Информатика

Грацианова Татьяна Юрьевна
Преподаватель подготовительных курсов факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова.

Эта страшная рекурсия: рекурсивные алгоритмы

В прошлом году мы с вами побывали на занятиях кружка по информатике, посвящённом рекурсии (см. «Потенциал» № 4 за 2013 г.). Там обсуждалось, что такое рекурсия, как она используется в математике, информатике и даже в искусстве. Сегодня речь пойдёт об использовании рекурсии в программировании.

– Начнём с определения. Вы уже знаете, что в общем случае рекурсивное определение – это определение, которое ссылается само на себя. То же можно сказать и о рекурсивных функциях и процедурах. Описание функции (процедуры) называется рекурсивным, если в описании встречается обращение к самой этой функции (процедуре). Бывает и косвенная рекурсия: в описании функции FA есть обращение к функции FB, а функция FB обращается к FA.

Во многих языках программирования можно писать рекурсивные процедуры и функции. Мы с вами сегодня будем писать фрагменты программ на языке Паскаль; те, кто больше любит другие языки, легко перепишут их на Си или Visual Basic – алгоритмы везде одинаковы.

Вычисление факториала

В прошлый раз мы установили, что факториал числа n ($n>0$) можно определить двумя способами: нерекурсивно

$n! = 1*2*3*...*(n-1)*n$

и рекурсивно

$F(1) = 1$,

$F(n) = F(n-1) * n$ при $n > 1$.

Следуя нерекурсивному определению, получаем описание функции с циклом (перемножаем числа от 1 до N). А теперь посмотрим на рекурсивное определение. Оно состоит из двух частей. Чтобы определить, какую часть определения использовать, надо посмотреть значение N. При N=1 используется первая часть, при N>1 – вторая. Именно это аккуратно и запишем на Паскале. Получаем вот такие описания функций:

Нерекурсивно	Рекурсивно

Function
  Factor(N:Integer) : Integer;
Var I, F : Integer;
Begin F:=1;
  For I:=2 to N do
  F:=F*I;
  Factor:=F
End;
	Function
	  FactorR (N:Integer) : Integer;
	
	Begin
	If N=1 Then FactorR:=1
	  Else FactorR:=
	  FactorR (N-1)*N
	End;

Как видите, по размеру программы примерно одинаковые – несколько строчек. Результаты тоже получаются одинаковые – можете проверить (напомним, чтобы проверить, как работает функция, надо написать основную программу, в которой обратиться к этой функции с каким-либо значением аргумента и напечатать ответ). Как вы считаете, какое описание лучше?

– Мне больше рекурсивное нравится – оно в точности определению соответствует.

– Верно. Рекурсивные описания функций часто бывают более наглядными и короткими, чем их нерекурсивные аналоги. Но давайте посмотрим, как работают эти функции, как используют память. В нерекурсивной функции используются два имени в заголовке (аргумент и имя функции) и описаны две локальные переменные. Вот с этими 4 значениями она и работает, для их хранения ей нужна память во время её работы.

В рекурсивной функции локальных переменных нет. Означает ли это, что ей нужно меньше памяти? Давайте разберём её работу для N = 4.

Аргумент не равен 1, поэтому должен быть выполнен оператор присваивания

```
FactorR:= FactorR (N-1)*N.
```

Вспомним, как выполняется оператор присваивания. Сначала вычисляется значение выражения, которое стоит в правой части, то есть надо подсчитать FactorR (N–1)*N, а для этого надо обратиться к функции. Но у функции уже будет другое значение аргумента, вместо N он будет равен N–1. Можно сказать, что выполняется присваивание N:=N–1. Но, если такое присваивание выполнить, прежнее значение N потеряется, «забудется». А оно ведь нам понадобится! Значит, его надо сохранить. Примерно так должен «рассуждать» компьютер (точнее, программисты, которые его проектировали). Отведём в памяти место, где будем сохранять такие «рабочие» значения и запишем туда значение N, число 4.

Теперь выполняем нужные нам действия, обращаемся к функции FactorR с параметром 3. Опять дол-

жен работать оператор присваивания FactorR:=FactorR(N–1)*N, только теперь N=3. И опять это значение придётся сохранить в некотором специально отведённом месте (теперь у нас там хранится последовательность 4, 3). Далее можем видеть, что к этой последовательности добавится число 2 (получается 4, 3, 2). При этом происходит обращение к функции с аргументом 1 и – ура! – работает первая ветвь, получаем, что FactorR:=1. Но это ещё не ответ, ведь мы только подсчитываем значение выражения в операторе присваивания `FactorR:= FactorR (1)*N`, а значение N – то, которое с «прошлого раза» осталось (мы его запоминали самым последним), достанем из нашего хранилища. Это число 2. Выполняем умножение, получаем FactorR=2. Но и это ещё не ответ, это значение надо тоже подставить в выражение в правой части оператора присваивания `FactorR:= 2*N`, а N опять «старое», которое запоминали. Вытаскиваем его, получаем произведение 2*3, которое тоже надо подставить (уже в последний раз) в выражение в операторе присваивания `FactorR:= 6*N`, а N=4 опять вытащить из хранилища. При N=4 мы запоминали 3 значения переменной, для, например, N=6 пришлось бы запоминать 5 значений. Таким образом, при работе обычной функции объём необходимой памяти зависит только от количества переменных, описанных в ней; при работе рекурсивной функции объём памяти зависит ещё и от количества рекурсивных вызовов (глубины рекурсии). Обычно для работы рекурсивной функции памяти требуется больше, чем для работы нерекурсивного аналога. Да и времени для рекурсивного алгоритма обычно требуется больше, ведь оно тратится ещё и на работу с хранилищем.

– А если бы мы произведение наоборот переписали:

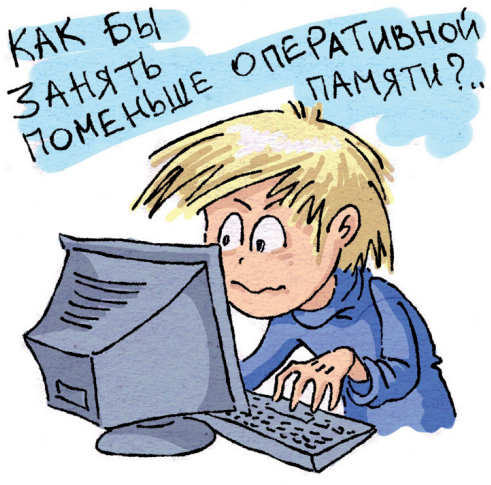
`FactorR:= N*FactorR (N-1)` –

тогда не пришлось бы значение N запоминать?

– Нет, оно всё равно понадобилось бы – умножение возможно, только когда известны оба сомножителя. К тому же мы здесь достаточно условно разобрали, что запоминает компьютер при работе с рекурсивными функциями. На самом деле он не «смотрит», что понадобится, а что нет (это слишком сложно), а запоминает значения всех переменных, да и не только их – он ведь ещё должен знать, куда вернуться, что делать после вычисления очередного шага рекурсии, поэтому запоминается довольно много информации

Давайте посмотрим, в каком порядке компьютер записывает и считывает информацию из хранилища: элемент, записанный (положенный на хранение) последним, будет прочитан (вынут из хранилища) первым, и наоборот – то, что записали в самом начале (в нашем случае это значение N, с которым первоначально обратились к функции), будет прочитано в последнюю очередь.

Такой тип хранилища информации называется **стеком**. В литературе можно также встретить название



FILO – от первых букв слов во фразе First In Last Out (первым пришёл – последним ушёл). Вы можете в нашем случае представить себе стек как колодец, в который мы на верёвке опускаем числа. Сначала привязали к верёвке четвёрку, немного опустили, привязали тройку, ещё немного опустили – и так далее. А когда дошли до единицы, начинаем наши числа из колодца вынимать – тянем за верёвку. Вынимаем сначала двойку, потом тройку, а четвёрку, которую привязали первой, вытащим в самом конце. Чем больше рекурсивных вызовов, тем длиннее верёвка понадобится, тем глубже должен быть колодец (помните, мы уже использовали термин «глубина рекурсии»). Такой способ хранения и извлечения информации оказывается удобен во многих случаях, не только при работе рекурсивных алгоритмов, поэтому он часто используется в информатике.

Вычисление числа Фибоначчи

– Вернёмся к рекурсивным функциям. Давайте напишем функцию, к рекурсивному определению которой мы привыкли (а нерекурсивным обычно никто и не пользуется). Это вычисление числа Фибоначчи (n>=1):

$F(1)=1$, $F(2)=1$,

$F(n)=F(n-1)+F(n-2)$ для $n>2$.

Также сделаем это рекурсивно и нерекурсивно. Рекурсивное описание здесь выглядит тоже гораздо проще и понятнее – в точности повторяет определение, в нерекурсивном придётся немножко подумать, как заменить рекурсию циклом, переприсваивая на каждом шаге значения переменных.

Нерекурсивно

```
Function Fib(N:Integer):Integer;
Var A,B,F, I:Integer;
Begin A:=1; B:=1;  F:=1;
      For I:=3 to N do
      Begin F:=A+B;
            A:=B;
            B:=F
      End;
      Fib:=F
End;
```

Рекурсивно

```
Function FibR(N:Integer):Integer;

Begin
  If N<=2 Then FibR:=1
          Else
          FibR:=FibR(N-1)+FibR(N-2)
End;
```

Рекурсивное определение получилось гораздо короче. Но давайте посмотрим, как работают эти функции. Нерекурсивная производит вычисления так же, как это делаем мы, когда нам надо посчитать N-е число Фибоначчи: принимает первое и второе равными единице, а далее вычисляет каждое следующее как сумму двух предыдущих, пока не дойдёт до нужного. Рекурсивная функция берётся за дело с другого конца. Чтобы вычислить, например, пятое число последовательности, она выполняет оператор присваивания FibR:=FibR(4)+FibR(3). А в этом операторе два рекурсивных вызова:

FibR:=FibR(3)+FibR(2)

и

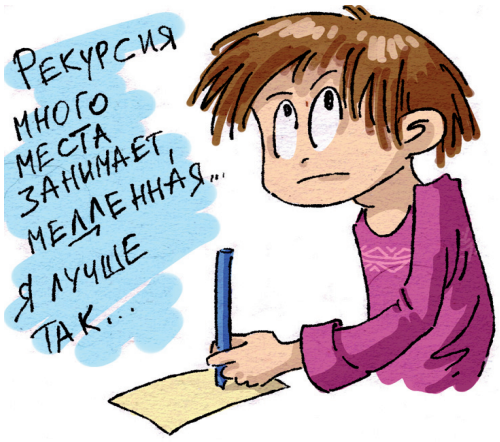
FibR:=FibR(2)+FibR(1).

Причём, обратите внимание, для вычисления Fib(4) надо знать Fib(3), которая вычисляется во втором слагаемом. Однако функция эти вычисления производит ещё раз. Если вы возьмёте число побольше и распишете, как здесь работает рекурсия, вы увидите, как лавинообразно возрастает количество рекурсивных

вызовов, сколько раз вычисления дублируются. Поэтому рекурсивное описание функции вычисления чисел Фибоначчи обычно приводят как пример нерационального. То, что рекурсивная функция работает гораздо медленнее, можно заметить даже на наших быстрых компьютерах. Если вы подсчитаете достаточно большое число Фибоначчи (например, 40-е), то увидите, что нерекурсивная функция выдаёт ответ мгновенно, а при работе рекурсивной приходится некоторое время подождать. (Заметим: для того чтобы вычислить такое большое число, придётся значение функции определить как LongInt).

– А зачем же тогда нужны рекурсивные функции, если они и памяти больше требуют, и работают медленнее? Неужели нельзя без них обойтись?

– Обойтись можно. Но иногда рекурсивные алгоритмы бывают настолько удобны, что можно пренебречь их недостатками; мы ведь только начали знакомство с рекурсией.



Вывод последовательности

Давайте подумаем, как будет работать вот такая процедура:

```
Procedure Print;
var a : integer;
Begin read(a);
    If a=0 then Writeln
           else Begin
                     Writeln(a);
                     Print
                End
End;
```

– Вводится целое число. Если это ноль, то просто переводится строка и работа завершается. А если не ноль, то это число печатается и происходит обращение снова к процедуре, то есть опять вводится число и печатается – и так, пока не будет введён ноль.

– Верно. Только обратите внимание: вводятся числа с помощью оператора READ, а выводятся с помощью WRITELN. Это означает, что ввести числа надо все сразу, в строчку через пробел, а выводиться они будут в столбик (то есть вводимые числа не перемешаются с выводимыми).

– Ну и что в этом великого? Мы такое писали на первом же уроке, когда циклы проходили. Подумаешь: ввели – вывели, и всё это пока ноль не ввели.

– Верно, здесь ничего примечательного нет. Но давайте теперь попробуем два рядом стоящих оператора Writeln(a) и Print поменять местами. Как будет работать такая процедура? Давайте исследуем это на компьютере, добавим к описанию

процедуры основную программу с вызовом процедуры, введём числа (не забудьте – несколько чисел в строчку через пробел, а в конце 0).

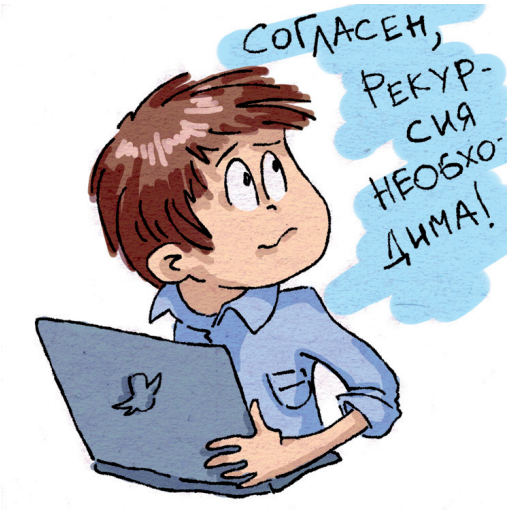
```
Program RecProc;
  Procedure Print;
    var a : integer;
  Begin read(a);
     If a=0 then Writeln
            else Begin
                    Print
                  Writeln(a);
                End
   End;
Begin {Основная программа}
   Writeln('Введите строку из нескольких целых чисел через пробел. ');
   Writeln('В конце поставьте 0');
   Print
End.
```

– Ой, как интересно! Она те же числа так же в столбик печатает, только в обратном порядке!!!

– Видите, такую задачу уже просто так нерекурсивно не сделаешь. Мы с вами печатали числа в обратном порядке без использования рекурсии: для этого приходилось заводить массив, записывать в него числа, а потом печатать. Но ведь для работы с массивом мы должны знать, сколько чисел будет введено (объявим массив больше, чем нужно, – много памяти понапрасну истратим, объявим меньше – не поместятся в него все числа). К тому же объявленный массив будет занимать место в памяти, даже когда мы числа из него напечатали, и он уже не нужен. Здесь же таких проблем нет, так как массив нам не понадобился.

– И что же, нерекурсивно без массива такого не сделать?

– Сделать можно, но сложно, и все равно придётся использовать какое-либо хранилище для чисел, если не массив, то файл или какую-либо иную структуру. С рекурсивной процедурой получается проще, потому что она многие заботы берёт на себя. Давайте разберёмся, как она работает.



Вводится число. Если оно равно нулю – всё так же, как и в нерекурсивном варианте. А если не равно, происходит обращение к процедуре и, следовательно, опять вводится число. Но вспомним, мы говорили, что при рекурсивном обращении происходит запоминание информации в стеке. Таким образом, введённое число попадает в стек. Вводится следующее число и тоже попадает в стек – и так до тех пор, пока не будет введён ноль. Рекурсивные вызовы закончились, при окончании работы очередной вызванной процедуры одно число вынимается из стека.

Причём мы помним, сначала будет вынуто то число, которое положили последним, то есть числа будут выниматься из стека в обратном порядке. Вот и получается то, что мы видим. Чтобы написать такую процедуру без рекурсии (и без массива), придётся самостоятельно определять некоторое хранилище данных (обычно каким-либо образом моделируют стек), писать процедуры для работы с ним. Рекурсивная процедура всё это выполняет автоматически.

Ханойская башня

Конечно, приведённая здесь задачка слишком проста и не может показать всю важность рекурсии. Давайте рассмотрим ещё один пример. Вспомним задачу о «Ханойской башне». Не будем сейчас говорить о том, кто её придумал и почему она так называется (кому интересно, посмотрите в Википедии), сформулируем условие.

Задача. Имеется три стержня, на одном из них нанизаны колечки разного размера: большие внизу, маленькие сверху (как в детской пирамидке). Надо перенести за наименьшее число ходов всю пирамидку на другой стержень. За один ход разрешается перекладывать только одно кольцо (естественно, надо брать с любого стержня самое верхнее кольцо), причём при любом перекладывании верхнее кольцо должно быть меньше нижнего.

Составим рекурсивный алгоритм. База рекурсии: пусть пирамидка из двух колец на стержне 1, надо перенести её на стержень 2. Верхнее кольцо переносим на стержень 3, нижнее – на 2. Остаётся маленькое кольцо поставить на место. Рекурсивный шаг: пусть на стержне 1 пирамидка из N колец, и её надо перенести на стержень 2. Переносим пирамидку из N–1 кольца на стержень 3, нижнее (самое большое кольцо) освободилось, переносим его на стержень 2, и теперь пирамидку из N–1 колец тоже переносим на 2. Давайте оформим это в виде программы.

– А как же у нас компьютер будет кольца переносить? Мы ему руки приделаем?

– Увы, приделать нашему компьютеру руки мы не можем, руками будем действовать сами. Напишем процедуру, которая будет печатать инструкцию, как переставлять кольца. Входными данными для неё будет число колец, номер стержня, на котором находится пирамидка в начале, и номер стержня, на который надо её переставить. Давайте в процессе решения подсчитывать количество операций. Инструкцию будем выписывать в виде номера действия, номера стержня, с которого надо взять верхнее кольцо, и номера стержня, на который надо положить. Какое колечко взять со стержня, указывать не надо, так как можно брать только самое верхнее.

Возникает задача, совершенно не связанная с рекурсией: как узнать номер дополнительного стержня?

Допустим, в начале пирамидка на стержне № 1, переложить её надо на № 2. В этом случае дополнительным является № 3. Но существует еще 5 различных вариантов начального и конечного размещения пирамидки. Чтобы их все не расписывать по отдельности, придётся придумать формулу, по которой можно получить номер дополнительного стержня, зная номера начального и конечного. Если стержни пронумеровать 1, 2 и 3, сумма номеров будет 6, таким образом, чтобы узнать номер дополнительного, надо от 6 отнять номера двух других.

– А зачем так мучиться? Давайте объявим, что всегда будем перекладывать с 1-го на 2-й, – и ничего считать не надо.

– Без этого у нас рекурсия не получится. В каждом рекурсивном вызове стержни «меняются местами». Если нам изначально надо пирамидку из K колец переложить с 1-го на 2-й, то, значит, пирамидку из K–1 кольца надо переложить сначала с 1 на 3-й, а потом с 3-го на 2-й.

Мы с вами долго разговариваем, а программа получается несложная: то, что мы проговорили, надо написать на Паскале:

```
Program Hanoy_B;
  const Z=' Переложите колечко с '; {глобальные переменные и константы}
Var Kp:Integer;{количество перекладываний}
Procedure Hanoy(K,N1,N2:Integer); {Описание рекурсивной процедуры}
Var Nd:Integer; {Локальная переменная, номер дополнительного стержня}
Begin  Nd:=6-N1-N2;
 If K=1 Then Writeln(Z,N1, ' на ', N2) {база рекурсии}
        Else {база рекурсии}
            If K=2 Then Begin kp:=Kp+1; Writeln(Kp,Z, N1,' на ',Nd);
                              kp:=Kp+1; Writeln(Kp,Z, N1,' на ',N2);
                              kp:=Kp+1; Writeln(Kp,Z, Nd,' на ',N2)
                        End
                   Else Begin {рекурсивный шаг}
                              Hanoy(K-1,n1,nd);
                              kp:=Kp+1; Writeln(Kp, Z,N1,' на ',N2);
                              {kp - номер операции}
                              Hanoy(K-1,Nd,N2);
                         End
End;
 {Основная программа}
 Var K,N1,N2: Integer;
Begin
 Write('Сколько колечек в башне '); Readln(K);
 Write('Номер стержня, на котором колечки '); Readln(N1);
 Write('Номер стержня, на который надо переставить '); Readln(N2);
 Kp:=0; {Глобальная переменная для подсчета количества операций}
 Hanoy(K,N1,N2)
End.
```

Вот такое несложное рекурсивное описание. Проверьте: следуя этой инструкции, вы сможете переставить с одного стержня на другой пирамидку из любого разумного количества колечек (если количество «неразумное», слишком большое, программа может работать неправильно, но не потому, что в ней ошибка, а потому, что не хватит памяти в стеке). Дело в том, что для перекладывания пирамидки из 3 колечек понадобится 7 операций, из 4 – 15. Улавливаете зависимость? Для перекладывания пирамидки из

N колечек надо сделать 2^N-1 операций. Мы этот результат получили экспериментально, давайте попробуем его вывести, наш рекурсивный алгоритм в этом нам поможет.

На прошлом занятии мы говорили, что рекурсивные алгоритмы перекликаются с доказательствами методом математической индукции. Вспомним, что это за метод. С его помощью можно доказать истинность некоторого утверждения для всех натуральных чисел. Сначала надо проверить истинность утверждения для какого-то одного числа K_0 (чаще единицы, иногда для двойки, бывает, что и для какого-то другого числа) – это база индукции. Далее предполагаем, что утверждение верно для некоторого $K > K_0$, и доказываем, что из этого следует истинность утверждения для числа K+1 – это шаг индукции, индукционный переход. Если эти два условия удалось выполнить, можно считать доказанным, что утверждение верно для любого натурального $K \geq K_0$.

Вот и попробуем воспользоваться этим методом на основе нашего алгоритма.

База индукции. Для перекладывания пирамидки из двух колечек надо сделать 3 операции – мы в том убедились, $3=2^2-1$, то есть для N=2 утверждение верно.

Индукционный переход. Пусть верно, что для пирамидки из N колечек надо сделать 2^N-1 операций. Надо доказать, что из этого следует, что для пирамидки из N+1 колечка понадобится $2^{N+1}-1$ операций.

Возьмём пирамидку из N+1 колечка. Следуя алгоритму, надо сначала N верхних колечек переложить на дополнительный стержень. Для этого нам понадобится (исходя из предположения) 2^N-1 операций. Ещё одна операция – перекладываем самое большое кольцо на тот стержень, где должна оказаться пирамида в результате. И ещё 2^N-1 операций – перекладываем пирамидку из N–1 кольца с дополнительного стержня на первое кольцо. Получаем

$$2^N-1+1+2^N-1=2*2^N-1=2^{N+1}-1,$$

что и требовалось доказать.

Таким образом, используя рекурсивный алгоритм, мы не только смогли составить программу с инструкций, но и доказали утверждение. А попробуйте написать программу без использования рекурсии! Она получится очень сложной, потому что нерекурсивный алгоритм работы с Ханойскими башнями достаточно нетривиален.

Как видите, нам удалось привести несколько примеров «полезной рекурсии». Конечно, и эти задачи не имеют особого практического значения (трудно представить, чтобы для решения какой-то серьезной задачи понадобилось переставлять пирамидки), и пока вам придётся просто поверить, что бывают случаи, когда рекурсивные алгоритмы (и программы) имеют право на существование. Мы ещё не «дозрели» до серьёзных рекурсивных алгоритмов, сначала надо научиться писать простые программы. Этим мы и займёмся в следующий раз.

Калейдоскоп Калейдоскоп Калейдоскоп

Неожиданные определения

Учёный – это человек, который в чём-то почти уверен.

Ж. Ренар

Учёность – сладкий плод горького корня.

Д. Катан

Прошлое – лучший пророк для будущего.

Дж. Байрон