Поиск нерекурсивного решения на примере задачи Дейкстры

Условие задачи

Вычислить значение функции заданной следующими условиями:

1. F(1) = 1.
2. F(2N) = F(N).
3. F(2N + 1) = F(N) + F(N + 1).

При этом запрещается использовать массивы в любом виде, в том числе динамические и запрещается использование рекурсии.

Начнем наши рассуждения. Для начала заметим, что данное определение рекуррентное. Это означает, что для вычисления любого значения функции необходимо определить некоторое количество ее значений от меньших аргументов. То есть мы имеем задачу на вычисление последовательного ряда значений. Самый простой способ вычисления числовых рядов и последовательностей — это моделирование процесса расчетов. Сущность метода моделирования заключается в том, что операторами языка мы описываем процесс расчетов.

Такой метод должен опираться на какие-то особенности ряда. Например, расчет факториала можно описать так:

```
Fact=1;
For I:=2 to N do
  Fact:=Fact*I;
```

Этот фрагмент использует особенность множителей, участвующих в построении факториала, заключающуюся в том, что множители представляют собой последовательность повторяющихся чисел, отличающихся друг от друга на единицу. В нашем случае такой простой закономерности не видно. Можно было бы конечно просто запоминать все уже вычисленные значения функции от меньших аргументов, но для этого нужен массив, а массивом пользоваться запрещено. Кроме того, в момент получения аргумента не известно, какие меньшие аргументы понадобятся в процессе расчетов.

Вторая хорошая идея моделирования опирается на рекуррентный характер функции. Везде, где есть рекуррентные определения, можно строить рекурсивные программы, но рекурсия также запрещена, и поэтому метод моделирования вряд ли применим. Следовательно, нужен иной подход.

Другой подход заключается в поиске хорошей математической закономерности, которая позволила бы упростить проблему расчета. Закономерность можно попытаться найти анализируя свойства функции, но для этого скорее всего потребуется хороший математический аппарат. А можно просчитать некоторое количество числовых примеров, сопоставить результаты и может быть закономерность удаться увидеть. Конечно, закономерность, полученную таким образом, еще придется доказывать (примеры ничего не доказывают), но если закономерность окажется верной для значительного числа примеров, то вероятность ее истинности будет высока. В общем надо придумать метод счета вручную и просчитать несколько хороших примеров.

Что такое хороший пример? Хороший пример — это серьезная проблема. С одной стороны, хороший пример должен продемонстрировать, как можно больше особенностей

исследуемого процесса, а следовательно, он должен быть сложным. Но с другой стороны, если пример сложен, то возникает вероятность ошибки, и чем пример сложнее, тем эта вероятность выше. Если же пример прост, то вероятность ошибки невелика или вообще равна нулю, но такой пример может быть очень неинформативен и тогда в нем мало смысла. Сформулируем правило построения хорошего примера.

Правило построения хорошего примера

Для того чтобы получить хороший пример, посмотрим, какие особенности исследуемого вычислительного процесса необходимо продемонстрировать. Сложность и трудоемкость примера должны быть таковы, чтобы исследуемые особенности были представлены полно, но не более того.

Покажем на нашей задаче, как применить полученное ранее правило. Функция определяется тремя правилами, каждое из которых вносит какие-то нюансы в вычислительный процесс. Значит, нужен пример, в котором каждое правило применялось бы хотя бы 2–3 раза. Конечно, больше было бы лучше, но с ростом аргумента будет нарастать трудоемкость вычислений, а значит расти и вероятность ошибки. Далее мы поищем закономерность на одном примере, что, конечно, неправильно. Такое исследование требует большого количества расчетов, но мы все же удовольствуемся одним, чтобы не загромождать текст.

Говоря о правиле расчетов, заметим, что на каждом шагу счета функция может быть представлена либо одной функцией, либо двумя. Это наводит на мысль представить процесс расчетов, как дерево уменьшающихся аргументов. Вычислим функцию от 113

Таблица Расчеты функции от аргумента 113.

<table>
<tr><td colspan="14">113</td></tr>
<tr><td colspan="3">56</td><td colspan="11">57</td></tr>
<tr><td colspan="3">28</td><td colspan="3">28</td><td colspan="8">29</td></tr>
<tr><td colspan="3">14</td><td colspan="3">14</td><td colspan="3">14</td><td colspan="5">15</td></tr>
<tr><td colspan="3">7</td><td colspan="3">7</td><td colspan="3">7</td><td colspan="4">7</td><td>8</td></tr>
<tr><td colspan="2">3</td><td>4</td><td colspan="2">3</td><td>4</td><td colspan="2">3</td><td>4</td><td colspan="3">3</td><td>4</td><td>4</td></tr>
<tr><td>2</td><td>1</td><td>2</td><td>2</td><td>1</td><td>2</td><td>2</td><td>1</td><td>2</td><td>2</td><td>1</td><td>2</td><td>2</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

Результат расчетов F(113) = 13.

В этой таблице уже кое-что видно. Во-первых, видно, что на каждом шагу вычислений мы имеем только два аргумента. Во-вторых, видно, что один из них обязательно четный, а второй обязательно нечетный. Появляется идея. Результат расчетов очевидно равен количеству единичных аргументов. Количество аргументов на каждом шагу — это сумма количеств четных и нечетных аргументов. Следовательно, необходимо

поискать закономерность между количествами четных и нечетных. Заметьте, мы смогли увидеть идею только потому, что данные были представлены наглядным образом. Сформулируем правило наглядности.

Правило наглядности

Для того чтобы получить хорошую информацию из результатов численного эксперимента, данные нужно так расположить на бумаге, чтобы взаимосвязи между ними были, как можно более наглядны.

Нарушение данного правила может свести на нет вычислительную работу. Например, вычислим значение функции немного иным способом. По определению. Посчитаем значение функции от аргумента 113.

F(113) = F(56) + F(57) = F(28) + F(28) + F(29) = 2F(28) + F(29) = 2F(14) + F(14) + F(15) = 3F(14) + F(15) = 3F(7) + F(7) + F(8) = 4F(7) + F(8) = 4F(3) + 4F(4) + F(4) = 4F(3) + 5F(4) = 4F(1) + 4F(2) + 5F(2) = 4F(1) + 9F(2) = 4F(1) + 9F(1) = 13F(1) = 13

Может быть, вам эта запись и не покажется слишком плохой, но согласитесь она все же намного проигрывает в восприятии в сравнении с таблицей первого метода. Вернемся к таблице и применим полученное правило еще раз. Распишем количества четных и нечетных на каждом шаге.

Таблица Таблица четных и нечетных количеств.

Шаг	Четных	Нечетных
1	1	1
2	2	1
3	3	1
4	1	4
5	5	4
6	9	4
7	0	13

Последний шаг как-то выпадает, здесь появляется ноль, но это последний шаг, он вполне может выпадать из общей схемы расчетов, потому что расчеты уже просто закончены, а вот предыдущие результаты очень любопытны. Обратите внимание, что на каждом шагу, начиная со второго, одно из количеств равно сумме двух верхних, а второе количество равно одному из верхних. Одна из величин равна сумме двух величин предыдущего шага, но иногда равна сумме четных, а иногда сумме нечетных.

Для того чтобы разобраться в том, как именно образуются новые количества четных и нечетных, посмотрим, что происходит на очередном шаге. Пусть на очередном шаге есть два числа ЧЕТНОЕ и НЕЧЕТНОЕ. НЕЧЕТНОЕ всегда распадается на два аргумента один четный, второй нечетный. Следовательно, нечетное число ничего не изменяет в

текущей ситуации. ЧЕТНОЕ уменьшается в два раза. При этом результат деления может стать как четным, так и нечетным. Видимо здесь и зарыта причина возможных изменений. Рассмотрим эти два варианта развития событий.

❑ ЧЕТНОЕ опять дает ЧЕТНОЕ. В этом случае общее количество четных увеличивается на количество нечетных, нечетных остается столько, сколько и было. Следовательно, будут верны формулы:

- Кол-во НЕЧЕТНЫХ = Кол-ву НЕЧЕТНЫХ;
- Кол-во ЧЕТНЫХ = Кол-во ЧЕТНЫХ + Кол-во НЕЧЕТНЫХ.

❑ ЧЕТНОЕ дает НЕЧЕТНОЕ. В этом случае общее количество нечетных увеличивается на количество четных, четных становится столько, сколько было нечетных. Следовательно, будут верны формулы:

- Кол-во НЕЧЕТНЫХ = Кол-во НЕЧЕТНЫХ + Кол-во ЧЕТНЫХ
- Кол-во ЧЕТНЫХ = Кол-ву НЕЧЕТНЫХ

В записанных формулах, конечно же, справа от знака равенства стоят количества текущего (предыдущего) шага, а слева количества следующего (текущего) шага.

Вот в общем-то и все искомые закономерности. Осталось обсудить идею алгоритма и записать сам алгоритм. А идея алгоритма такова: на каждом шаге расчетов мы имеем четыре числа ЧЕТНОЕ, НЕЧЕТНОЕ, и два количества: количество ЧЕТНЫХ и количество НЕЧЕТНЫХ. Шаг работы алгоритма заключается в вычислении по имеющимся четырем числам четырех новых и так до тех пор, пока очередное нечетное число не окажется равно 1. Когда это случится, мы сложим количество четных с количеством нечетных и полученное значение будет искомым ответом.

В своей идее, мы исходим из того, что на каждом шаге есть два числа: ЧЕТНОЕ и НЕЧЕТНОЕ. Однако есть один случай выбивающийся из данного правила. Это первый шаг расчетов, если в качестве исходного аргумента взято четное число. Можно конечно попытаться так изменить идею, чтобы этот случай в нее укладывался, но поступим проще. Воспользуемся тем фактом, что в случае с четным аргументом расчет функции не разветвляется. Поэтому не случится ничего страшного, если алгоритм, получив аргумент, будет делить его до тех пор, пока аргумент не станет нечетным, после чего основная идея станет достаточной. Осталось записать алгоритм и текст программы.

```
Вводим Аргумент
Пока Аргумент четный делать
  Аргумент = Аргумент / 2
НЕЧЕТНОЕ = Аргумент
Четная сумма =1
Нечетная сумма =1
Пока НЕЧЕТНОЕ больше 1 делать
  Вычислить новое четное и новое нечетное
  Если четное деленное пополам даст четное
    То применить формулы (1)
```

```
    Иначе применить формулы (2)
Вывести результат
```

Листинг

```
program example;
 uses crt;
 var
  c,chet,nechet,sum_chet,sum_nechet:word;
begin
 clrscr;
 readln(nechet);
 while nechet mod 2=0 do nechet:=nechet div 2;
 sum_chet:=1;
 sum_nechet:=1;
 repeat
  chet:=nechet div 2;
  nechet:=nechet - chet;
  if chet mod 2 = 1 then
   begin
    c:=chet;
    chet:=nechet;
    nechet:=c;
   end;
  if ((chet div 2) mod 2)=0 then
    sum_chet:=sum_chet+sum_nechet
  else
   begin
    c:=sum_nechet;
    sum_nechet:=sum_nechet+sum_chet;
    sum_chet:=c;
   end;
  writeln(chet,'  ',sum_chet,'  ',nechet,'  ',sum_nechet);
 until nechet=1;
 write(sum_nechet);
end.
```

Решенная задача хорошо иллюстрирует необходимость тщательного математического анализа. В задачах такого типа, а они встречаются очень часто, знание языка не дает почти ничего, чистое же умение алгоритмизации дает немного. Во главу угла здесь встал поиск математической закономерности.

Еще раз повторим, что было существенно важно в поисках закономерности.

- Исходная информация получена из примеров. Логика утверждает, что примеры ничего не доказывают, но примеры могут дать хорошую зацепку для дальнейшего анализа. В тексте дан только один удачный пример. Конечно, трудно ожидать, что первый попавшийся пример окажется хорошим, поэтому надо быть готовым прорешать их значительное множество.

- Необходимо придумать наглядное представление результатов расчетов, иначе вся масса практической работы может оказаться просто бесполезной.

Между прочим, если разрешить использование рекурсии, то задача становится элементарной и фактически сводится к записи ее рекуррентного определения .

Листинг

```
program example;
 uses crt;
 var
  n:integer;
function Rec(n:integer):integer;
begin
 if n=1 then Rec:=1
  else
   if n mod 2=0 then Rec:=Rec(n div 2)
    else Rec:=Rec(n div 2)+Rec((n div 2)+1);
end;
begin
 clrscr;
 read(n);
 write(Rec(n));
end.
```

В заключение.

Конечно, второе решение несравнимо проще, но так бывает не всегда и необходимо уметь искать математические закономерности. К тому надо заметить, что если нечетных чисел в получаемых разложениях окажется много, то вычислительный процесс начнет стремительно разветвляться, требуя большого объема стековой памяти, и не исключена ситуация, когда ресурсов памяти просто не хватит. Поэтому еще раз повторимся, умение поиска математических закономерностей — это исключительно важное умение.