составленная по нему, все равно не будет работать, поэтому продолжим анализ. Первая ошибка бросается в глаза сразу. Если какая-то величина используется, то ее значение должно быть определено. В нашем тексте используется переменная Сумма, а ее значение никак и нигде не изменяется. Ее изменение учесть легко, как только к разряду (любому) прибавляется 1, сумма также должна измениться на единицу. Новый вариант алгоритма:

```
Процедура Много_Циклов(параметр)
Если параметр меньше N и Сумма меньше L, то
  Для всех значений от параметра до длины массива делать
    К текущему разряду прибавить единицу
    К Сумме прибавить единицу
    Вызывать процедуру Много_Циклов(параметр)
```

Следующую довольно серьезную ошибку мы исправим исходя из свойств рекурсии, независимо от смысла задачи. Выход из рекурсивного вызова предполагает, что данные, с которыми работала процедура до того, как она вызвала сама себя, сохранились. Это не всегда так, бывает в программировании всякое, но вообще-то смысл рекурсии в том, что ее механизм берет на себя заботу о сохранности промежуточных данных. В нашем алгоритме до вызова увеличивается разряд и увеличивается значение суммы. Вернем их значения обратно, то есть после вызова уменьшим значения разряда и суммы.

И последнее, сумму можно сделать глобальной переменной, а можно и локальной. Локальные величины всегда лучше, но возиться с передачей массива не очень хочется, поэтому массив у нас будет глобальным, а переменную Сумма сделать локальной труда не составляет. Впрочем, это уже вопрос вкуса. С учетом сказанного наш алгоритм окончательно выглядит так:

```
Процедура Много_Циклов(Сумма, параметр)
Если параметр меньше N и Сумма меньше L, то
  Для всех значений от параметра до длины массива делать
    К текущему разряду прибавить единицу
    К Сумме прибавить единицу
    Если Сумма = L, то массив Числа вывести на экран
    Вызывать процедуру Много_Циклов(параметр)
    Отнять единицу от разряда
    Отнять единицу от Суммы
```

Итак, мы решили главную логическую проблему и у нас есть уже решенный вопрос о получении всех двоичных чисел. Осталось прописать общую структуру алгоритма. Подробно разбирать его не будем, это всего лишь небольшая техническая работа.

```
Заполняем нулями массив двоичного числа
Пока не получено последнее двоичное число, делать
  Получить очередное двоичное число и сразу посчитать
  минимальную сумму
  Если минимальная сумма меньше или равна L, то
    Массив Числа заполнить единицами в соответствии с массивом
    двоичного числа
    Если полученная минимальная сумма равна L, то массив Числа
    вывести на экран, так как это искомый результат
    Вызывать рекурсивную процедуру дополнения массива Числа
```

Вот собственно и все. Теперь можно написать программу.

Листинг

```
program example;
 uses crt;
 var
  b,a:array[1..100] of word;
  num,i,n,l,sum:word;
function final:boolean;
var
 i:integer;
 q:boolean;
begin
 q:=false;
 for i:=1 to n do
  if b[i]=0 then q:=true;
 final:=q;
end;
procedure Init;
var
 i:integer;
begin
 for i:=1 to n do
   if (b[i]=1) then a[i]:=1  else a[i]:=0;
end;
procedure print;
var
 i:integer;
begin
 num:=num+1;
 write(num,') ');
 for i:=1 to n do
  write(a[i],' ');
 write('     ');
 for i:=1 to n do write(b[i],' ');
 writeln;
end;
procedure Plus_1(sum,x:integer);
var
 i:integer;
begin
 if (x<=n) and (sum<l) then
   for i:=x to n do
          if b[i]=1 then
      begin
       a[i]:=a[i]+1;
       sum:=sum+1;
       if sum=l then print;
       if sum<l then  Plus_1(sum,i);
       a[i]:=a[i]-1;
       sum:=sum-1;
      end;
end;
function digit:word;
var
 i,k,m:integer;
begin
 i:=1;
 while (b[i]=1) and (i<=n) do i:=i+1;
 b[i]:=1;
 for k:=1 to i-1 do b[k]:=0;
 m:=0;
 for i:=1 to n do
  if b[i]=1 then m:=m+1;
 digit:=m;
```

```
end;
begin
 clrscr;
 read(n,l);
 n:=n*n;
 num:=0;
 for i:=1 to n do b[i]:=0;
 while final do
  begin
   sum:=digit;
   if sum<=l then
    begin
     Init;
     if sum=l then print;
     Plus_1(sum,1);
    end;
  end;
end.
```

В заключение.

Обратите внимание на уровень сложности разработанного алгоритма и написанной по нему программы. Это работает золотое правило механики. С его действием мы уже сталкивались раньше и будем говорить о нем и далее. Сущность его в том, что выигрыш в одном компенсируется проигрышем в другом. Наш выигрыш в эффективности программы потребовал существенно более сложной логики.