ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА»

РЕКУРСИВНЫЕ АЛГОРИТМЫ В C#

Самара 2017

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА»
»

РЕКУРСИВНЫЕ АЛГОРИТМЫ В C#

Составитель *Е.В. Симонова*

С А М А Р А
Издательство Самарского университета
2017

УДК 519.876.5

ББК 22.18я73

Составитель ***Е.В. Симонова***

Рецензент: канд. техн. наук, доц. Л.С. З е л е н к о

Рекурсивные алгоритмы в C#: [Электронный ресурс]: метод. указания / *Е.В. Симонова*. – Самара: Изд-во Самарского университета, 2017. – 39 с. : ил. Электрон. текстовые и граф. дан. (Кбайт).- 1 эл. опт. диск (CD-ROM)

Методические указания содержат теоретические сведения по организации и реализации рекурсивного вычислительного процесса, большое количество программных фрагментов, реализующих алгоритмы обработки списков, а также варианты заданий для выполнения лабораторных и самостоятельных работ.

Предназначены для студентов направления 09.03.01 – «Информатика и вычислительная техника» в качестве методических указаний по курсу «Программирование».

Подготовлены на кафедре информационных систем и технологий.

УДК 519.876.5
ББК 22.18я73

© Самарский университет, 2017

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ…………..……………………………………………… 5
ВВЕДЕНИЕ………………..………………………………………………… 5
1. ОРГАНИЗАЦИЯ СТЕКА…………………..…………………………….. 6
2. ОРГАНИЗАЦИЯ РЕКУРСИВНОГО ВЫЧИСЛИТЕЛЬНОГО ПРОЦЕССА ………………. ……………………………………………….. 9
3. ВИДЫ РЕКУРСИВНЫХ АЛГОРИТМОВ……………………………… 18
4. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ…………………………………. 32
5. КОНТРОЛЬНЫЕ ВОПРОСЫ……………………………………………. 33
6. ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ……………………………………… 33
ЗАКЛЮЧЕНИЕ ………..……………………………………………… 38
БИБЛИОГРАФИЧЕСКИЙ СПИСОК……………..……………………… 38

ПРЕДИСЛОВИЕ

В методических указаниях описаны структуры данных и алгоритмы, которые широко используются при решении разнообразных задач в широком спектре предметных областей.

Методические указания посвящены рассмотрению рекурсии как метода организации множества объектов и вычислительных процессов. Подробно описывается структура рекурсивного вычислительного процесса, особенности его реализации. Приводится большое количество примеров рекурсивных алгоритмов, нашедших широкое применение при решении задач различной природы.

Примеры программ, описывающих алгоритмы обработки структур данных различных типов, реализованы на языке C# версии 3.0 с использованием Visual Studio 2008 .NET Framework 3.5.

Методические указания предназначены для студентов, обучающихся по направлению 09.03.01 – Информатика и вычислительная техника.

Содержание методических указаний соответствует разделам рабочей программы по дисциплине «Программирование» федерального компонента ГОС подготовки бакалавров по направлению 09.03.01 – Информатика и вычислительная техника.

ВВЕДЕНИЕ

Цель лабораторной работы – изучение теоретических основ построения и методов программной реализации рекурсивных алгоритмов, получение навыков разработки программ, реализующих рекурсивные алгоритмы.

1 ОРГАНИЗАЦИЯ СТЕКА
1.1 Автоматическая память

Стек (автоматическая память) – это область памяти, которая сохраняет данные по принципу: “последним пришел – первым вышел” (*LIFO – Last Input First Output*). Стек предназначен для хранения данных значимых типов, а также для передачи параметров при выполнении методов. Доступ к элементам хранения переменных, находящихся в стеке, осуществляется по именам переменных. При идентификации именованием существует статическая связь между именем объекта и его элементом хранения (рис. 1).

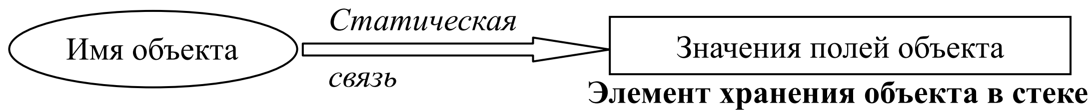


Рис. 1. Идентификация именованием объектов значимых типов

Стек поддерживает область видимости (время жизни) переменных. Если переменная *b* объявлена внутри области видимости переменной *a*, то вначале завершается вложенный блок кода и переменная *b* покидает область видимости, после чего из области видимости выходит переменная *a*. Таким образом, времена жизни переменных вложены друг в друга.

```
{
   int a = 10;
   // действия
  {
     double b = 20.5;
     // действия
   }
}
```

В любом компилируемом языке программирования высокого уровня компилятор преобразует имена переменных, указанные в программе, в адреса, используемые процессором. Для работы стека используется *указатель стека* – специальная переменная, поддерживаемая операционной системой и определяющая адрес следующего свободного байта в стеке.

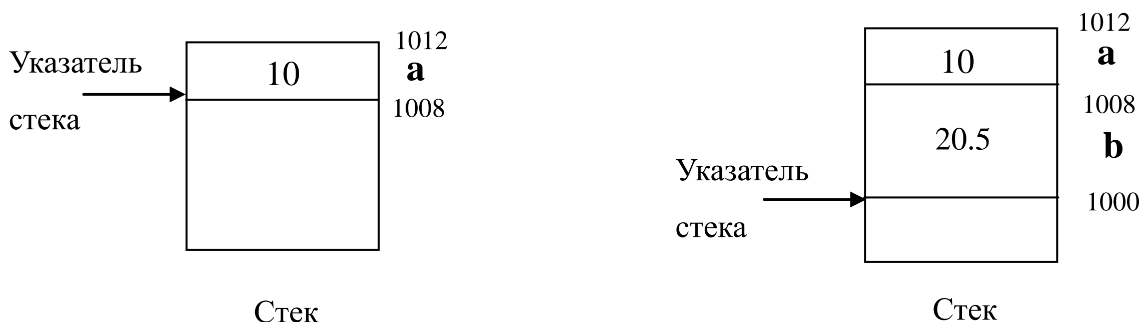


Рис. 2. Размещение переменных в стеке

Когда программа впервые запускается на выполнение, указатель стека установлен на верхний адрес блока памяти, зарезервированного для стека, т.к. стек заполняется от старших адресов памяти к младшим. Когда данные помещаются в стек, указатель стека уменьшается в соответствии с размером элемента хранения размещенных данных (рис. 2).

Например, пусть указатель стека первоначально установлен на адрес 1012. Переменная *a* размещается по адресу 1012, после чего указатель стека уменьшается на величину размера элемента хранения типа *int*, т.е. на 4 байта. Переменная *b* размещается по адресу 1008, после чего указатель стека уменьшается на величину размера элемента хранения типа *double*, т.е. на 8 байт и указывает на байт, расположенный в памяти по адресу 1000.

Удаление переменных из стека производится в порядке, обратном их размещению, по мере выхода переменных из областей их видимости. Указатель стека при этом увеличивается в соответствии с размером элемента хранения удаленных данных.

Стек управляется также директивами программы, связанными с вызовами методов и их окончанием. Каждому методу для работы требуется индивидуальная локальная среда, которая называется фреймом активации метода. ***Фрейм активации*** включает значения фактических параметров, подставляемых на место формальных параметров, указанных в заголовке метода, значения локальных переменных, описываемых внутри метода, а также элемент хранения адреса возврата из метода. Фрейм активации однозначно характеризует метод, т.к. содержит набор объектов, необходимых для его выполнения. Размещение локальной среды связано с активацией метода и происходит автоматически в момент его вызова, а удаление локальной среды связано с пассивацией метода при завершении его выполнения. В программе может одновременно существовать несколько активных методов. Последовательность активации и пассивации методов связана с вложенностью их вызовов (первым должен завершиться метод, который был позже всех вызван). Для обеспечения корректного выполнения вызовов методов в соответствии с дисциплиной *LIFO* используется структура стека. При активации каждого нового метода указатель стека “опускается вниз” на величину, определяемую размером локальной среды данного метода. При пассивации метода указатель стека “поднимается вверх” на эту же величину.

Ниже для фрагмента программы приведена иллюстрация распределения стека (рис. 3). В области *Main* программы описаны две переменные значимых типов: *x* и *y*. В стеке для переменной с именем *x* выделен элемент хранения размером *4 байта*, в который в результате выполнения операции присваивания занесено значение 10, для переменной с именем *y* выделен элемент хранения размером *8 байтов*, который инициализирован значением 0. Значение переменной *y* должно измениться после завершения выполнения метода *W2*, поэтому оно будет передаваться по ссылке.

```
using System;
namespace Storage_Stack
{
  class Storage_Stack
  {
     public static void W1()
       {
          int b;
          Console.WriteLine("W1");
       }

        public static void W2( int x1, ref float y1)
        {
           char a;
           y1 = 2 * x1; W1();
         }

         static void Main( string[] args )
         {
            int x = 10; float y = 0;
            W2( x, ref y );
            Console.WriteLine( y );
          }
   }
}
```

Активация метода *W2* приведет к созданию локальной среды, в которой будут размещены элементы хранения следующих объектов:

- значение фактического параметра *x* (4 байта), равное 10, подставляемого на место формального параметра *int x1*, т.к. данный параметр передается по значению,
- адрес (4 байта) в стеке переменной *y* – фактического параметра, подставляемого вместо формального параметра *ref float y1,* т.к. данный параметр передается по ссылке,
- значение локальной переменной *char a* (2 байта),
- адрес возврата из метода (4 байта).

В процессе выполнения метода *W2* происходит вызов метода *W1*. Активация метода *W1* приведет к созданию локальной среды, в которой будут размещены элементы хранения следующих объектов:

- значение локальной переменной *int b* (4 байта),
- адрес возврата из метода (4 байта).

Пассивация методов *W1*, *W2* и, соответственно, освобождение локальной среды каждого из них происходит в обратном порядке.

Если бы параметр *y1* был объявлен выходным – *out float y1*, распределение стека и передача параметров выполнялась бы аналогично, только не потребовалась бы инициализация фактического параметра *y*.

Рис. 3. Распределение стека

2 ОРГАНИЗАЦИЯ РЕКУРСИВНОГО ВЫЧИСЛИТЕЛЬНОГО ПРОЦЕССА

2.1. Итерация и рекурсия в программировании

2.1.1. Понятие рекурсии

Рекурсия – есть метод определения множества объектов или процесса в терминах самого себя.

Рекурсивные определения

Любое рекурсивное определение содержит две части: ***базисную часть*** и собственно ***рекурсивную***. Например, понятие нечетного целого числа определяется следующим образом:

- базисная часть: число 1 является нечетным целым числом;
- рекурсивная часть: если какое либо число К является нечетным целым числом, то нечетными целыми будут числа, определяемые выражениями К-2 и К+2.

Это определение состоит из двух независимых частей: базисной (или базы) и рекурсивной. Базисная часть является нерекурсивным утверждением, которое задает определение для некоторой фиксированной группы объектов. Рекурсивная часть определения записывается таким образом, чтобы при цепочке повторных применений утверждение из рекурсивной части приводилось бы к базисной части. Таким образом, базисная часть задает один или более случаев, которые удовлетворяют определению, а рекурсивная часть показывает, как применить определение, чтобы проверить, удовлетворяют ли ему другие случаи.

Например, докажем, что число К = 7 является нечетным целым числом. Для этого применим рекурсивную часть определения:

число К = 7 является нечетным целым числом, если нечетным целым является число К - 2 = 7 - 2 = 5;

число К = 5 является нечетным целым числом, если нечетным целым является число К - 2 = 5 - 2 = 3;

число К = 3 является нечетным целым числом, если нечетным целым является число К - 2 = 3 - 2 = 1;

число К = 1 является нечетным целым числом. Таким образом, рекурсивное утверждение удалось привести к базе, следовательно, первоначальное утверждение о том, что число К = 7 - нечетное целое число является истинным.

Рекурсивные алгоритмы

Рекурсивный алгоритм реализует какое-либо рекурсивное определение посредством разбиения решаемой задачи на подзадачи меньшей размерности, выполняемые с помощью одного и того же алгоритма. Процесс разбиения завершается при достижении простейших возможных решаемых задач минимальной размерности, которые называются условиями завершения. Таким образом, рекурсивное определение алгоритма включает две части:

- ***условия завершения*** (одно или несколько), которые могут быть вычислены для определенных параметров. Условия завершения соответствуют базисной части рекурсивного определения;
- ***шаг рекурсии***, в котором текущие значения некоторых переменных в алгоритме могут быть определены с использованием их предыдущих значений. В конечном итоге шаг рекурсии должен приводить к выполнению условий завершения.

Рекурсивные алгоритмы реализуются через ***рекурсивные методы (функции)***. Рекурсивным называется метод, который в процессе выполнения явно или неявно вызывает сам себя. ***Прямая (явная) рекурсия*** характеризуется наличием в теле метода оператора обращения к нему же самому (рис. 4.а). В случае ***косвенной (неявной) рекурсии*** один метод обращается к другому, который (возможно через цепочку вызовов других методов) вновь обращается к первому методу (рис. 4.б). Далее будем рассматривать только прямую рекурсию.

Метод А
……………..

Метод А **Метод В** **Метод С**

а **б**

Рис. 4. Схема прямой и косвенной рекурсии:
а – прямая рекурсия; б – косвенная рекурсия

2.1.2. Итеративная и рекурсивная схема организации вычислительного процесса

Для того чтобы лучше понять особенности рекурсивных алгоритмов, полезно сопоставить итеративнную и рекурсивную организацию процесса вычислений в программе. Особенности итеративного и рекурсивного вычислительного процесса рассмотрим на примере вычисления значения факториала некоторого натурального числа *N*.

Итеративная схема организации вычислительного процесса

Итеративный процесс можно проиллюстрировать с помощью схемы, приведенной на рис. 5. Этот процесс состоит из четырех блоков: инициализации, принятия решения (о продолжении вычислений), вычисления и модификации.

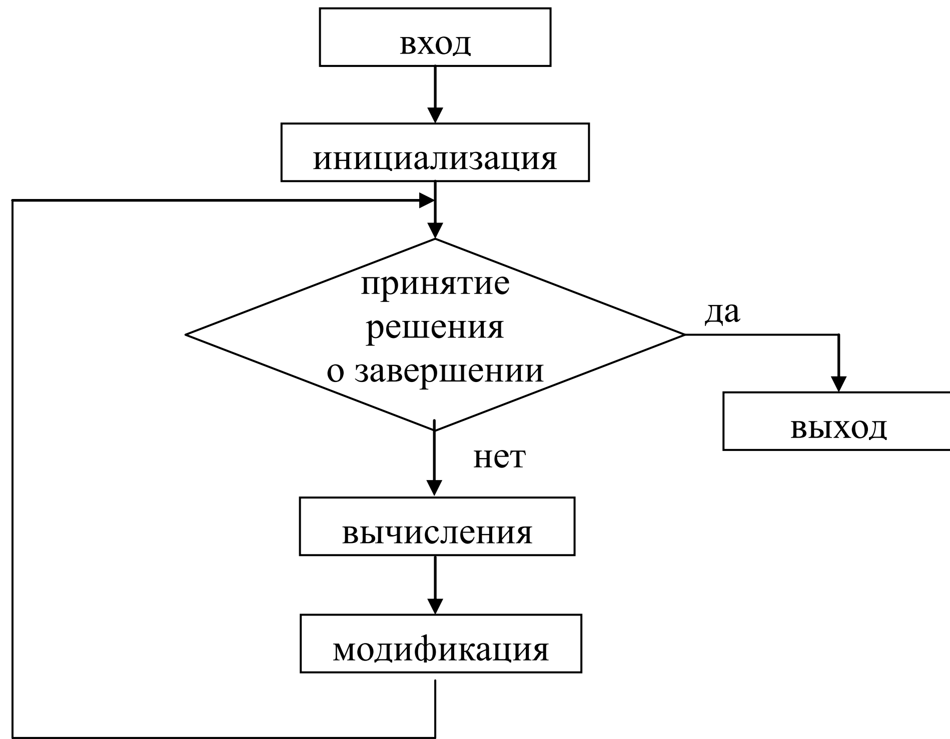


Рис. 5. Схема итеративного вычислительного процесса

В основе итеративного вычислительного процесса лежит ***итеративный цикл*** *While (с предусловием или постусловием)*, *For*. Наиболее универсальным является цикл *While*:

While < условие цикла > < тело цикла >;

Итеративная схема вычисления факториала:

N! = 1 * 2 * 3 * … * N.

Метод, реализующий итеративную схему вычисления факториала, приведен ниже:

```
using System;
namespace Iteration
{
  class Iteration
  {
     public static long Iter_fact( int n )
     {
        int i =1;  long f =1;                    // инициализация
        while ( i < = n )                        // решение о завершении
        {
          f = f * i;                             // вычисления
          i++;                                   // модификация
```

```
    }
    return f;
  }

  static void Main()
  {
    Console.Write( "Введите целое число:  " );
    int x = int.Parse( Console.ReadLine() );
    long y = Iter_fact( x);
    Console.WriteLine( "Итеративный факториал = {0}", y );
  }
 }
}
```

Существует два важных положения, известных в математике и в программировании, определяющих соотношение между итерацией и рекурсией:

1. любой итеративный цикл может быть заменен рекурсией;
2. рекурсия не всегда может быть заменена итерацией.

Рекурсивная схема организации вычислительного процесса

Общая схема рекурсивного вычислительного процесса представлена на рис. 6.

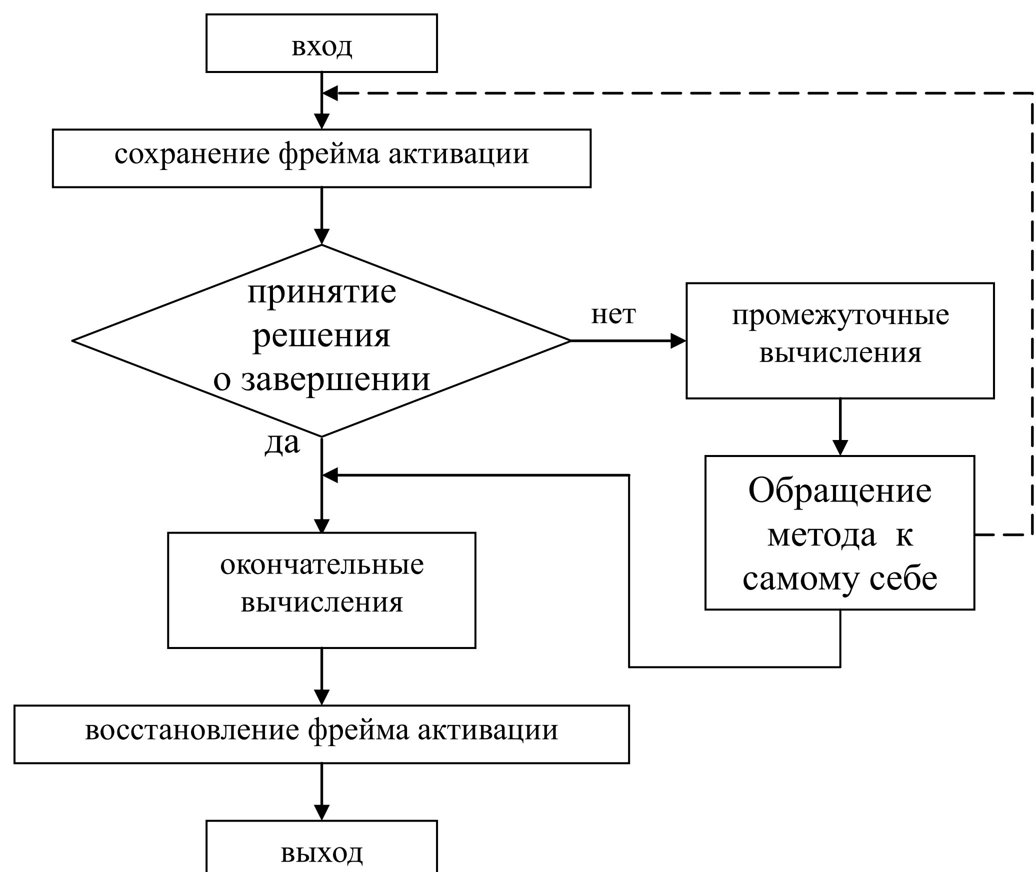


Рис. 6. Схема рекурсивного вычислительного процесса

Так как обращаться к рекурсивному методу можно как из него самого, так и извне, каждое обращение к рекурсивному методу вызывает его независимую активацию. При каждой активации образуются копии всех локальных переменных и формальных параметров рекурсивного метода, в которых “оставляют следы” операторы текущей активации. Таким образом, для рекурсивного метода может одновременно существовать несколько активаций. Для обеспечения правильного функционирования рекурсивного метода необходимо сохранять адреса возврата в таком порядке, чтобы возврат после завершения каждой текущей активации выполнялся в точку, соответствующую оператору, непосредственно следующему за оператором рекурсивного вызова. Совокупность локальных переменных, значений фактических параметров, подставляемых на место формальных параметров рекурсивного метода, и адреса возврата однозначно характеризует текущую активацию и образует ***фрейм активации***. Фрейм активации необходимо сохранять при очередной активации и восстанавливать после завершения текущей активации.

В блоке принятия решения (о продолжении вычислений) производится проверка, являются ли значения входных параметров такими, для которых возможно вычисление значений выходных параметров в соответствии с базисной частью рекурсивного определения. На основании этой проверки принимается решение о выполнении промежуточных или окончательных вычислений. Блок промежуточных вычислений можно объединить с блоком обращения к методу, если промежуточные вычисления очень просты. В блоке окончательных вычислений производится явное определение параметров-переменных метода для конкретных значений входных параметров, соответствующих текущей активации метода.

В основе рекурсивного вычислительного процесса лежит ***рекурсивный цикл***, который реализуется через вызов рекурсивного метода, причем каждая активация рекурсивного метода эквивалентна одному проходу итеративного цикла *While*.

Общая схема рекурсивного цикла:

```
[спецификаторы] тип Recursion_Cycle (…)
{
  if  < условие цикла >
  {
     < тело рекурсивного цикла; >
     Recursion Cycle (…);
   }
}
```

В теле рекурсивного цикла (в блоке промежуточных вычислений) обязательно должны содержаться операторы, изменяющие значения переменных, от которых зависит условие завершения рекурсивного цикла.

Напомним, что выполнение условия завершения рекурсивного цикла соответствует достижению базиса рекурсивного определения. Если значения этих переменных не успевают измениться в теле цикла до очередной активации рекурсивного метода, то возникает ***бесконечный рекурсивный цикл***.

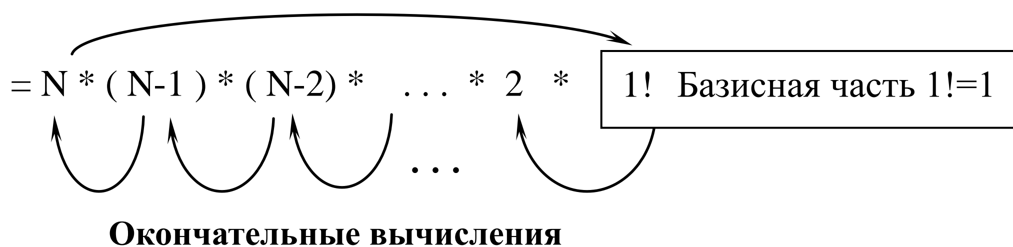
Общая схема бесконечного рекурсивного цикла:

```
[спецификаторы] тип Infinite Recursion_Cycle (…)
{
  if  < условие цикла >
  {
    Infinite Recursion_Cycle (…);
   < тело рекурсивного цикла; >
  }
}
```

Рекурсивная схема вычисления факториала:

Базисная часть:
0! = 1;   1! = 1;
Рекурсивная часть:
N! = N * (N-1)! = N * (N-1) * (N-2)! = N * (N-1) * … * (N – (N-1))! =

Промежуточные вычисления и обращения метода к самому себе



Метод, реализующий рекурсивную схему вычисления факториала:

```
using System;
namespace Recursion
{
  class Recursion
  {
     public static long Recurs_fact( int n )
     {
       long f;
       if  ( n == 0 ||  n == 1)                 // принятие решения о завершении  вычислений:
         f = 1;                            // да – окончательные вычисления для базисной части
```

```
        else
        {
          f = Recurs_fact ( n-1);                // нет – промежуточные вычисления и
                                                    обращение метода к самому себе
          f = f * n;             // окончательные вычисления для рекурсивной части { 1 }
         }
          return f;
       }                                                                     // { 2 }

     static void Main()
     {
       Console.Write( "Введите целое число:  " );
       int x = int.Parse( Console.ReadLine() );
       long y = Recurs_fact( x );
       Console.WriteLine( "Рекурсивный факториал = {0}", y );
     }
   }
}
```

{ 1 } – адрес возврата после завершения активации,

{ 2 } – завершение активации.

С каждым обращением к рекурсивному методу ассоциируется ***номер уровня рекурсии*** (номер фрейма активации). Считается, что при первоначальном вызове рекурсивного метода из основной программы номер уровня рекурсии равен единице. Каждый следующий вход в метод имеет номер уровня на единицу больше, чем номер уровня метода, из которого производится это обращение.

Другой характеристикой рекурсивного метода является ***глубина рекурсии***, определяемая максимальным уровнем рекурсии в процессе вычисления при заданных аргументах. В общем случае эта величина неочевидна, исключение составляют простые рекурсивные методы, например, при вычислении значения N! глубина рекурсии равна N.

Так как при выходе из текущей активации самым первым должен быть восстановлен фрейм, который был позже всех сохранен, для хранения фреймов используется область системного стека. Таблица 1 и рис. 7 поясняют механизм рекурсивных вычислений.

Таблица 1. Трасса вычисления 3!

№ уровня рекурсии	Знач-е вход. пар-ра n	Содержимое стека		Знач-е выход. пар-ра F	Выход из уровня
		До вызова N,F,(*)возврата	После вызова N,F,(*)возврата		
1	3	–,–,–	3, □ ,(*1*)		
2	2	3, □ ,(*1*)	2, □ ,(*1*) 3, □ ,(*1*)		
3	1	2, □ ,(*1*) 3, □ ,(*1*)	1, □ ,(*1*) 2, □ ,(*1*) 3, □ ,(*1*)		
		До возврата из уровня	После возврата из уровня		
3	1	1, □ ,(*1*) 2, □ ,(*1*) 3, □ ,(*1*)	2, □ ,(*1*) 3, □ ,(*1*)	1	(*2*)
2	2	2, □ ,(*1*) 3, □ ,(*1*)	3, □ ,(*1*)	2	(*2*)
1	3	3, □ ,(*1*)	–,–,–	6	(*2*)

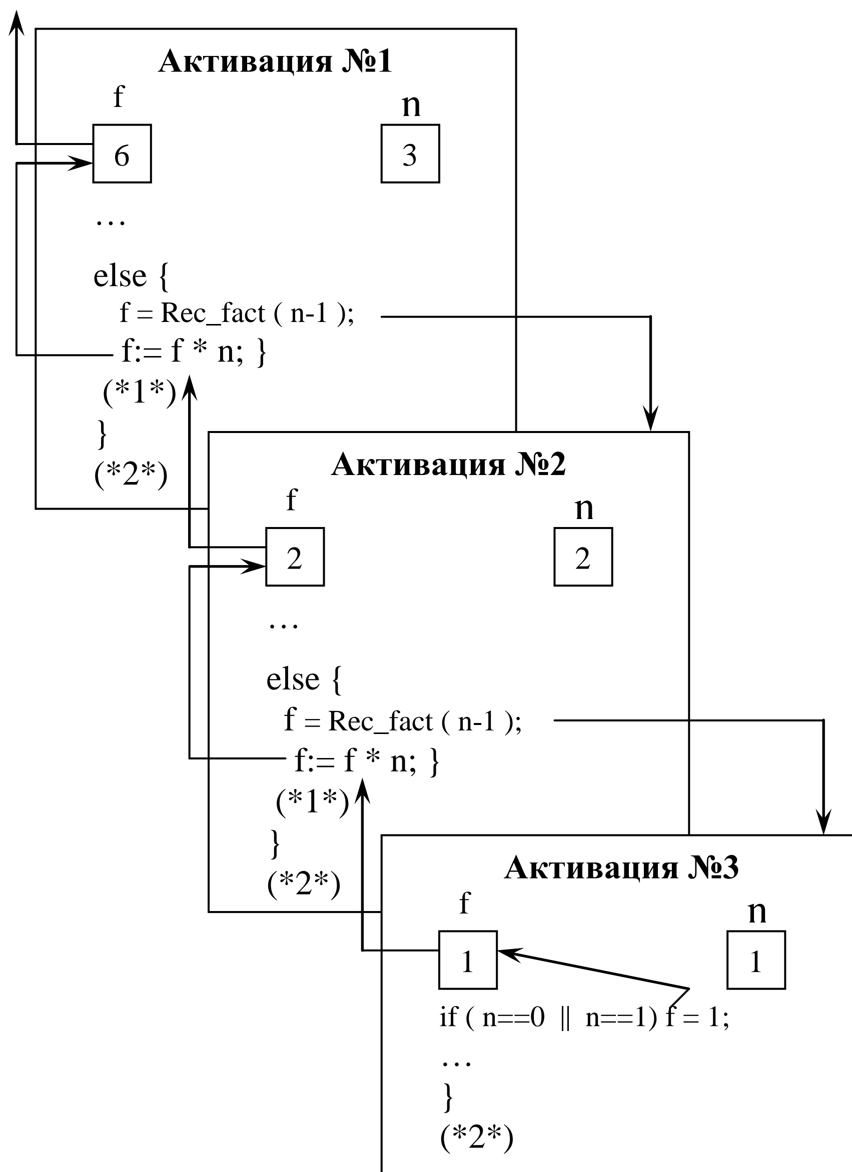


Рис. 7. Фреймы активации при вычислении 3!

3 ВИДЫ РЕКУРСИВНЫХ АЛГОРИТМОВ

3.1. Вычислительные алгоритмы

а) Вычисление n-го по счету члена числовой последовательности.

Примером числовой последовательности являются числа Фибоначчи. Числа Фибоначчи составляют последовательность, очередной элемент которой вычисляется по двум предыдущим значениям:

$$F_n = F_{n-1} + F_{n-2}.$$

Нулевое и первое значения должны быть заданы и равны единице. Последовательности такого рода применяются, например, в программных генераторах случайных чисел:

```
using System;
namespace Fibonacci;
{
  class Fibonacci
  {
     public static long Fib( int n )
    {
      if ( n == 0  ||  n == 1 ) return 1;                    // условие завершения
     else return  Fib( n-2 ) + Fib( n-1 )                    // шаг рекурсии
     }

   static void Main()
   {
     Console.WriteLine("Введите значение n ");
     int n = int.Parse( Console.ReadLine() );
     Console.WriteLine("{0}", Fib( n )  );
    }
  }
}
```

Каждое обращение при $n > 1$ приводит к двум дальнейшим обращениям, т.е. общее число обращений растет экспоненциально. Очевидно, что числа Фибоначчи более эффективно можно вычислять по итеративной схеме.

Однако существуют такие числовые последовательности, для которых применение рекурсии – единственный способ решения. Например:

$$a(1) = 1; \; a(n) = n - a(a(n-1)), \; n > 1.$$

б) Вычисление значений полиномов порядка n в заданной точке x. Например, полиномы Чебышева определяются следующим образом:

$$T_0(x) = 1; \qquad T_1(x) = x; \qquad T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x).$$

Таким образом, текущее значение полинома определяется по двум предыдущим значениям.

в) Решение разностных уравнений. Рассмотрим, например, уравнение следующего вида:

$$A(0) = 1; \qquad A(n) = A(n \ div \ 2) + A(n \ div \ 3).$$

Простого и очевидного способа итерационного программирования функции *A(n)* не существует. Решение разностных уравнений выполняется с помощью рекурсии.

г) Сортировки.

В качестве примера рассмотрим ***алгоритм быстрой сортировки***.

1. Выбрать в массиве некоторый элемент, называемый опорным элементом, желательно, чтобы опорный элемент разбивал множество элементов массива на две примерно равные части.
2. Выполнить операцию разделения массива таким образом, чтобы все элементы, меньшие или равные опорного элемента, оказались слева от него, а все элементы, большие опорного – справа от него. Это достигается путем просмотра массива попеременно с обоих концов, причем каждый элемент сравнивается с опорным. Элементы из левой части, не меньшие опорного, меняются местами с элементами из правой части, не большими опорного. После завершения процедуры разделения опорный элемент оказывается на своем окончательном месте. Все элементы, которые меньше опорного, будут стоять слева от него, а все элементы, которые больше опорного, - справа от него.
3. Выполнить пункты 1 и 2 над частями массива слева и справа от опорного элемента.

Базой рекурсии являются массивы, состоящие из одного или двух элементов, которые уже отсортированы. Алгоритм всегда завершается, поскольку за каждую итерацию по крайней мере один элемент размещается на его окончательном месте.

д) Бинарный поиск элемента с заданным значением в упорядоченном множестве объектов (массиве).

При бинарном поиске берется некоторый ключ и просматривается упорядоченный массив из *N* элементов на предмет совпадения с этим ключом (элементы индексируются от *1* до *N*). Результатом поиска является индекс совпавшего с ключом элемента или *0* при отсутствии такового. Алгоритм бинарного поиска может быть описан рекурсивно. Пусть упорядоченный массив *A* характеризуется нижним индексом *low* и верхним индексом *high*. Поиск совпадения с ключом начинается в середине массива (индекс *mid*):

mid = (low + high) / 2; сравнить *A[mid]* с ключом.

Если совпадение произошло, условие останова достигнуто, что позволяет прекратить поиск и возвратить индекс *mid*. Если совпадения не происходит, можно воспользоваться тем фактом, что массив упорядочен, и ограничить диапазон поиска "нижним подмассивом" (слева от *mid*) или "верхним подмассивом" (справа от *mid*).

Если ключ меньше *A[mid]*, совпадение может произойти только в левой половине массива в диапазоне индексов от *low* до *mid – 1*. Если ключ больше *A[mid]*, совпадение может произойти только в правой половине массива в диапазоне индексов от *mid + 1* до *high*.

Шаг рекурсии направляет бинарный поиск для продолжения в один из подмассивов. Рекурсивный процесс просматривает массивы все меньшего размера. Поиск заканчивается неудачей, если подмассивы исчезли, т.е. если нижняя граница превосходит верхнюю. Условие *low > high* – второе условие останова рекурсивного процесса. В этом случае алгоритм возвращает *0*.

3.2. Перебор с возвратами

Перебор с возвратами (бектрекинг - back tracking – обратное следование) – это способ поиска решения, когда при возврате после рассмотрения "пробного" варианта решения на один шаг назад все переменные программы восстанавливают свои значения. Классическим примером перебора с возвратами является ***алгоритм прохождения лабиринта***, идея которого состоит в следующем. Лабиринт есть множество перекрестков, связанных между собой. Предположим, что каждый перекресток имеет три атрибута, определяющих возможность перемещения налево, прямо и направо. Если значение некоторого атрибута равно единице, то движение в данном направлении возможно. Нуль показывает, что движение в данном направлении заблокировано. Три нулевых значения атрибутов перемещения из некоторого перекрестка представляют тупик. Проход по лабиринту считается успешным при достижении точки "Выход".

Процесс поиска выхода включает ряд рекурсивных шагов. В каждом перекрестке необходимо исследовать возможные варианты. Если возможно, сначала следует пойти налево. Достигнув очередного перекрестка, необходимо снова рассмотреть варианты и попытаться пойти налево. Если выбор левого направления заводит в тупик, следует отступить на один шаг назад и выбрать движение прямо, если это направление не заблокировано. Если этот выбор заводит в тупик, вновь необходимо отступить на один шаг и пойти направо. Если все альтернативы движения из данного перекрестка ведут в тупики, следует вернуться к предыдущему перекрестку и сделать новый выбор. Если произошел возврат к исходной точке, и из нее нет новых путей, задача поиска выхода из лабиринта не имеет решения. Рассмотрим варианты в лабиринте, изображенном на рис. 8.

Перекресток	Выбор	Результирующий перекресток
1	прямо	2
2	налево	3
3	налево	7
7	тупик: вернуться к 3	3
3	прямо	4
4	прямо	6
6	тупик: вернуться к 4	4
4	направо	5
5	тупик: вернуться к 4, 3, 2	2
2	прямо	8
8	налево	9
9	прямо	10 - выход

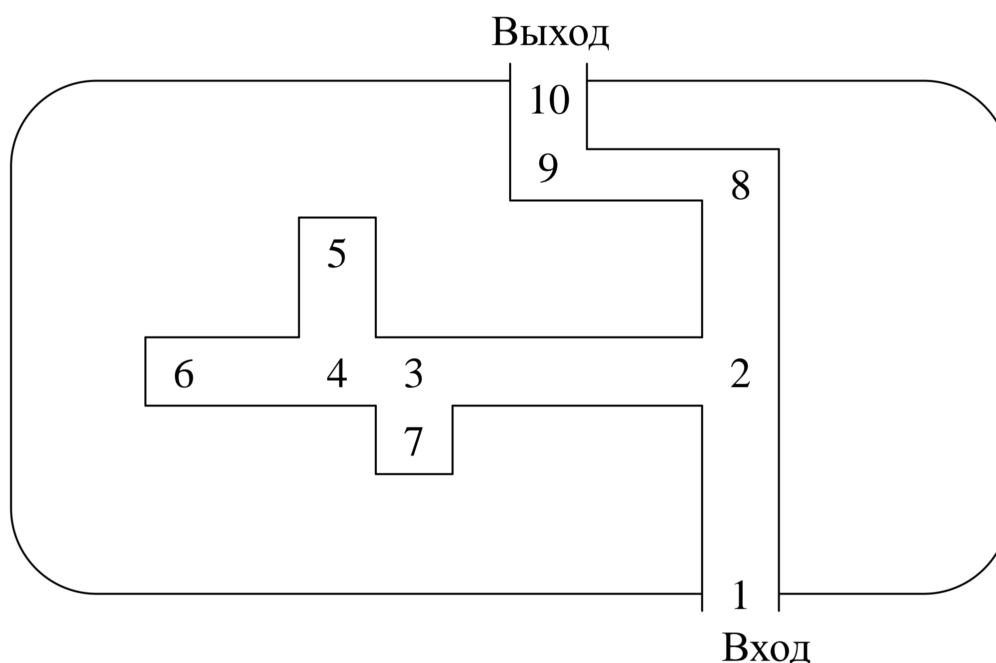


Рис. 8. Лабиринт

С помощью алгоритма бектрекинга решаются разнообразные ***шахматные задачи***. Наиболее известными являются следующие из них:

- задача о расстановке восьми ферзей. Требуется найти все возможные способы расстановки восьми ферзей на шахматной доске так, чтобы ни один из них не нападал на любого другого;
- задача о расстановке ладей. Требуется определить все возможные способы расстановки n ладей на шахматной доске размером $n * n$ так, чтобы они не угрожали друг другу;
- задача обхода шахматной доски размером 8 * 8 ходом коня так, чтобы конь побывал на каждом поле один раз.

3.3. Комбинаторика

Рекурсия находит широкое применение в комбинаторике – разделе математики, занимающемся подсчетом объектов.

Перестановка из N элементов (1,2, …, N) есть упорядоченное расположение этих элементов. Для N = 3 (1 3 2), (3 2 1), (1,2,3) – различные перестановки. В комбинаторике установлено, что число перестановок равно N!. Для позиции 1 существуют N вариантов, т.к. все N элементов доступны. Для позиции 2 имеются N – 1 вариантов, т.к. один элемент зафиксирован в позиции 1. Число вариантов уменьшается на 1 по мере продвижения по позициям. Общее число перестановок есть произведение числа вариантов в каждой позиции.

$$Permulation(N) = N * (N-1) * (N-2) * \ldots *2*1 = N!$$

Рекурсивный алгоритм генерирует перечень всех перестановок из N элементов для N > 1. Сформируем 24 (4!) перестановки из четырех элементов. Перестановки с одинаковыми первыми элементами записываются в отдельный столбец. Каждый столбец затем делится на три пары с одинаковыми вторыми элементами:

1	**2**	**3**	**4**
1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1
1 4 2 3	2 4 1 3	3 4 1 2	4 3 1 2
1 4 3 2	2 4 3 1	3 4 2 1	4 3 2 1

Алгоритм вычисления числа перестановок иллюстрируется иерархическим деревом (рис. 9), которое содержит упорядоченные пути, соответствующие перестановкам. В исходном положении имеется четыре варианта – 1, 2, 3 и 4, соответствующие четырем столбцам. По мере продвижения вниз по дереву нижние уровни разделяются на 3, 2 и 1 элемент, соответственно. Алгоритм генерации всех перестановок моделирует проход по путям дерева. Продвигаясь от уровня к уровню, мы тем самым указываем очередную позицию в перестановке. Этот процесс представляет собой шаг рекурсии.

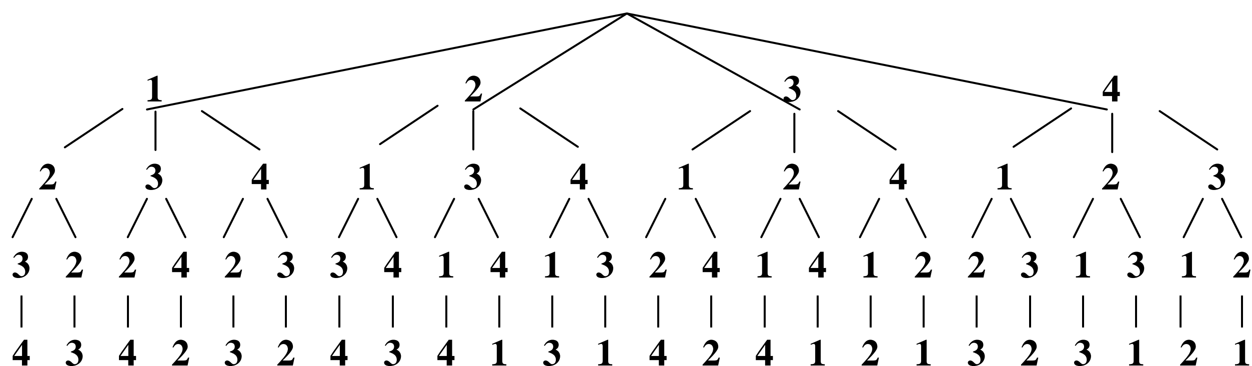
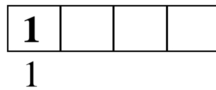
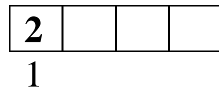
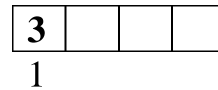
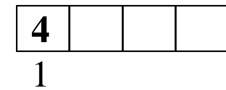


Рис. 9. Иллюстрация работы алгоритма перестановок

Шаг рекурсии выполняется следующим образом. Представим перестановку в виде массива из N элементов. Индекс элемента массива – это номер позиции перестановки. В каждом элементе массива хранится значение, соответствующее позиции перестановки, определяемой индексом элемента.

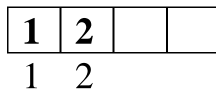
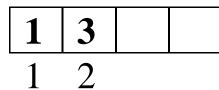
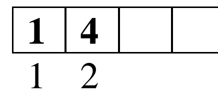
Итерационный процесс проверяет все возможные значения первого элемента перестановки. В нашем примере существует N=4 возможных значения первого элемента:

Индекс 1:

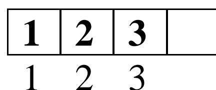
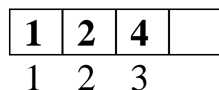
На следующем уровне дерева каждый узел дает начало N-1 перестановкам, содержащим N-1 отличных от первого элементов. Например, узел 1 соответствует перестановкам, которые начинаются с единицы в первой (индекс 1) позиции. Путь, ведущий к узлу 2 следующего уровня, соответствует перестановкам со значениями 1 и 2 в первых двух позициях и т.д. Можно итерационно определить второй элемент перестановки перебором узлов 2, 3 и 4:

Индекс 2:

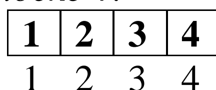
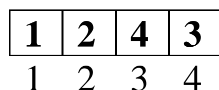
На следующем уровне дерева каждый узел разветвляется на два пути, которые представляют перестановки из двух элементов. Например, если во второй позиции (индекс 2) находится элемент 2, в третьей позиции (индекс 3) могут быть только элементы 3 или 4:

Индекс 3:

Так как третий элемент перестановки зафиксирован, последний элемент определяется однозначно, поскольку перестановка не допускает повторяющиеся значения:

Индекс 4:

Определение всех N элементов перестановки является ***условием завершения*** работы рекурсивного алгоритма.

Алгоритм перестановок можно запрограммировать, оперируя массивом A[1] ,…, A[n], в котором хранится перестановка чисел 1..n. Рекурсивный метод [11] распечатывает все перестановки (t позиций), в которых зафиксирована первая позиция.

```
using System;
namespace Permulation
{
  class Permulation
  {
    public static void Generate( int A[], int t )          // Перестановки чисел 1..N
    {
      int i, j, k;
      if  ( t == A.Length-1 )                               // Условие завершения шага
      {                                                     // Перестановка готова
        for ( i=0; i<A.Length; i++ ) Console.Write( "{0}", A[i]);
        Console.ReadLine(); Console.WriteLine();
       }
   else
     for ( j = t+1; j<A.Length; j++)              // Формирование перестановки: t < N
     {
       k = A[t+1]; A[t+1] = A[j]; A[j] = k;             // Поменять местами a[t+1] и a[j]
       Generate( A, t+1 );
       k = A[t+1]; A[t+1] = A[j]; A[j] = k;             // Поменять местами a[t+1] и a[j]
      }
    }

   static void Main()
   {
     Console.WriteLine("Введите целое число: ");
     int n = int.Parse( Console.ReadLine() );
     int[] Array = new int[n];                        // Массив для хранения перестановок
      for ( int i=0; i<A.Length; i++) Array[i] = i;    // Заполнение массива значениями
     int t = 0;
     Generate( Array,t );
    }
  }
}
```

3.4. Игры и головоломки: задача о “ханойских башнях”

Согласно легенде, у жрецов храма Брахмы в горах Тибета есть медная платформа с тремя алмазными стержнями A, B и C. На одном стержне A нанизано 64 золотых диска, каждый из которых немного меньше того, что

под ним. Конец света наступит, когда жрецы переместят диски со стержня А на стержень С. Задача имеет следующие условия:

- за один раз можно перемещать только один диск,
- больший диск нельзя класть на меньший,
- снятый диск нельзя отложить, его необходимо сразу же надеть на другой стержень.

Несомненно, жрецы все еще работают, т.к. задача включает в себя $2^{64} - 1$ ходов. Если тратить по одной секунде на ход, то потребуется примерно 500 миллиардов лет.

Решение данной задачи носит рекурсивный характер. Задача разбивается на последовательность подзадач одного и того же типа, но меньшей размерности. Условием завершения является выполнение простой задачи перемещения одного диска. Сформулируем алгоритм решения данной задачи для N дисков. Обозначим стержни: начальный (*start*), промежуточный (*temp*), конечный или целевой (*destination*). На начальный стержень нанизано N дисков в порядке возрастания размера, т.е. самый большой диск лежит внизу. Цель задачи – переместить все N дисков с начального стержня на конечный, следуя определенным выше условиям, и распечатать перечень ходов. Предполагается, что промежуточный стержень будет использоваться для временного хранения дисков.

Иллюстрация решения задачи для N = 1 диска приведена на рис. 10, для N = 2 дисков - на рис. 11, для N = 3 дисков – на рис. 12. Алгоритм требует $2^N - 1$ ходов.

Если N = 1, выполняется условие завершения (базисное условие рекурсивного алгоритма), которое может быть обработано путем перемещения единственного диска с начального стержня на конечный и распечатки соответствующего хода. Для N > 1 выполняется трехшаговый процесс перемещения N дисков с начального стержня на конечный, причем стержень А является начальным (*A – start*), стержень В – промежуточным (*B – temp*), стержень С – конечным (*C – dest*).

На первом шаге алгоритма перемещаются N – 1 дисков с начального стержня на промежуточный с использованием конечного стержня для временного хранения. При этом стержень А выполняет роль начального (*A – start*), стержень В выполняет роль конечного (*B – dest*). Конечный стержень С используется как промежуточный (*C – temp*).

На втором шаге самый большой диск просто перемещается с начального стержня на конечный: *start -> dest*.

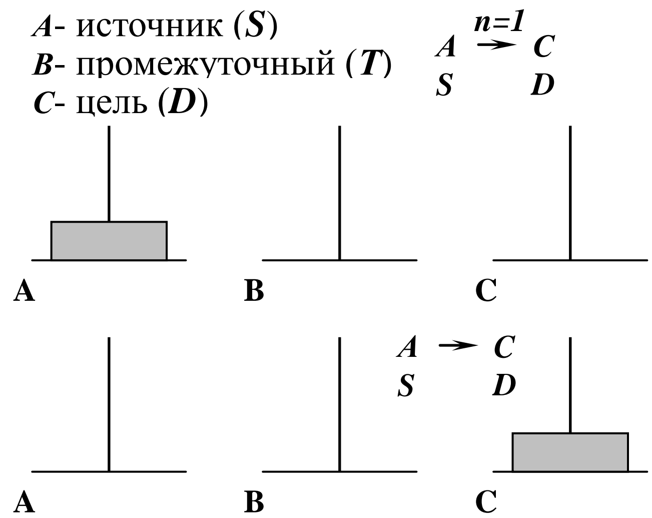


Рис. 10. Решение задачи о “ханойских башнях” для N = 1 диска

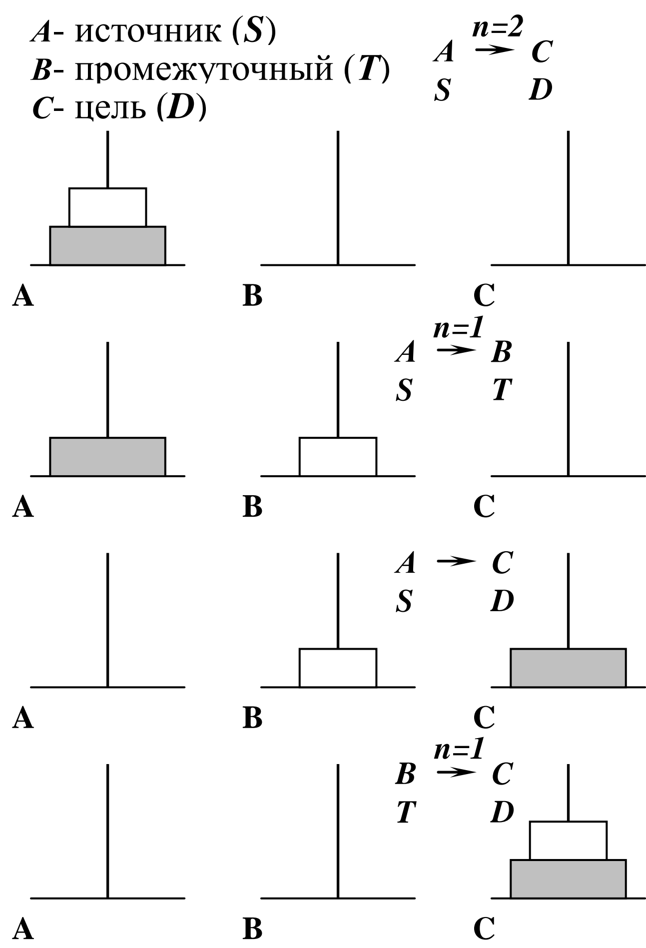


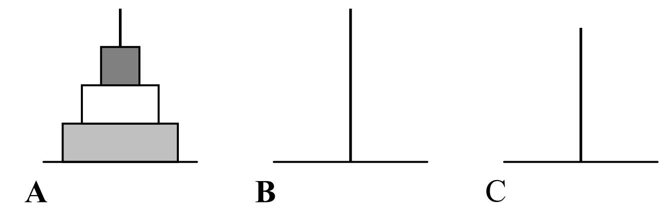
Рис. 11. Решение задачи о “ханойских башнях” для N = 2 дисков

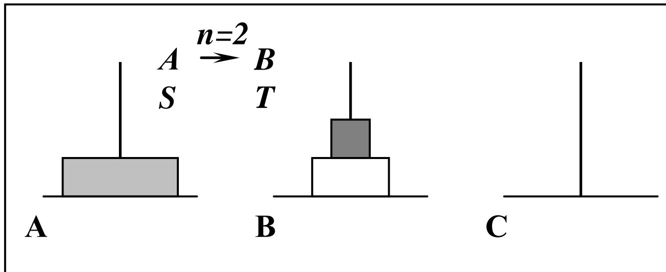
На третьем шаге N – 1 дисков перемещаются со среднего стержня на конечный с использованием начального стержня для временного хранения. При этом стержень B выполняет роль начального (*B – start*), стержень C

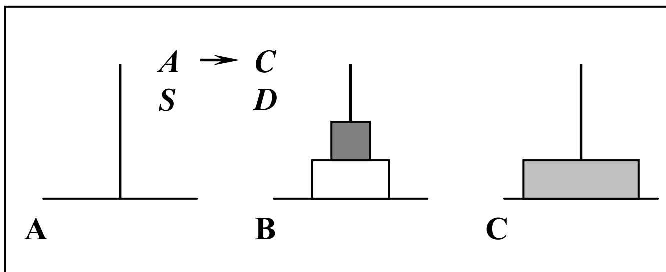
выполняет роль конечного (*C – dest*). Начальный стержень А используется как промежуточный (*A – temp*).

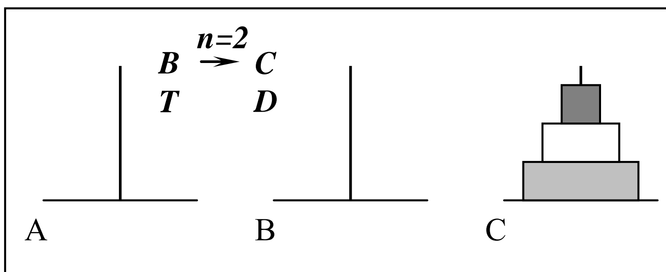
A- источник (***S***)
B- промежуточный (***T***)
C- цель (***D***)

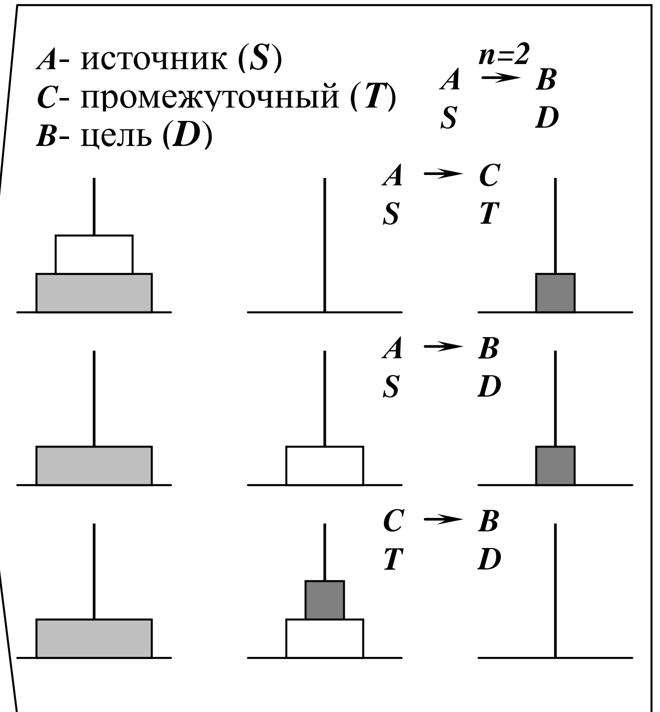
n=3
A → ***C***
S ***D***











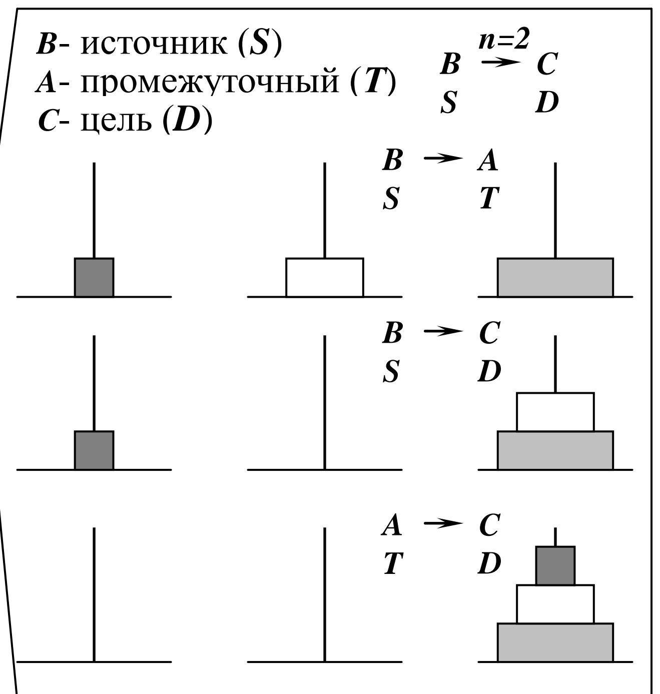


Рис. 12. Решение задачи о “ханойских башнях” для N = 3 дисков

Программная реализация алгоритма решения задачи о “ханойских башнях” следует ниже:

```
// перенести N дисков с начального стержня на конечный, используя промежуточный
//стержень для временного хранения дисков
Using System
{
  namespace Hanoi
  {
     public static void Hanoi( int n, char start, char tmp, chat dest )
     {
       if  ( n ==1 )                          // условие завершения: перемещение одного диска
          Console.WriteLine( "{0}, '-->', {1}", start, dest );
      else
      {
        // перенести N–1 дисков с начального стержня на промежуточный, используя
           конечный  стержень для временного хранения дисков
        Hanoi( n-1, start, dest, temp );
        // перенести нижний диск с начального стержня на конечный
        Console.WriteLine( "{0}, '-->', {1}", start, dest );
        // перенести N–1 дисков с промежуточного стержня на конечный, используя
           начальный  стержень для временного хранения дисков
        Hanoi( n-1, temp, start, dest )
      }
     }

    static void Main()
    {
       Console.WriteLine(" Введите количество дисков");
       int n = int.Parse( Console.ReadLine() );
       Hanoi( n, 'a', 'b', 'c');
       Console.ReadLine();
    }
  }
}
```

Более короткий вариант решения задачи о “ханойских башнях”:

```
// перенести N дисков с начального стержня на конечный, используя промежуточный
//стержень для временного хранения дисков
     public static void Hanoi( int n, char start, char tmp, chat dest )
     {
       if  ( n>0 )                                            // условие продолжения }
      {
        // перенести N–1 дисков с начального стержня на промежуточный, используя
           конечный  стержень для временного хранения дисков
       Hanoi( n-1, start, dest, temp );
        // перенести нижний диск с начального стержня на конечный
       Console.WriteLine( "{0}, '-->', {1}", start, dest );
```

```
// перенести N–1 дисков с промежуточного стержня на конечный, используя
   начальный  стержень для временного хранения дисков
 Hanoi( n-1, temp, start, dest )
 }
}
```

3.5. Арифметические выражения

При описании арифметических выражений рекурсия означает возможность вложенности, т.е. использование в выражениях в качестве операндов подвыражений, заключенных в круглые скобки. Синтаксис языков программирования принято описывать с помощью рекурсивной нотации, называемой формой Бэкуса-Наура (БНФ). Она состоит из правил подстановки, определяющих, каким образом нетерминальный символ (т.е. символ, требующий дальнейшего уточнения) может быть замещен другим нетерминальным или терминальным символом (т.е. символом, не требующим уточнения). В правилах подстановки используется знак «|» для разделения альтернативных подстановок. Нетерминальные символы заключаются в угловые скобки «<>». Символ «::=» означает «это есть».

Определение простейшего арифметического выражения с помощью БНФ выглядит следующим образом:

<арифметическое выражение>::=<операнд><знак операции><операнд>
<операнд> ::= <идентификатор> | (<арифметическое выражение>)
<знак операции> :: = + | - | * | /
<идентификатор> ::= a | b | … | z | A | B | … | Z

Эти правила указывают, что арифметическое выражение состоит из двух операндов, разделенных знаком операции. В свою очередь, любой операнд может представлять собой однобуквенный идентификатор или арифметическое выражение, заключенное в круглые скобки. Это означает, что арифметическое выражение определяется в терминах самого себя, следовательно, данное определение рекурсивно.

Примеры простейших арифметических выражений, соответствующих данному определению: *X+Y*   *X*(Y+Z)*   *(Y*Z)-X*   *(X+Y)*(Z-W)*   *(X/(Y+Z))*W* и т.п.

Рекурсивные алгоритмы наиболее эффективны и удобны для обработки рекурсивных структур данных.

3.6. Рекурсивные алгоритмы обработки динамических линейных структур данных на примере списков

Рекурсивное определение списка: список – это пустая структура или структура, состоящая из особого элемента (первого) и списка, на который указывает особый элемент. Рекурсивную интерпретацию линейного односвязного списка иллюстрирует рис. 13.

Как следует из рекурсивного определения списка, условием завершения его обработки является достижение пустого списка в процессе применения шагов рекурсии.

Рис. 13. Рекурсивная интерпретация списка

Рассмотрим два рекурсивных метода, один из которых распечатывает содержимое информационных полей линейного односвязного списка в прямом порядке (*Print_Front*), а второй – в обратном порядке (*Print_Back*):

```
public void Print_Front( Node first )          // распечатка содержимого списка
{                                              // в прямом направлении
  if  ( first != null )
  {
    Console.WriteLine( first.Info );
    Print_Front( first.Link )                  // хвостовая рекурсия
  }
}
```

```
public void Print_Back( Node first )            // распечатка содержимого списка
{                                                // в обратном направлении
  if  ( first != null )
  {
    Print_Back( first.Link )
    Console.WriteLine( first.Info );
  }
}
```

3.7. Эффективность рекурсивных вычислений

Так как для хранения фреймов активации рекурсивного метода используется стек, при обработке данных нерекурсивной природы следует использовать итеративные алгоритмы, не требующие дополнительного расхода памяти, если только рекурсивные алгоритмы не оказываются более ясными и понятными. Например, при вычислении значения факториала натурального числа N несомненно следует предпочесть итеративный метод.

Если метод содержит единственный рекурсивный вызов и он является последним действием метода, то говорят, что имеет место ***хвостовая рекурсия*** (*tail recursion*) (см. метод *Print_Front*, приведенный в разделе 1.3). Этот рекурсивный **вызов требует затрат на создание фреймов активации и запоминание** их в стеке. Когда рекурсивный вычислительный процесс доходит до условия завершения, выполняется серия возвратов, выталкивающих фреймы активации из стека. Если при наличии хвостовой рекурсии фреймы активации не используются для окончательных вычислений, следует предпочесть итеративную реализацию (см. метод *Print*, приведенный выше). Рекурсивный метод *Print_Back*, не содержащий хвостовой рекурсии, имеет более эффективную реализацию, чем итеративный метод, выполняющий те же действия.

3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Лабораторная работа выполняется в четыре этапа.

1. *Ознакомительный этап*, на котором студент изучает методы представления и обработки динамических списковых структур.
2. *Подготовительный этап*, на котором составляются алгоритм и программа работы с динамическими списковыми структурами.
3. *Лабораторный этап*, на котором производится отладка программы.
4. *Составление отчета*. Отчет должен содержать:
   - название лабораторной работы и текст варианта задания;
   - листинг программы.

Для сдачи отчета необходимо продемонстрровать работу программы.

Требования к программе:

- корректность исходных данных, вводимых с клавиатуры, должна контролироваться программой;
- работа программы должна иллюстрироваться средствами графического и текстового вывода;
- должен быть разработан “дружественный” пользовательский интерфейс, обеспечивающий многократное выполнение программы с переопределением исходных данных, без выхода из оболочки программы.

4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем заключаются особенности работы с автоматической памятью?

2. Из каких частей состоит рекурсивное определение?

3. Из каких частей состоит рекурсивный алгоритм?

4. Что такое прямая и косвенная рекурсия?

5. Сравните итеративную и рекурсивную организацию вычислительного процесса. Чем они отличаются?

6. Что такое бесконечная рекурсия? Какова причина ее возникновения?

7. Что такое уровень рекурсии? Что такое глубина рекурсии?

8. Каким образом используются фреймы активации в процессе работы рекурсивного алгоритма?

9. Что такое бектрекинг? Как реализуются алгоритмы поиска с возвратом?

10. В чем особенности рекурсивной реализации комбинаторных алгоритмов?

11. Что такое “хвостовая” рекурсия? Почему при реализации алгоритмов ее следует избегать?

5 ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

1. Реализуйте рекурсивный алгоритм для нахождения биномиальных коэффициентов, пользуясь следующим определением

2. Реализуйте рекурсивный алгоритм для вычисления квадратного

корня числа. В качестве исходных данных используйте тройку чисел N, A и E, где N - число, из которого требуется извлечь квадратный корень, A - приближенное значение корня, E - допустимая ошибка результата.

3. Реализуйте рекурсивный алгоритм подсчета Q_{MN} - числа способов, с помощью которых можно представить целое число M в виде суммы, каждое слагаемое которой не превосходит N. Воспользуйтесь следующим определением

$$Q_{MN} = \begin{cases} 1, & \text{если } M = 1, \text{ при всех } N, \\ 1, & \text{если } N = 1, \text{ при всех } M, \\ Q_{MM}, & \text{если } M < N, \\ 1 + Q_{M,M-1}, & \text{если } M = N, \\ Q_{M,N-1} + Q_{M-N,N}, & \text{если } M > N. \end{cases}$$

4. Реализуйте рекурсивный алгоритм, который определяет N членов ряда Фибоначчи.

5. Функция Аккермана определяется следующим образом:

A(0,y) = y+1;
A(x,0) = A(x-1, 1);
A(x,y) = A(x-1, A(x, y-1)).

Здесь x и y – целые неотрицательные числа. Определим модулярную функцию Аккермана как A mod m, где параметр m - целое неотрицательное число. Реализуйте рекурсивный алгоритм построения таблицы значений этой функции.

6. Реализуйте рекурсивный алгоритм вычисления последовательности *n* вложенных корней

$$x(i)=\sqrt{m+\sqrt{m+\ldots+\sqrt{m}}}, \quad m\geq 0,\ i=\overline{1,\ldots,n}.$$

7. Реализуйте алгоритм вычисления суммы с использованием рекурсии

$$y(x,i)=\sin x+\sin(\sin x)+\ldots+\sin(\sin\ldots(\sin x)),\quad i=\overline{1,\ldots,n}.$$

8. Реализуйте алгоритм вычисления цепной дроби для произвольного значения *n >= 0* с помощью рекурсии

$$\cfrac{1}{1+\cfrac{1}{3+\cfrac{1}{5+\cfrac{1}{7+\cfrac{1}{9+\cfrac{1}{11+\ldots}}}}}}$$

9. Реализуйте рекурсивный алгоритм вычисления суммы n первых членов ряда

$$1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+\ldots+\frac{x^{n-1}}{(n-1)!}+\ldots$$

10. Реализуйте рекурсивный алгоритм вычисления суммы n первых членов ряда

$$x+\frac{x^3}{2!}+\frac{x^5}{3!}+\frac{x^7}{4!}+\ldots+\frac{x^{2n-1}}{n!}+\ldots$$

11. Пусть в алгебраической записи выражения имеется единственная операция – умножение, обозначаемое обычным образом (два сомножителя следуют непосредственно друг за другом). Выражение состоит из строки символов и скобок- ограничителей: ( ) [ ] { }. Реализуйте рекурсивный алгоритм, выполняющий проверку соответствия открывающих и закрывающих скобок. Например, запрещены выражения вида (ab] или a(b[c])d].

12. Реализуйте рекурсивный алгоритм решения уравнений вида F(x) =x методом простых итераций.

13. Реализуйте рекурсивный алгоритм перевода натурального числа из десятичной системы счисления в систему счисления с основанием N, N <

10.

14. Реализуйте рекурсивный алгоритм, распечатывающий десятичную запись заданного целого положительного числа N как последовательности цифр.

15. Задана строка S из N символов. Реализуйте рекурсивный алгоритм проверки, является ли симметричной часть строки, начинающаяся i-м и заканчивающаяся j-м ее элементом.

16. Реализуйте рекурсивный алгоритм, распечатывающий различные представления заданного натурального числа N в виде суммы не менее двух натуральных слагаемых. Представления, отличающиеся лишь порядком слагаемых, различными не считаются.

17. Реализуйте рекурсивный алгоритм, распечатывающий по одному разу в лексикографическом порядке все последовательности длины N, составленные из натуральных чисел 1..K.

18. Реализуйте рекурсивный алгоритм, распечатывающий все возрастающие последовательности длины N, элементами которых являются натуральные числа 1..K.

19. Реализуйте рекурсивный алгоритм, распечатывающий все перестановки натуральных чисел 1..N по одному разу. Перестановка - последовательность длины N, в которую каждое из этих чисел входит по одному разу.

20. Функция f(n) определена для целых положительных чисел следующим образом:

$$f(n) = \begin{cases} 1, & n = 1, \\ \sum_{i=2}^{n} f(n \text{ DIV } i), & n >= 2. \end{cases}$$

Реализуйте рекурсивный алгоритм вычисления этой функции.

21. Реализуйте рекурсивный алгоритм вычисления значения суммы для заданного целого n

$$\sum_{i1=1}^{n} \sum_{i2=1}^{n} \dots \sum_{in=1}^{n} \frac{1}{i1 + i2 + \dots + in}.$$

22. Реализуйте рекурсивный алгоритм вычисления определителя заданной матрицы, пользуясь формулой разложения по первой строке:

$$\det A = \sum_{k} (-1)^{k+1} a_{1k} * \det(B_k),$$

где матрица B_k получается из A вычеркиванием первой строки и k-го столбца.

23. Реализуйте рекурсивный алгоритм вычисления значения многочлена вида

$$P_n(x) = a_0 * x^n + a_1 * x^{n-1} + a_2 * x^{n-2} + \dots + a_{n-1} * x + a_n$$

в заданной целочисленной точке согласно формуле

$$P_n(x) = x * P_{n-1}(x) + a_n, \quad \text{где}$$

$$P_{n-1}(x) = a_0 * x^{n-1} + a_1 * x^{n-2} + \dots + a_{n-2} * x + a_{n-1}.$$

24. Реализуйте рекурсивный алгоритм нахождения наибольшего общего делителя последовательности N натуральных чисел.

25. Реализуйте рекурсивный алгоритм нахождения суммы делителей данного натурального числа.

26. Реализуйте рекурсивный алгоритм двоичного поиска элемента с заданным значением в упорядоченном целочисленном массиве.

ЗАКЛЮЧЕНИЕ

Методические указания “Рекурсивные алгоритмы в C#” посвящены рассмотрению структур данных и алгоритмов, которые являются фундаментом современной методологии разработки программ.

Методические указания подробно описывают организацию рекурсивного вычислительного процесса и методы реализации рекурсивных алгоритмов.

Теоретический материал иллюстрируется большим количеством примеров программ, реализующих рекурсивные алгоритмы обработки данных на языке C#.

Логическим завершением методических указаний являются контрольные вопросы и индивидуальные задания для самостоятельной работы студентов.

Вопросы, имеющие практическое значение для студентов при выполнении домашних заданий и лабораторных работ, освещены с необходимой для использования полнотой.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Н. Вирт. Алгоритмы и структуры данных: пер. с англ. / Н.Вирт. Изд. 2-е, испр. – СПб.: Невский диалект, 2005. – 352 с.
2. Ахо А. Структуры данных и алгоритмы: пер. с англ. / А.Ахо, Д. Хопкрофт, Д.Ульман. – М.: Вильямс, 2016. – 400 с.
3. Павловская Т.А. C#. Программирование на языке высокого уровня: учебник для вузов / Т.А.Павловская. – Спб.: Питер, 2014. – 432 с.
4. Фаронов В.В. Создание приложений с помощью C#. Руководство программиста / В.В. Фаронов. – М.: Эксмо, 2008. – 576 с.
5. Биллиг В.А.Основы программирования на C#: учебное пособие / В.А.Биллиг – М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2009. – 483 с.
6. Шилдт, Герберт. C# 3.0: руководство для начинающих: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2009. – 688 с.
7. Шилдт, Герберт. C# 4.0: Полное руководство: Пер. с англ. – М.: Издательский дом «Вильямс», 2011. – 1056 с.
8. Гросс, Кристиан. C# 2008 и платформа NET 3.5 Framework: базовое руководство: Пер. с англ. – М.: Издательский дом «Вильямс», 2009. – 480 с.
9. Петцольд, Чарльз. Программирование для Microsoft Windows 8. Пер. с англ. – СпБ.: Издательский дом «Питер», 2014. – 1008 с.
10. Стиллмен, Эндрю, Грин, Дженнифер. Изучаем C#. Пер. с англ. – СпБ.: Издательский дом «Питер», 2017. – 816 с.
11. Кнут Д. Искусство программирования для ЭВМ: в 3 т. Т 1: пер. с англ. / Д.Кнут. – М.: Вильямс, 2000. – 720 с.

Методические материалы

РЕКУРСИВНЫЕ АЛГОРИТМЫ В C#

Методические указания

Составитель ***Симонова Елена Витальевна***

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА»
(Самарский университет)
443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34.

Изд-во Самарского университета.
443086 Самара, Московское шоссе, 34.