МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

Юго-Западный государственный университет

(ЮЗГУ)

Кафедра вычислительной техники

ПРОГРАММИРОВАНИЕ ПОДПРОГРАММ. РЕКУРСИЯ

Методические указания по выполнению лабораторной работы

для студентов направления подготовки 09.03.01

Курск 2016

УДК 621.3
Составитель: Э.И. Ватутин

Рецензент
Кандидат технических наук, доцент *В.С. Панищев*

Программирование подпрограмм. Рекурсия: методические указания по выполнению лабораторных работ по дисциплине «Программирование» / Юго-Зап. гос. ун-т; сост.: Э.И. Ватутин; Курск, 2016. 26 с.: ил. 3.

Методические рекомендации содержат сведения по разработке подпрограмм на современных языках программирования высокого уровня.

Предназначены для студентов направления подготовки 09.03.01 «Информатика и вычислительная техника».

Текст печатается в авторской редакции

Подписано в печать _____. Формат 60x84 1/16.
Усл. печ. л. Уч. – изд.л. Тираж 30 экз. Заказ . Бесплатно.
Юго-Западный государственный университет
305040, Курск, ул. 50 лет Октября, 94.

Содержание

Подпрограммы

Целью работы является получение практических навыков при разработке подпрограмм и программировании рекурсии.

В Delphi существует два типа подпрограмм: процедуры и функции. Процедуры выполняют определенный набор действий, функции отличаются от них наличием возвращаемого значения – результата функции (по аналогии с понятием функций в математике). Общение подпрограмм с внешним миром осуществляется через механизм передачи параметров, благодаря которому при запуске подпрограмме известны исходные данные, с которыми ей предстоит работать, а после окончания ее работы сформированы соответствующие результирующие значения. Разделение подпрограмм на процедуры и функции является в достаточной степени условным, т.к. они отличаются незначительными отличиями в оформлении в программе и являются взаимозаменяемыми: любую процедуру синтаксически можно представить как функцию и наоборот с большей или меньшей степенью удобства. В некоторых языках программирования нет функций (например, в языке ассемблера есть аппаратная поддержка подпрограмм, которые скорее напоминают процедуры), в других – нет процедур (например, в языке Си и производных от него есть понятие функции, которая ничего не возвращает, что на самом деле очень похоже на процедуру).

Детали реализации подпрограммы скрыты в ее теле, образованном последовательностью выполняемых операторов, при этом операторы могут работать как с глобальными объектами программы (переменными, константами и пр.), так и иметь в своем распоряжении локальный набор объявлений, недоступный извне, что решает проблему возможного совпадения имен переменных в различных частях программы и минимизирует возможность возникновения связанных с этим потенциальных ошибок.

При оптимизации программного кода локальные переменные простых типов с большой долей вероятности будут расположены в регистрах процессора, что снижает время выполнения программы, т.к. при этом требуется меньшее количество обращений к сравнительно медленной оперативной памяти. Статические локальные переменные структурных типов (массивы, записи) размещаются в специальной области оперативной памяти, именуемой стеком.

На подпрограммы распространяются те же ограничения, что и на все остальные элементы программы: подпрограмму сперва требуется объявить, а лишь потом использовать (это правило можно частично обойти с использованием директивы `forward`). Удобным способом группировки подпрограмм по смысловому назначению является их объединение в модули.

Процедуры и функции объявляются в программе следующим образом:

```
procedure Имя_процедуры (Список_параметров); директивы;

    Область_локальных_объявлений

begin
  Тело_процедуры (Операторы);
end;

function Имя_процедуры (Список_параметров): Тип_результата; директивы;

    Область_локальных_объявлений

begin
  Тело_функции (Операторы);
end;
```

Как уже было отмечено выше, функции отличаются от процедур наличием возвращаемого значения (результата функции), тип которого задается после двоеточия, завершающего список параметров в круглых скобках (при его наличии).

В качестве примера функции можно рассмотреть функцию `sinh` из модуля `Math.pas`, которая производит вычисление гиперболического синуса:

```
function SinH(const X: Extended): Extended;
begin
```

```
  Result := (Exp(X) - Exp(-X)) / 2;
end;
```

Имя функции `SinH` задано непосредственно после ключевого слова `function`, далее в скобках указан единственный параметр функции – переменная `X` вещественного, тип результата функции указан после списка параметров в круглых скобках и двоеточия, заголовок функции завершен символом «;». В теле функции располагается один оператор присваивания, который вычисляет значение арифметического выражения в соответствии с известной формулой $\sinh x = \frac{e^x - e^{-x}}{2}$ и возвращает его как результат функции путем присваивания предопределенной переменной `Result`, которая автоматически доступна при записи операторов в теле функции.

Одной из типичных ошибок является отсутствие в теле функции присваивания возвращаемого значения ее результата. В результате вызова подобной функции будут выполнены операторы, входящие в составе ее тела, однако результат будет не определен, что может привести к некорректному и непредсказуемому поведению программы. При наличии подобной ситуации компилятор выдаст соответствующее предупреждение, однако синтаксически программа при этом считается корректной и будет запущена на исполнение. Некоторые другие среды разработки (например, Microsoft Visual Studio последних версий) квалифицируют подобную ошибку как синтаксическую.

Если список параметров подпрограммы пуст, то круглые скобки можно не писать как при объявлении подпрограммы, так и при ее вызове. В некоторых других языках (например, в Си) наличие круглых скобок обязательно. Так приведенные ниже заголовки функций являются синтаксически правильными. Использование того или иного стиля записи определяется личными предпочтениями программиста. Некоторые из них считают, что использование скобок позволяет визуально отличать вызов

функции от обращения к переменной или константе при просмотре исходного кода программы, что является достаточно удобным.

```
function A(): Integer;
function B: Integer;
```

Вызов разработанных подпрограммы ничем не отличается от уже неоднократно использованных ранее вызовов стандартных подпрограмм:

```
Имя(параметры)
```

Для приведенного выше примера подпрограммы он может выглядеть так:

```
y := SinH(x);
```

Оформления списка параметров подпрограммы имеет ряд особенностей. По оформлению и использованию его элементы очень похожи на обычные переменные: они имеют имя, тип и могут использоваться в пределах подпрограммы. Однако сфера их использования несколько шире: с их помощью подпрограмма может обмениваться данными с другими подпрограммами и основной программой.

Вместо параметров можно использовать глобальные переменные, однако делать этого настоятельно не рекомендуется, т.к. с ростом объема исходного кода программы разобраться в том, какая переменная используется в каких подпрограммах будет все сложнее.

Список параметров, как уже было отмечено выше, задается в заголовке подпрограммы в круглых скобках и по правилам записи похож на объявление переменных. В общем виде он состоит из конструкций вида:

```
Модификатор Имя1, …, ИмяN: Тип;
```

отделяемых друг от друга символом «;». Параметры, указывается в заголовке подпрограммы, называются *формальными параметрами*, а указываемые при

вызове подпрограммы – *фактическими параметрами*. В приведенном выше примере с выводом матрицы на экран переменная `A` является фактическим параметром, а параметр `M` – формальным. Типы фактических и формальных параметров должны совпадать или быть приводимыми друг к другу. При вызове подпрограммы формальные параметры получают значения от соответствующих фактических параметров. Например, при выполнении оператора присваивания

```
y := SinH(0.5);
```

формальный параметр `X` получает значение `0.5`, указанное в виде фактического параметра.

На имена параметров накладываются те же ограничения, что и на любые другие идентификаторы, в качестве типа может выступать любой из объявленных ранее типов, пояснения требует элемент модификатор, который может быть указан перед группой параметров. При его указании возможны четыре различных ситуации:

1. модификатор отсутствует – производится т.н. передача параметра по значению;
2. модификатор `var` – производится т.н. передача параметра по ссылке, обычно используется для передачи значения в подпрограмму и его возврата обратно;
3. модификатор `const` – производится передача параметра по ссылке, значение параметра нельзя изменять в подпрограмме (при попытке изменения будет выдано соответствующее сообщение об ошибке, программа не будет откомпилирована);
4. модификатор `out` – передача по ссылке, используется только для возврата значения параметра из подпрограммы.

При передаче параметра по значению модификатор не используется, в качестве параметра допускается указывать как обычную переменную, так и

выражение, изменение значения формального параметра в подпрограмме никак не отражается на изменении значения фактического параметра (иногда говорят, что работа ведется с копией оригинала). При передаче значения формального параметра по ссылке на самом деле в подпрограмму передается адрес той переменной, которая указана в качестве фактического параметра, и все действия, выполняемые операторами подпрограммы с данным параметром, производятся с данной переменной (работа непосредственно с оригиналом). Значение параметра, передаваемого по ссылке, должно иметь вполне конкретный адрес (другими словами, быть леводопустимым), поэтому передача выражений в данных параметрах недопустима:

```
procedure A(var X: Integer);
begin
  ...
end;

var
  P, Q: Integer;

begin
  A(P);    // Правильно
  A(P+Q);  // Ошибка – выражение P+Q не является леводопустимым
  A(0.5);  // Аналогичная ошибка
end.
```

Модификатор `var` обычно используется, если с передаваемым значением параметра необходимо что-то сделать и возвратить результат изменений. Например, с его использованием можно реализовать аналог процедуры `Inc`, входящей в состав стандартной библиотеки:

```
procedure MyInc(var X: Integer; Y: Integer);
begin
  X := X + Y;
end;
```

При передаче значений простых типов с точки зрения скорости работы программы не представляет принципиальной разницы то, как передается параметр. Однако при передаче в качестве параметра переменной структурного типа (например, записи или статического массива) на создание

в памяти ее копии может уйти дополнительное время. Чтобы этого избежать, рекомендуется использовать модификатор `const`.

При передаче параметров используется специальная область памяти, называемая *стеком*, в которой сохраняются значения параметров, адрес возврата и некоторые другие данные (например, значения регистров процессора). Стек располагается в специальном сегменте памяти, называемом сегментом стека, и имеет ограничение на размер, задаваемое в свойствах линкера (Project → Options → Linker → Max stack size). При превышении заданного ограничения программа аварийно завершается с сообщением о переполнении стека (англ. stack overflow). Подобная ситуация может возникать при попытке передачи по значению больших структур данных. Выходом из ситуации является добавление модификатора с целью передачи параметра по ссылке.

Модификатор `out` по задумке разработчиков языка необходимо использовать в том случае, если значение исходное значение параметра не используется, нас интересует лишь возвращаемое результирующее значение. С точки зрения низкоуровневой реализации данный модификатор в точности повторяет функциональность модификатора `var`, поэтому на практике модификатор `out` используют редко. С использованием одного из этих модификаторов можно подтвердить высказанный выше тезис о том, что процедуры и функции являются взаимозаменяемыми, оформив функцию `SinH` в виде процедуры:

```
procedure SinH(const X: Double; var Y: Double);
begin
  Y := (Exp(X) - Exp(-X)) / 2;
end;
```

Механизм передачи параметров через стек не является быстрым, поэтому при включении оптимизации компилятор старается передавать значения параметров простых типов через регистры процессора или стек сопроцессора, что требует меньшего количества ассемблерных команд и работает быстрее.

При передаче в качестве параметров динамических массивов или объектов тип передачи параметра (по ссылке или по значению) не играет особой роли, т.к. работа с параметром производится через указатель и копирования оригинала не происходит. При этом нужно соблюдать определенную осторожность, т.к. кажущаяся работа с копией оригинала может оказать влияние на оригинал.

```
type
  TArr = array of Integer;

var
  A: TArr;

procedure Test(X: TArr);
begin
  X[0] := 20;
end;

begin
  SetLength(A, 1);

  A[0] := 10;

  Test(A);

  Writeln(A[0]);  // Несмотря на передачу по значению A[0] = 20, а не 10
end.
```

При передаче параметров по значению можно использовать т.н. *значения по умолчанию*: если значение параметра не указано, оно задается равным указанному значению по умолчанию. Поясним сказанное на примере.

```
procedure SomeProc(X: Integer; Y: Integer = 1);
begin
  Writeln(X, ' ', Y);
end;

begin
  SomeProc(10, 20);  // На экран будет выведено «10 20»
  SomeProc(100);     // На экран будет выведено «100 1»
  Readln;
end.
```

В приведенном примере первый параметр является обычным, а второй имеет значение по умолчанию, равное 1. При вызове процедуры с указанием одного (первого) параметра он получает значение 100, второй параметр при этом получает значение по умолчанию 1.

Параметры со значениями по умолчанию допускается использовать только последними в списке параметров подпрограммы, иначе при компиляции программы может возникнуть неоднозначность. Данным приемом удобно пользоваться, например, при передаче значения, которое при большинстве вызовов подпрограммы является одним и тем же, и лишь в редких случаях требует внесения корректив в логику работы подпрограммы. Рассмотрим следующий пример: необходимо найти среднее арифметическое среди элементов массива, при этом в некоторых случаях необходимо не учитывать нулевые элементы. Указанное действие может быть реализовано следующим образом:

```
function GetAverageValue(const A: TArray; UseZeroes: Boolean = True): Double;
var
  S, C, I: Integer;
begin
  S := 0;
  C := 0;

  for I := Low(A) to High(A) do begin
    if not UseZeroes and (A[I] = 0) then
      continue;

    S := S + A[I];
    C := C + 1;
  end;

  Result := S / C;
end;
```

При этом при вызове функции из большинства мест можно использовать более короткий формат вызова с меньшим числом параметров:

```
Avg := GetAverageValue(Arr);
```

Если же требуется специфичное поведение, то можно указать второй параметр:

```
NoneZeroAvg := GetAverageValue(Arr, False);
```

В процессе разработки большой программы часто бывает так, что подпрограмма уже написана, отлажена и используется во многих модулях программы, как вдруг требуется незначительно изменить ее

функциональность, для чего требуются дополнительные параметры. При этом в большинстве мест вызова требуется старое поведение и лишь в некоторых действительно необходимы тонкости. Если просто расширить список параметров подпрограммы путем добавления дополнительных параметров, то во всех местах ее вызова компилятором будет сформирована синтаксическая ошибка, в программу придется внести много изменений. Элегантным выходом из ситуации является добавление группы новых параметров со значениями по умолчанию: в таком случае программа в целом останется синтаксически корректной и не потребует внесения изменений во все места вызова подпрограммы.

Синтаксисом языка предусмотрена возможность передачи нетипизированных параметров:

```
procedure SomeProc(var A);
```

Данная возможность используется в программах достаточно редко и может быть реализована другими средствами (например, через указатели). При подобном вызове ответственность за корректную работу целиком ложится на программиста, т.к. компилятор не может проверить корректность вызова на предмет совпадения типов фактических и формальных параметров.

При передаче параметров иногда возникает соблазн сделать подпрограмму, которая будет оперировать с произвольным числом параметров произвольного типа (в отличие от жестко заданного прототипа вызова, как в рассмотренных выше примерах). Для этого в различных языках придуманы различные конструкции, в Delphi для этого используется специальный тип аргумента – `array of const`:

```
procedure Test(X: array of const);
```

При этом переменное число параметров на самом деле передается через массив `X`, элементы которого имеют предопределенный тип `TVarRec`

и фактически представляют собой запись типа TVarRec с вариантной частью.

```
TVarRec = record
  case Byte of
    vtInteger:    (VInteger: Integer; VType: Byte);
    vtBoolean:    (VBoolean: Boolean);
    vtChar:       (VChar: Char);
    vtExtended:   (VExtended: PExtended);
    vtString:     (VString: PShortString);
    vtPointer:    (VPointer: Pointer);
    vtPChar:      (VPChar: PChar);
    vtObject:     (VObject: TObject);
    vtClass:      (VClass: TClass);
    vtWideChar:   (VWideChar: WideChar);
    vtPWideChar:  (VPWideChar: PWideChar);
    vtAnsiString: (VAnsiString: Pointer);
    vtCurrency:   (VCurrency: PCurrency);
    vtVariant:    (VVariant: PVariant);
    vtInterface:  (VInterface: Pointer);
    vtWideString: (VWideString: Pointer);
    vtInt64:      (VInt64: PInt64);
end;
```

Вызов указанной процедуры производится, например, следующим образом:

```
Test([1, 'x']);
```

Для обращения к отдельным параметрам в рамках массива X используются соответствующие элементы массива и поля вариантной части записи:

```
Writeln(X[0].VInteger, ' ', X[1].VChar);
```

Если типы параметров не соответствуют тому, что ожидает подпрограмма, сообщений о синтаксической ошибке не выводится, однако поведение программы является непредсказуемым. Например, если для указанной выше подпрограммы произвести вызов как

```
Test(['x', 1]);
```

то вместо ожидаемых значений «x» и «1» на экране отображается число «120» и смайлик. Типы элементов массива можно определить с использованием поля VType, принимающего одно из указанных в объявлении записи значений (vtInteger, vtBoolean, vtChar, …).

Данная возможность языка используется нечасто, контроль за соблюдением правильности вызова при использовании указанной конструкции ложится на программиста, т.к. компилятор не может проверить совместимость типов передаваемых параметров.

Для преждевременного завершения подпрограммы может быть использована предопределенная процедура `exit`. Один из возможных примеров ее использования приведен ниже.

```
function XXX(...): Integer;
begin
  if исходные_данные_некорректны then begin
    Result := 1;  // Код ошибки
    exit;
  end;

  // Выполнение основных действий
  ...

  if ошибка_в_ходе_обработки then begin
    Result := 2;
    exit;
  end;

  // Ошибок нет
  Result := 0;
end;
```

В приведенном примере некая подпрограмма сперва проверяет корректность переданных ей параметров и прерывает свое выполнение в случае выявления ошибки. При этом результат функции является ненулевым, что свидетельствует о наличии ошибки и может быть легко проверено вызывающей подпрограммой. Если ошибок нет, то результат функции будет нулевым.

Рекурсия

Рекурсия – многогранное понятие, встречающееся в различных сферах деятельности, начиная от лингвистики или изобразительного искусства и заканчивая математикой или физикой. Неформальным образом рекурсию можно определить как факт наличия в описании или определении объекта

элементов самоподобия, выражения через собственные частности. В контексте повествования нас будет интересовать рекурсия применительно к точным наукам, в первую очередь к математике, и ее отражение в алгоритмических приемах и программировании. Рекурсия в математике – вычислительный процесс (функция), при котором вычисления на текущей итерации зависят от одной или нескольких предыдущих. В программировании под рекурсией обычно понимается процесс вызова подпрограммой самой себя, что во многом соответствует математической трактовке. Число вложенных вызовов называется *глубиной рекурсии*, каждый *рекуррентный вызов* (или *спуск*) производит погружение на один уровень рекурсии вниз, чему впоследствии соответствует *рекуррентный возврат*, обеспечивающий подъем на один уровень вверх. Для решения различных задач могут быть разработаны как итеративные, так и рекуррентные алгоритмы и соответствующие им программные реализации, причем для решения некоторых задач бывает эффективнее использовать итеративные подходы взамен рекурсивных, иногда наоборот.

Обычно рекуррентное описание вычислительного процесса характеризуется двумя обязательными элементами: выражением для рекуррентного спуска (выражение общего через частное) и условием завершения рекурсии (иначе рекурсия будет бесконечной). При отсутствующем или неправильно поставленном условии завершения рекурсии рано или поздно происходит переполнение стека, т.к. каждый новый вызов подпрограммы добавляет несколько байт в стек, а он имеет ограничение на размер. Рекурсия бывает *прямой*, когда подпрограмма вызывает сама себя, или *косвенной*, когда группа подпрограмм вызывают друг друга по цепочке.

Рассмотрим несколько примеров рекурсии в математике и их реализацию в Delphi. Общеизвестным и одним из наиболее простых примеров использования рекурсии является функция факториал, которая

определяется как $n!=1\cdot 2\cdot ...\cdot (n-1)\cdot n=\prod_{k=1}^{n} k$. Подобная запись называется *итеративной* и не содержит рекуррентных элементов. Чтобы перейти к рекуррентному описанию данной функции, необходимо заметить, что в приведенной формуле произведение первых $(n-1)$ сомножителей есть ни что иное, как $(n-1)!$, значит $n!=n\cdot(n-1)!$. Полученная запись называется *рекуррентной*, она выражает текущее значение функции через предыдущее, но для полного описания ей не хватает условия завершения рекурсии. Т.к. в математике принято считать факториал для положительных чисел (иногда к ним добавляют ноль), то условием завершения рекурсии в данном случае будет $1!=1$. Математически строгое описание рекурсии в данном случае будет выглядеть следующим образом:

$$f_n=\begin{cases} n\cdot f_{n-1}, n>1; \\ 1, n=1. \end{cases}$$

Аналогичным образом можно получить, например, формулу для рекуррентного вычисления результата операции возведения в степень x^n:

$$g_n(x)=\begin{cases} x\cdot g_{n-1}(x), n>0; \\ 1, n=0. \end{cases}$$

Функции вычисления факториала по приведенным выше формулам, соответствующие итеративной и рекуррентной формулам, приведены ниже.

```
function FactIterative(N: Integer): Extended;
var
  I: Integer;
begin
  Result := 1;
  for I := 1 to N do
    Result := Result * I;
end;

function FactRecurrent(N: Integer): Extended;
begin
  if N = 1 then
    Result := 1
  else
    Result := N * FactRecurrent(N-1);
end;
```

Они приводят к одному и тому же результату, однако итеративная функция в данной задаче считается более эффективной, т.к. она не использует каскадный вызов подпрограмм и сопутствующую ему работу со стеком. Вместо этого в ее теле расположен простой цикл со счетчиком, который выполняется с близкой к максимальной эффективностью. Рекурсия в приведенном примере называется *хвостовой*. Обычно такой тип рекурсии допускает тривиальное преобразование в итеративную запись, что не всегда возможно для более сложных видов рекуррентных зависимостей.

При вычислении значения факториала по рекуррентной схеме сперва производится серия рекуррентных спусков:

$$
\begin{gathered}
f_n = \\
= n \cdot f_{n-1} = \\
= n(n-1) \cdot f_{n-2} = \\
= n(n-1)(n-2) \cdot f_{n-3} = \\
= n(n-1)(n-2)(n-3) \cdot f_{n-4} = \\
\ldots \\
= n(n-1)(n-2)(n-3) \cdot \ldots \cdot 4 \cdot 3 \cdot 2 \cdot f_1
\end{gathered}
$$

Глубина рекурсии при этом равна n и определяется достижением уровня, на котором располагается последний сомножитель $f_1 = 1$, соответствующий условию завершения рекурсии. При достижении данного условия начинается обратно движение – рекуррентные возвраты, в ходе которых производится вычисление искомого значения функции (порядок выполнения действий отмечен квадратными скобками, цифрами обозначены уровни рекурсии):

$$
f_n = \underbrace{n\Big[(n-1)\Big[(n-2)\Big[(n-3)\cdot\ldots\cdot\Big[4\cdot\Big[3\cdot\Big[2\cdot\underbrace{f_1}_{n}\Big]_{n-1}\Big]_{n-2}\Big]_{n-3}\Big]\Big]\Big]}_{2}\Big]_{1}.
$$

Для того, чтобы убедиться в этом, можно открыть в отладчике окно Call Stack и произвести пошаговое выполнение рекуррентной функции, наблюдая за изменением значений переменных (рис. 1).

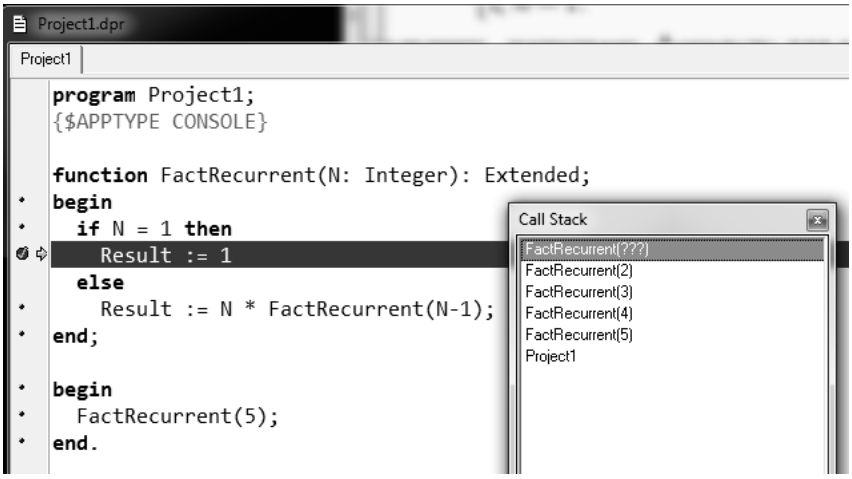

Рис. 1. Максимальная глубина рекурсии в окне Call Stack

Пример программы

Задача о ладьях. Подсчитать число способов K расстановки N ладей на шахматной доске размером $N \times N$ клеток так, чтобы они не атаковали друг друга.

Шахматная ладья представляет собой фигуру, которая ходит и атакует только по горизонталям и вертикалям (рис. 2).

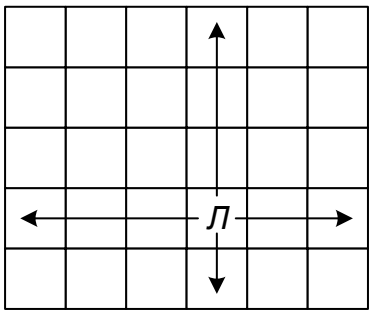

Рис. 2. Ход шахматной ладьи

Пример возможных решений задачи для случая $N = 3$ приведен на рис. 3.

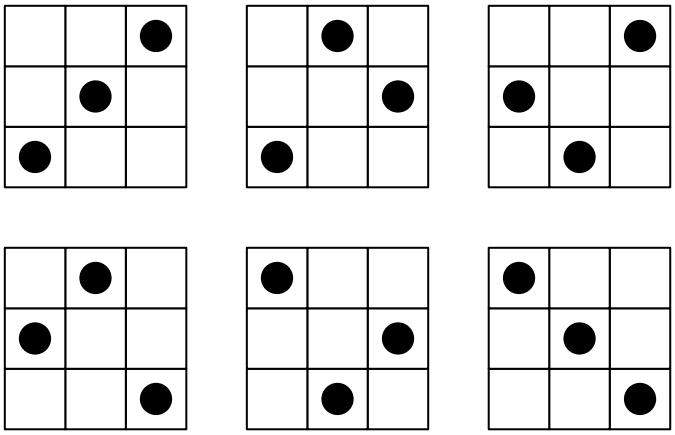

Рис. 3. Решение задачи о ладьях для $N = 3$, число способов $K = 6$

Для кодирования расположения ладей на шахматной доске возникает желание использовать двумерный массив, однако можно обойтись и одномерным, если заметить, что решения, удовлетворяющие условиям задачи, должны содержать такое расположение ладей, чтобы на каждой горизонтали и вертикали стояла ровно одна ладья. Соответственно приведенные решения можно кодировать в программе как координаты расположения одной из N ладей на каждой из N вертикалей. Для этого потребуется одномерный массив $A = [a_1, a_2, ..., a_N]$ из N элементов, причем приведенные на рис. 3 решения с использованием указанного способа представления будут выглядеть в программе как

$$[1, 2, 3], \quad [1, 3, 2], \quad [2, 1, 3],$$
$$[2, 3, 1], \quad [3, 1, 2], \quad [3, 2, 1].$$

Несложно заметить, что они представляют не что иное как перестановки. В этом нетрудно убедиться, т.к. в данной задаче важен порядок следования элементов (при мысленной перестановке двух столбцов шахматной доски мы получим другое решение), и повторения недопустимы (в противном случае

на одной горизонтали будет расположено несколько атакующих друг друга ладей).

При расстановке в аналогичной ситуации, например, пешек или слонов, которые могут располагаться на одной горизонтали и вертикали, подобный подход не подойдет и решения придется кодировать либо с использованием двумерного массива, либо с использованием двух одномерным массивов, хранящих координаты фигур.

Заполнение доски ладьями будем производить по столбцам слева направо. Первую ладью будем ставить на первую вертикаль шахматной доски, при этом ее можно установить в любую позицию a_1, чему соответствуют значения в массиве в диапазоне от 1 до N: $S_1 = \{1, 2, ..., N\}$, понимая под S_i множество доступных позиций для i-й вертикали. Вторую ладью будем устанавливать на вторую вертикаль во все возможные позиции, кроме позиции, соответствующей расположению первой ладьи: $a_2 \in S_2$, $S_2 = \{1, 2, ..., N\} \setminus \{a_1\}$. Рассуждая аналогично, $S_3 = \{1, 2, ..., N\} \setminus \{a_1, a_2\}$, $S_4 = \{1, 2, ..., N\} \setminus \{a_1, a_2, a_3\}$ и т.д.

Заполнение массива значениями в программе также будем производить слева направо, следуя описанному выше методу. При этом если N известно заранее, то программу можно оформить в виде N вложенных циклов. Для приведенного выше примера при $N = 3$ решение может быть следующим:

```
for I1 := 1 to N do
  for I2 := 1 to N do
    if I2 <> I1 then
      for I3 := 1 to N do
        if (I3 <> I1) and (I3 <> I2) then
          Writeln(I1, ' ', I2, ' ', I3);
```

В приведенном примере решения не используется массив, вместо него используются индексы циклов `I1`, `I2` и `I3`.

Пример является вполне работоспособным, однако что делать в случае, если значение N определяется во время работы программы? При этом потребуется разместить в программе заранее неопределенное количество

циклов, что на первый взгляд кажется невозможным. На выручку приходит рекурсия: на каждом рекуррентном уровне в программе будет расположен ровно один цикл `for`, а рекуррентный спуск будет аналогичен переходу к более вложенному циклу. Меняя число уровней рекурсии, фактически меняется число вложенных циклов: программа реализует задуманное, пусть и несколько иным способом, чем приведено выше. Условием завершения рекурсии будет погружение на N уровней в глубину, чему соответствует поочередное заполнение значений массива A, причем текущее значение глубины рекурсии, соответствующее текущей вертикали и, соответственно, текущей позиции массива A, передается в параметре `CurrN` рекуррентной процедуры `Rec`. Для быстрого определения уже использованных горизонталей в процедуру добавлен второй параметр `Used` типа множество, значение которого соответствует множеству уже использованных горизонталей. Программа, реализующая требуемые действия, приведена ниже.

```
const
  N = 4;

type
  TSet = set of Byte;

var
  Pos: array [1..N] of Integer;


procedure Rec(CurrN: Integer; Used: TSet);
var
  I, J: Integer;
begin
  // Условие завершения рекурсии
  if CurrN = N+1 then begin
    // Вывод шахматной доски
    for I := 1 to N do begin
      for J := 1 to Pos[I]-1 do
        Write('-');
      Write('o');
      for J := Pos[I]+1 to N do
        Write('-');
      Writeln;
    end;

    // При необходимости можно вывести соответствующую перестановку
    // for I := 1 to N do
    //   Write(Pos[I], ' ');

    Writeln;

    // Рекуррентный возврат
    exit;
  end;

  for I := 1 to N do
```

```
    if not (I in Used) then begin
      // Рекуррентный спуск
      Pos[CurrN] := I;
      Rec(CurrN+1, Used + [I]);
    end;
end;


begin
  Rec(1, []);
  Readln;
end.
```

Индивидуальные задания

1. Вычислить и вывести на экран таблицу чисел Стирлинга второго рода

$$S(n,m)=\begin{cases}1,\ m=n,\ n\geq 0,\\ 0,\ m=0,\ n\geq 0,\\ S(n-1,m-1)+m\cdot S(n-1,m),\ 0<m<n,\end{cases}$$

используя итеративную и рекуррентную подпрограммы. Например, $S(4,2)=7$, $S(6,4)=65$.

2. Проверить справедливость соотношения $x^n=\sum_{k=0}^{n}S(n,k)\cdot(x)_k$, где $S(n,k)$ – число Стирлинга второго рода (см. задание 1), $(x)_k=x(x-1)\cdot\ldots\cdot(x-k+1)$.

3. Проверить справедливость соотношения $S(m,n)=\sum_{i=n-1}^{m-1}C_{m-1}^{i}S(i,n-1)$, где $S(m,n)$ – число Стирлинга второго рода (см. задание 1), $C_n^k=\frac{n!}{k!(n-k)!}$ – число сочетаний из *n* по *k*.

4. Проверить справедливость формулы Добинского $\frac{1}{e}\sum_{k=0}^{\infty}\frac{k^n}{k!}=\sum_{m=1}^{n}S(n,m)$, где $S(n,m)$ – число Стирлинга второго рода (см. задание 1).

5. Проверить справедливость соотношения $e^{e^x-1}=\sum_{n=0}^{\infty}\frac{B_n}{n!}x^n$, где $B_n=\sum_{m=1}^{n}S(n,m)$ – число Белла.

6. Проверить справедливость формулы Паскаля $C_n^m=C_{n-1}^m+C_{n-1}^{m-1}$, $C_n^0=1$, где $C_n^m=\frac{n!}{m!(n-m)!}$ – число сочетаний из *n* по *m*, путем расчета значения по итеративной и рекуррентной формулам.

7. Вычислить значение полинома Чебышева первого рода порядка *n*: $T_{n+1}(x)=2xT_n(x)-T_{n-1}(x)$, $T_0(x)=1$, $T_1(x)=x$. Проверить полученное

значение с использованием итеративной формулы $T_n(x)=\dfrac{\left(x+\sqrt{x^2-1}\right)^n-\left(x-\sqrt{x^2-1}\right)^n}{2}$.

8. Вычислить значение полинома Чебышева второго рода порядка n: $U_{n+1}(x)=2xU_n(x)-U_{n-1}(x)$, $U_0(x)=1$, $U_1(x)=2x$. Проверить полученное значение с использованием итеративной формулы $U_n(x)=\dfrac{\left(x+\sqrt{x^2-1}\right)^{n+1}-\left(x-\sqrt{x^2-1}\right)^{n+1}}{2\sqrt{x^2-1}}$.

9. Проверить справедливость соотношения $T_n(\cos\alpha)=\cos(n\alpha)$, используя рекуррентную формулу для вычисления значения $T_n(x)$ (см. задание 7).

10. Проверить справедливость соотношения $U_n(\cos\alpha)=\dfrac{\sin\left((n+1)\alpha\right)}{\sin\alpha}$, используя рекуррентную формулу для вычисления значения $U_n(x)$ (см. задание 8).

11. Вычислить значение функции Аккермана $A(m,n)=\begin{cases}n+1, & m=0,\\ A(m-1,1), & m>0, n=0,\\ A\left(m-1, A(m,n-1)\right), & m>0, n>0.\end{cases}$ Например, $A(2,3)=29$, $A(1,4)=65533$.

12. Вычислить значение многочлена Лежандра $P_{n+1}(x)=\dfrac{2n+1}{n+1}xP_n(x)-\dfrac{n}{n+1}P_{n-1}(x)$, $P_0(x)=1$, $P_1(x)=x$.

13. Вычислить значение числа Каталана $C_n=\sum_{i=0}^{n-1}C_iC_{n-1-i}$, $C_0=1$.

14. Вычислить значение функции $f_n=f_{n-1}+2n-1$, $f_0=0$. Убедиться, что $f_n=n^2$.

15. Проверить справедливость соотношения $\dfrac{1-\sqrt{1-4x}}{2x}=\sum_{k=0}^{\infty}C_kx^k$, где C_k – последовательность чисел Каталана (см. задание 13).

16. Проверить справедливость соотношения $C_{n+1}=\dfrac{2(2n+1)}{n+2}C_n$, где C_n – последовательность чисел Каталана (см. задание 13).

17. Вычислить и вывести на экран таблицу чисел Стирлинга первого рода $s(n,k)=(n-1)s(n-1,k)+s(n-1,k-1)$, $s(0,0)=1$, $s(n,0)=0$ с использованием итеративной и рекуррентной форм вычисления.

18. Проверить справедливость равенства $\sum_{k=1}^{n} s(n,k) = n!$, где $s(n,k)$ – числа Стирлинга первого рода.

19. Вывести на экран таблицу гармонических чисел $H_n = \sum_{k=1}^{n} \frac{1}{k}$. Убедиться в справедливости формулы $H_n = \frac{1}{n!} s(n+1,2)$, где $s(n,k)$ – числа Стирлинга первого рода.

20. Кодом Грея G^n называется циклическая последовательность из 2^n неповторяющихся битовых строк длины n, соседние элементы которой отличаются значением одного бита. Например, $G^n = (000, 100, 110, 010, 011, 111, 101, 001)$. Синтезировать код Грея для заданного значения n с использованием следующего рекурсивного правила: первая половина битовых строк получается путем добавления нуля к битовым строкам кода G^{n-1}, вторая половина – путем добавления единицы к битовым строкам кода G^{n-1}, записанным в обратном порядке. Проверить полученную последовательность $G^n = \left(g_1^n, g_2^n, ..., g_{2^n}^n\right)$ с использованием формулы $g_i^n = i \oplus (i \gg 1)$, где $i \gg 1$ – поразрядный сдвиг вправо на 1 разряд.

Содержание отчета

1. Титульный лист.
2. Индивидуальное задание.
3. Краткое описание стратегии решения.
4. Листинг программы.
5. Тестовые примеры, результаты тестирования.
6. Выводы.

Контрольные вопросы

1. Для чего применяются подпрограммы?
2. В чем отличие процедур от функций?
3. Для чего используются параметры?
4. Какие модификаторы применяются при передаче параметров?
5. Чем отличается передача параметров по ссылке и по значению?
6. Что такое параметры по умолчанию?
7. Чем отличается определение понятия рекурсия в математике и программировании?

Библиографический список

1. Емельянов С.Г., Ватутин Э.И., Панищев В.С., Титов В.С. Процедурно-модульное программирование на Delphi: учебное пособие. М.: Аргамак-Медиа, 2014. 352 с.
2. Зотов И.В., Ватутин Э.И., Борзов Д.Б. Процедурно-ориентированное программирование на C++: учебное пособие. Курск: КурскГТУ, 2008. 211 с.