Рубанцев Валерий

Программирование в среде PascalABC.NET

Функциональное программирование на паскале

PascalABC.NET

Валерий Рубанцев

Функциональное программирование на *паскале*

От автора

Многие современные языки программирования поддерживают несколько парадигм, почему и называются **мультипарадигменными**. Например, на *C++*, *Си-шарпе*, *Яве*, *Питоне*, *паскале PascalABC.NET* можно писать программы в процедурном, объектно-ориентированном и функциональном стиле.

Функциональное программирование насчитывает уже несколько десятилетий, но только сравнительно недавно стало одной из ведущих парадигм программирования.

Программы, написанные в функциональном стиле более надёжные, короткие и понятные. Их проще писать и отлаживать. Они превращают императивное программирование в **декларативное**. Программы, написанные в императивном стиле, показывают, **как нужно что-то сделать**. А программы, написанные в декларативном стиле, показывают, **что мы хотим сделать**. Декларативное программирование помогает избежать множества мелких подробностей. Например, методы расширения для последовательностей *OrderBy* и *OrderByDescebding* умеют сортировать элементы по заданному условию. Теперь нет необходимости подробно расписывать функции сортировки. Весь код занимает единственную строку в программе.

Функции

Функции имеются во многих языках программирования, в том числе и в *паскале*. Однако функции в функциональном программировании существенно отличаются от них и имеют свойства *математических функций*.

Определение функции в *паскале* имеет вид:

```
function имя(список формальных параметров): тип возвращаемого значения;
```

Простейшая **функция Summa**, которая вычисляет и возвращает сумму двух целых чисел может быть такой:

```
function Summa(a,b : integer): integer;
begin
   Result := a + b;
end;
```

При вызове функции вместо формальных параметров подставляются фактические параметры, или **аргументы**:

```
var sum:= Summa(5,7);
```

Печатаем результат, возвращаемый функцией:

```
Println('Сумма чисел = ', sum);
```

Сколько бы раз вы ни вызывали функцию *Summa* с этими аргументами, вы всегда будете получать число 12.

Математические функции действуют точно так же. Если они получают одни и те же аргументы, то **всегда** возвращают один и тот же результат.

Аргументами могут быть не только литералы, но и **переменные** совместимого типа:

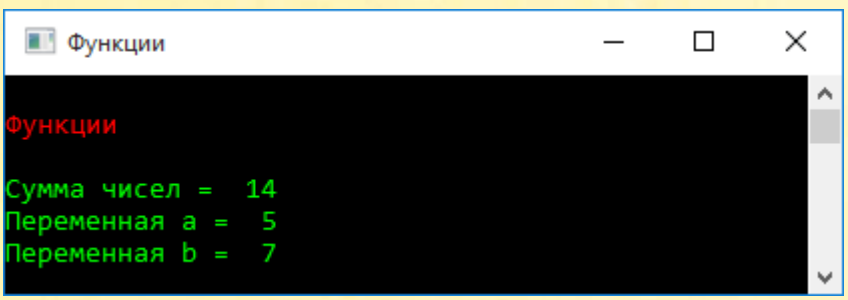
```
var a:= 5;
var b:= 7;
var sum:= Summa(a,b);
Println('Сумма чисел = ', sum);
Println('Переменная a = ', a);
Println('Переменная b = ', b);
Println;
```

После вызова функции *Summa* **значения переменных не изменились**:

Функция *Summa* получает **значения** переменных, но не сами эти переменные. В теле функции создаются **локальные** переменные с именами параметров, которые получают переданные аргументы. Внутри функции можно как угодно изменять значения этих переменных, и при этом значения внешних переменных не изменятся:

```
function Summa(a,b : integer): integer;
begin
    a := b;
    Result := a + b;
end;
```

Внутри функции переменная **a** получила значение переменной **b**, то есть 7, поэтому функция вернула число 14. Однако внешние переменные **a** и **b** остались неизменными:



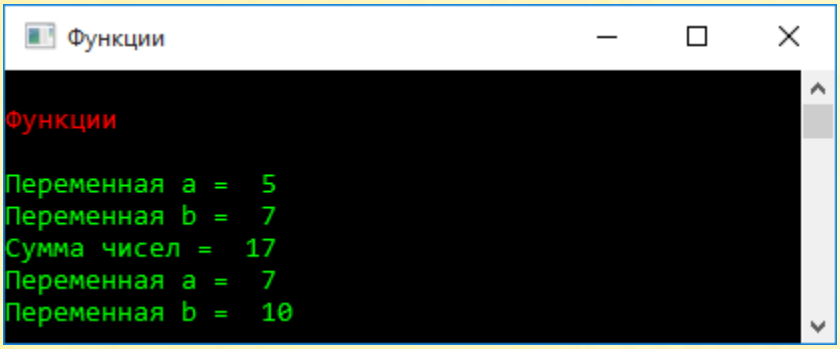
Но мы можем передать параметр **по ссылке**. Для этого перед идентификатором параметра нужно поставить ключевое слово **var**. Во второй версии нашей функции пусть будут **параметры-переменные**:

```
function Summa2(var a,b : integer): integer;
begin
    a := b;
    b:= 10;
    Result := a + b;
end;
```

Теперь при вызове функция получает не значения локальных переменных, а внешние переменные a и b:

```
var a:= 5;
var b:= 7;
Println('Переменная a = ', a);
Println('Переменная b = ', b);
var sum:= Summa2(a,b);
Println('Сумма чисел = ', sum);
Println('Переменная a = ', a);
Println('Переменная b = ', b);
```

Как вы видите на рисунке, значения внешних переменных **изменились**:



Внешние переменные могут называться иначе, чем параметры функции, но результат будет точно таким же:

```
var m:= 5;
var n:= 7;
Println('Переменная m = ', m);
Println('Переменная n = ', n);
var sum:= Summa2(m,n);
```

```
Println('Сумма чисел = ', sum);
Println('Переменная m = ', m);
Println('Переменная n = ', n);
```

Итак, функция *Summa2* изменила значения внешних переменных. Это действие называется **побочным эффектом**. Функции, не имеющие побочного эффекта, называются **чистыми**. Все математические функции относятся к чистым.

В большой программе может быть несколько функций, которые изменяют значения внешних переменных. Работу таких программ очень сложно контролировать, поскольку любая из этих функций может работать неверно.

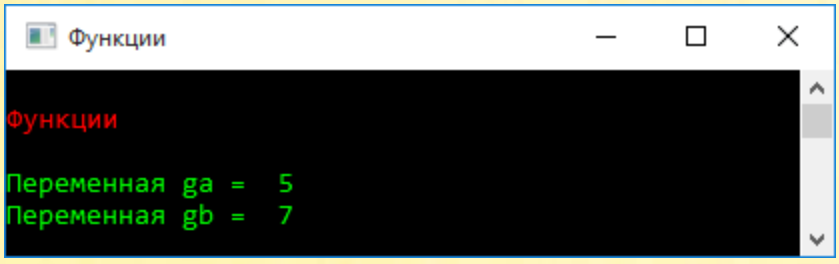
Функция может вообще не иметь параметров и использовать **глобальные переменные** как аргументы.

Объявим 2 глобальные переменные:

```
var ga, gb: integer;
```

В главном блоке присвоим этим переменным значения и напечатаем их:

```
ga:= 5;
gb:= 7;
Println('Переменная ga = ', ga);
Println('Переменная gb = ', gb);
```



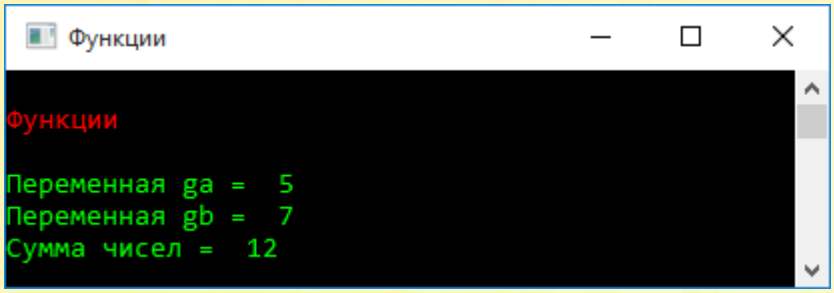
Пишем новую **функцию**:

```
function Summa3(): integer;
begin
```

```
    Result := ga + gb;
end;
```

Она не имеет параметров, а для вычисления значения использует глобальные переменные программы.

Мы присвоили глобальным переменным значения 5 и 7, так что сумму наша функция вычислила верно:



Так как значения глобальных переменных можно изменить в главном блоке программы и в любой функции, то функция *Summa3* вполне вероятно будет возвращать **различные** значения при нескольких вызовах. Такое поведение функций в функциональном программировании не допускается.

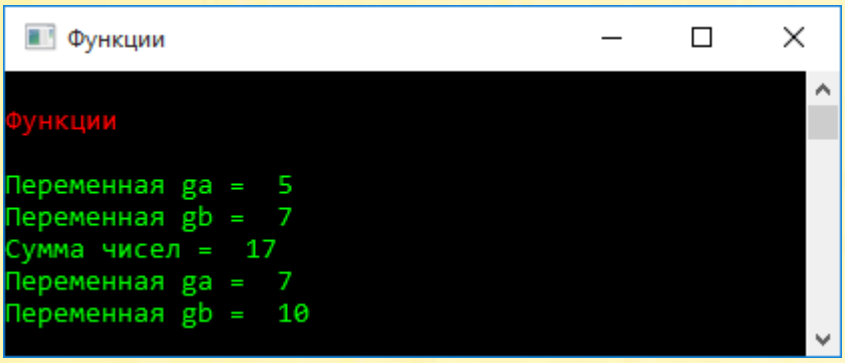
В функции Summa4 мы можем изменить значения глобальных переменных:

```
function Summa4(): integer;
begin
    ga := gb;
    gb:= 10;
    Result := ga + gb;
end;
```

Как вы помните, до вызова этой функции они имели значения 5 и 7, а после вызова – 7 и 10:

```
var sum:= Summa4();
Println('Сумма чисел = ', sum);
Println('Переменная ga = ', ga);
```

```
Println('Переменная gb = ', gb);
```



В чистых функциях нельзя использовать изменяемые значения. Например, ввод данных пользователя или показания системных часов. Понятно, что в таких случаях возвращаемое значение может изменяться при каждом вызове функции.

В функциональном программировании уже существующие данные **не изменяются**! К ним либо добавляются данные, либо создаются новые. При каждом изменении объекта программы создаётся **новый** экземпляр. К примеру, в *паскале* нельзя изменить последовательность. Всякий раз будет создаваться новая последовательность, которая может включать элементы исходной последовательности. Поэтому в функциональном программировании все переменные могут только единственный раз получить значение, то есть должны быть **константами**.

В мультипарадигменных языках программирования, к которым относится и *паскаль*, невозможно использовать только чистые функции, но вполне можно программировать в **функциональном стиле**.

Функции высших порядков

Функции высших порядков – это функции, которые могут принимать другие функции как аргументы.

В *паскале* в функции можно передавать **переменные процедурных типов**:

```
type
   //объявление процедурного типа:
```

```
   FiboFact = function(n: integer): decimal;
```

```
function GetFF(ff: FiboFact; n: integer): decimal;
begin
    Result:= ff(n);
end;

Println('Факториал числа 27 равен  ', GetFF(Factorial, 27));
Println('Число Фибоначчи 139 равно ', GetFF(Fibonacci, 139));
```

Или непосредственно ссылки на функции:

```
lstInt := BubbleSort(lstNum, CompareLen&<integer>);
```

Если код функции короткий, то её можно определить непосредственно при вызове функции высшего порядка:

```
lstNum.Sort(function(n1, n2) ->
            begin
               Result := n1 - n2;
            end
           );
```

Такие функции называются анонимными.

Для упрощения кода анонимные функции часто заменяют лямбда-выражениями:

```
lstNum.Sort((n1, n2) -> n1 - n2);
```

Лямбда-выражения – это реализация математических функций в программировании.

В книге подробно, с многочисленными примерами, рассказывается:

- о процедурном типе в *паскале*
- об анонимных функциях
- о лямбда-выражениях
- о генерировании последовательностей
- об языке интегрированных запросов *LINQ*
- обо всех встроенных методах расширения для последовательностей
- обо всех дополнительных методах расширения в языке *PascalABC.NET*
- о разработке собственных методов расширения

Книга адресуется: школьникам, изучающим *PascalABC.NET* на уроках или самостоятельно, учителям информатики, любителям программирования.

Валерий Рубанцев

Условные обозначения, принятые в книге:

Дополнение или замечание

Требование или указание

Исходный код

Задание для самостоятельного решения

Заголовок проекта:

Проект

Исходные коды всех проектов находятся в папке **_Projects**

Оглавление

Процедурный тип

Процедурные типы – это ссылочные типы, определяемые пользователем. Переменная процедурного типа (процедурная переменная) хранит ссылку на функцию, процедуру, метод класса, экземпляра класса или структуры, то есть их начальный адрес в памяти. **Множественные процедурные переменные** содержат целый **список вызовов**, в котором записаны ссылки на методы.

Образно процедурные переменные можно назвать **представителями** (в языке *Си-шарп* они называются *делегатами*) метода или методов в программе. При этом процедурная переменная предъявляет к методам определённые требования: они должны иметь тот же тип возвращаемого значения и такую же сигнатуру.

Чтобы пользоваться процедурной переменной в своей программе, вы должны:

- Объявить процедурный тип
- Объявить переменную процедурного типа и присвоить ей значение
- Вызвать процедуру, функцию или метод через процедурную переменную

Описание процедурного типа

Перед использованием процедурный тип должен быть **объявлен**:

```
type
имя_типа = procedure | function(список параметров): typeRet;
```

Здесь:

- **typeRet** – тип возвращаемого значения

Объявление процедурного типа можно сравнить с заголовком процедуры/функции: оно также содержит **тип возвращаемого значение** и **список формальных параметров** (сигнатуру).

Тип возвращаемого значения и сигнатура процедурного типа задают «формат» процедур/функций, на которые он может ссылаться.

Процедурный тип

Чтобы яснее понять работу с процедурными переменными, напишем новый проект. Пусть нам нужны **факториалы** и **числа Фибоначчи**. Функции для их вычисления могут быть такими:

```
// ФАКТОРИАЛ
function Factorial(n: integer): decimal;
begin
   if (n < 1) then
   begin
      Result := 1;
      exit;
   end;
   var res: decimal := 1;
   for var i := 2 to n do
   begin
      try
         res := res * i;
      except
         Println('Слишком большое число!');
         Result := 0;
         exit;
      end;
   end;
   Result := res;
end;

 // ЧИСЛО ФИБОНАЧЧИ
function Fibonacci(n: integer): decimal;
begin
```

```
    if (n <= 2) then
    begin
       Result := 1;
       exit;
    end;

    var x: decimal := 1;
    var y: decimal := 1;
    var res: decimal := 0;
    for var i := 3 to n do
    begin
       try
          res := x + y;
          x := y;
          y := res;
       except
          Println('Слишком большое число!');
          Result := 0;
          exit;
       end;
    end;
    Result := res;
end;
```

Обе функции возвращают значение типа **decimal** и имеют единственный параметр типа **integer**. Совершенно очевидно, что процедурный тип для этих функций также должен иметь:

- тип возвращаемого значения – *decimal*
- единственный параметр типа *integer*

Самое трудное в объявлении процедурного типа – придумать для него красивый **идентификатор**:

```
// процедурный тип:
type FiboFact = function(n: integer): decimal;
```

Объявление и создание процедурной переменной

После описания процедурного типа вы можете объявить переменную этого типа и инициализировать её.

Объявление процедурной переменной ничем не отличается от объявления переменных других типов: сначала указывается *идентификатор* переменной, а затем *процедурный тип*:

```
var имя_переменной : ProcType;
```

Поскольку процедурный тип – это **ссылочный** тип, то переменная после её объявления имеет значение *nil,* то есть не ссылается ни на один объект.

Вернёмся в наш проект и **объявим переменную**:

```
// объявляем переменную процедурного типа:
var fifa: FiboFact;
```

Пока этой переменной не присвоено значение, пользоваться ею нельзя. Поэтому мы должны **присвоить** ей **значение** подходящего типа. Например, ссылку на функцию *Factorial*:

```
// присваиваем ей значение:
fifa := Factorial;
```

С тем же успехом мы могли бы инициализировать её функцией *Fibonacci*.

Как обычно, **объявление переменной можно сочетать с её инициализацией**:

```
// создаём переменную процедурного типа:
var fifa2: FiboFact := Fibonacci;
```

После создания процедурной переменной ей можно **присвоить новое значение** подходящего типа:

```
fifa := Fibonacci;
```

Когда процедурная переменная получает значение, она ссылается на процедуру/функцию, то есть, по сути, процедурная переменная – это лишь **псевдоним** (или «представитель») процедуры/функции, имеющей сигнатуру, которую определяет процедурный тип.

> Такое образное определение процедурной переменной верно, только когда она ссылается на **единственную** процедуру/функцию. Как вы увидите дальше, процедурная переменная содержит список вызовов процедур/функций и тогда **представляет** сразу несколько процедур/функций.

Процедура/функция, на которую ссылается процедурная переменная, может быть также **методом класса** и **методом экземпляра класса**. Метод класса указывается после имени класса, а метод экземпляра класса – после имени объекта.

Чтобы присвоить процедурной переменной ссылку на **методы**, опишем 2 новых **класса** с методами *Factorial* и *Fibonacci*:

```
type
   FF = class
   public
      // ФАКТОРИАЛ
      class function Factorial(n: integer): decimal;
      begin
         if (n < 1) then
         begin
            Result := 1;
            exit;
         end;
         var res: decimal := 1;
         for var i := 2 to n do
         begin
            try
               res := res * i;
```

```
            except
               Println('Слишком большое число!');
               Result := 0;
               exit;
            end;
         end;
         Result := res;
      end;

      // ЧИСЛО ФИБОНАЧЧИ
      class function Fibonacci(n: integer): decimal;
      begin
         if (n <= 2) then
         begin
            Result := 1;
            exit;
         end;

         var x: decimal := 1;
         var y: decimal := 1;
         var res: decimal := 0;
         for var i := 3 to n do
         begin
            try
               res := x + y;
               x := y;
               y := res;
            except
               Println('Слишком большое число!');
               Result := 0;
               exit;
            end;
         end;
         Result := res;
      end;

   end;

   FF2 = class
   public
      // ФАКТОРИАЛ
      function Factorial(n: integer): decimal;
      begin
         if (n < 1) then
         begin
```

```
            Result := 1;
            exit;
        end;
        var res: decimal := 1;
        for var i := 2 to n do
        begin
            try
                res := res * i;
            except
                Println('Слишком большое число!');
                Result := 0;
                exit;
            end;
        end;
        Result := res;
    end;

    // ЧИСЛО ФИБОНАЧЧИ
    function Fibonacci(n: integer): decimal;
    begin
        if (n <= 2) then
        begin
            Result := 1;
            exit;
        end;

        var x: decimal := 1;
        var y: decimal := 1;
        var res: decimal := 0;
        for var i := 3 to n do
        begin
            try
                res := x + y;
                x := y;
                y := res;
            except
                Println('Слишком большое число!');
                Result := 0;
                exit;
            end;
        end;
        Result := res;
    end;
end;
```

Методы класса *FF* принадлежат **классу**, а методы класса *FF2* – **объекту**.

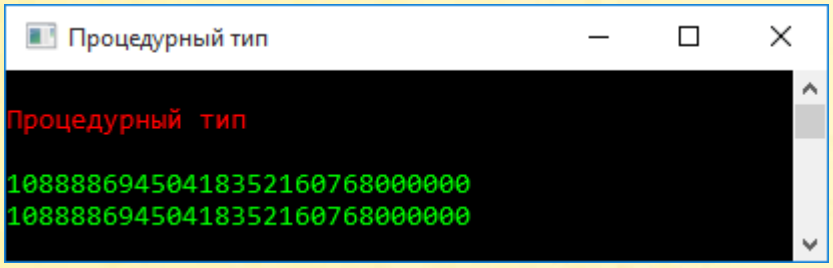
Метод класса можно присвоить процедурной переменной без создания объекта:

```
// метод класса:
fifa := FF.Factorial;
Println(fifa(27));
```

А теперь создадим экземпляр класса *FF2* и присвоим переменной **fifa** ссылку на метод *Factorial* этого объекта:

```
// метод экземпляра класса:
var ffe := new FF2();
fifa := ffe.Factorial;
Println(fifa(27));
```

Оба метода возвращают одно и то же значение факториала:



Когда процедурная переменная представляет метод класса, то она хранит только ссылку на точку входа этого метода. Если же процедурная переменная представляет метод экземпляра, то она хранит также ссылку и на экземпляр его класса.

Процедурной переменной можно присвоить **лямбда-выражение**. О лямбда-выражениях речь пойдёт дальше.

Вызов функции через процедурную переменную

Если мы хотим вычислить факториал числа 27, то должны вызвать функцию *Factorial* с соответствующим аргументом:

```
var dm1 := Factorial(27);
```

Но если переменная **fifa** ссылается на функцию *Factorial*, то вместо имени этой функции мы можем поставить имя процедурной переменной:

```
var fifa: FiboFact := Factorial;
var dm2 := fifa(27);
```

Результат мы получим точно такой же, поскольку и в том, и в другом случае вызывается одна и та же функция *Factorial*.

Переменной процедурного типа можно присвоить ссылку на другую функцию:

```
fifa := Fibonacci;
dm2 := fifa(27);
```

И тогда она будет вызывать эту функцию.

Так с помощью **одной** процедурной переменной мы можем вызывать **разные** функции - и *Factorial*, и *Fibonacci*.

Список вызовов

Пока речь шла о представительстве единственной процедуры/функции, можно было и не упоминать о том, что каждая процедурная переменная хранит **список вызовов** (*invocation list*), то есть указателей на процедуры, функции и методы.

Начните новый проект и перепишите методы *Factorial* и *Фибоначчи* так, чтобы они не возвращали значение, а просто печатали его в консольном окне:

```
// ФАКТОРИАЛ
procedure Factorial(n: integer);
begin
    if (n > 1) then
    begin
        var res: decimal := 1;
        for var i := 2 to n do
        begin
            try
                res := res * i;
            except
                Println('Слишком большое число!');
                exit;
            end;
        end;
        Println(res);

    end
end;

// ЧИСЛО ФИБОНАЧЧИ
procedure Fibonacci(n: integer);
begin
    if (n > 2) then
    begin
        var x: decimal := 1;
        var y: decimal := 1;
        var res: decimal := 0;
        for var i := 3 to n do
        begin
            try
                res := x + y;
                x := y;
                y := res;
            except
                Println('Слишком большое число!');
                exit;
            end;
        end;
        Println(res);
    end
```

```
end;
```

Объявление процедурной переменной нужно изменить соответствующим образом:

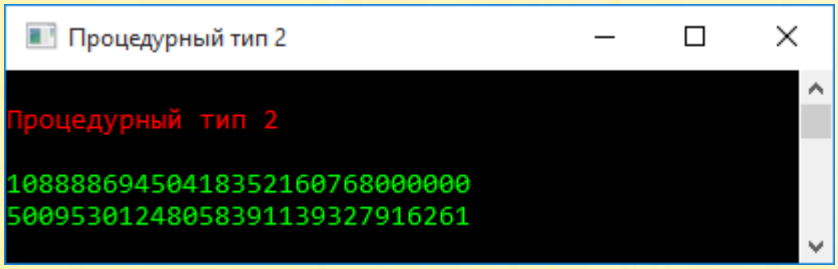
```
// процедурный тип:
type
    FiboFact = procedure(n: integer);
```

Объявим две процедурные переменные для этих процедур:

```
var fact : FiboFact := Factorial;
var fibo : FiboFact := Fibonacci;
```

Обе содержат указатель на **единственную** процедуру, и мы можем вызвать каждую переменную по отдельности с нужным аргументом:

```
fact(27);
fibo(139);
```



Но процедурные переменные можно **связывать** (объединять, комбинировать) друг с другом, получая новую - **множественную**, многоадресную процедурную переменную. Она объединяет списки вызовов других процедурных переменных таким образом, что получается единый список вызовов.

При вызове множественной процедурной переменной она последовательно вызывает процедуры/функции из общего списка.

Мы можем получить множественную (групповую) процедурную переменную с помощью метода **Combine** класса *System.Delegate*.

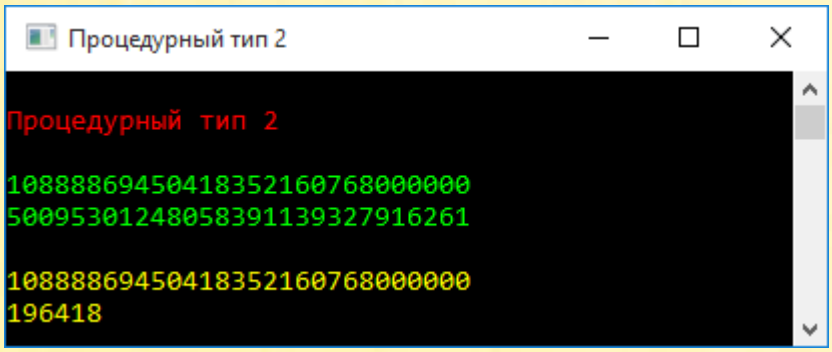
Он получает две процедурные переменные и возвращает новую процедурную переменную типа *Delegate*. Для конкретного случая **необходимо приведение типов**. Давайте объединим две процедурные переменные- *fact* и *fibo* - в одну, множественную процедурную переменную **fifa**:

```
var fifa := FiboFact(Delegate.Combine(fact, fibo));
```

Список вызовов переменной **fifa** состоит из списка вызовов первого аргумента, к которому дописан список вызовов второго аргумента, то есть при вызове переменной **fifa** сначала будут вызваны процедуры/функции/методы переменной **fact, а затем – переменной fibo**. В нашем случае – процедуры *Factorial* и *Fibonacci*. Выполним такой код:

```
Println;
Console.ForegroundColor := ConsoleColor.Yellow;
var fifa := FiboFact(Delegate.Combine(fact, fibo));
fifa(27);
```

С учётом предыдущего кода (зелёный цвет) мы получим такие результаты (жёлтый цвет):



Всё бы хорошо, но наши процедуры *Factorial* и *Fibonacci* при вызове требуют **аргумент**, который мы должны сообщить процедурной переменной. Но если в списке вызовов несколько процедур/функций/методов, то мы не можем передать

каждому из них свой аргумент – при вызове они получат **один и тот же** аргумент. Это значит, что не стоит создавать множественные процедурные переменные для подпрограмм с параметрами.

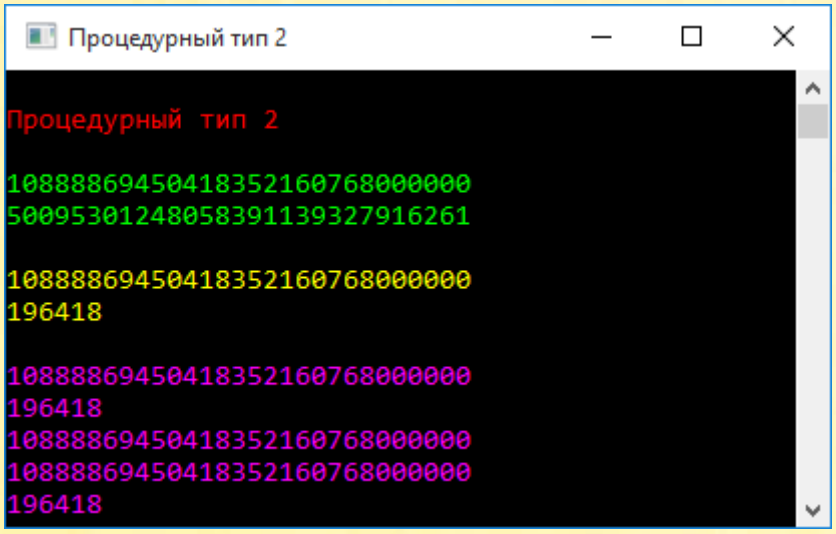
> Если процедура/функция/метод имеет **ссылочные** параметры, то их значение будет изменять каждый вызванный метод (процедура/функция) из списка.

Но давайте закроем на это глаза и воспользуемся более приятными свойствами множественных процедурных переменных, которые позволяют сцеплять списки любого числа процедурных переменных. Для этого нужно обратиться к перегруженному методу **Combine**, который принимает любое число переменных.

В нашем проекте мы можем комбинировать наши переменные как угодно, при этом они могут повторять любое число раз:

```
Println;
Console.ForegroundColor := ConsoleColor.Magenta;
fifa := FiboFact(Delegate.Combine(fact, fibo, fact, fact, fibo));
fifa(27);
```

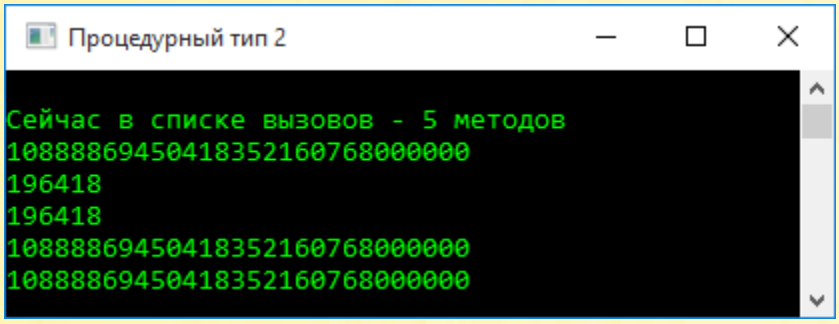
Запускаем приложение – и видим, что все процедуры вызваны верно:



Если одна из процедур генерирует **исключение**, то остальные процедуры в списке не вызываются, а исключение передаётся в метод, который вызвал процедурную переменную.

Число процедур/функций/методов в списке процедурной переменной можно узнать с помощью метода GetInvocationList:

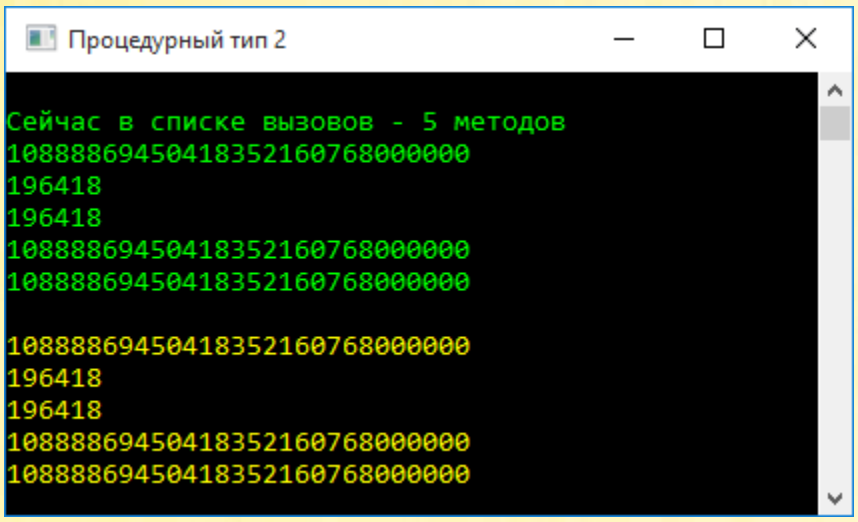
```
Println;
Console.ForegroundColor := ConsoleColor.Green;
fifa := fact + fibo + fibo + fact + fact;
var delegates := fifa.GetInvocationList();
Console.WriteLine('Сейчас в списке вызовов - {0} методов',
                  delegates.Length);
fifa(27);
```



Метод **GetInvocationList** возвращает массив переменных, в списке вызовов которых – единственный метод, поэтому общее число методов в точности равно числу элементов в массиве.

Вы даже можете самостоятельно вызвать процедуры/функции/методы из списка вызовов:

```
Println;
Console.ForegroundColor := ConsoleColor.Yellow;
foreach var d in delegates do
    d.DynamicInvoke(27);
```



Этот трюк позволяет произвольно вызывать процедуры/функции/методы из списка вызовов, то есть передавать им аргументы и получать возвращаемые значения.

Объявите новый процедурный тип для прежних версий методов:

```
FiboFact2 = function(n: integer): decimal;
```

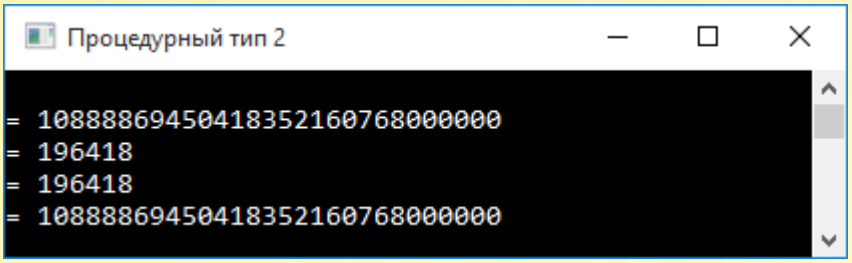
И добавьте к проекту, дополнив их имена *двойкой*, чтобы не смешивать с новыми версиями этих методов.

В **главном блоке** создаём множественную переменную **fifa2**, получаем её список вызовов и в цикле *foreach* (или *for*, или по индексу) извлекаем функции и вызываем их с нужным аргументом. Возвращаемое значение можно использовать, как угодно:

```
Println;
Console.ForegroundColor := ConsoleColor.White;
var fact2: FiboFact2 := Factorial2;
var fibo2: FiboFact2 := Fibonacci2;
var fifa2: FiboFact2 := fact2 + fibo2 + Fibonacci2 + Factorial2;
delegates := fifa2.GetInvocationList();
foreach var d in delegates do
begin
   var ff2 := FiboFact2(d);
```

```
   Println('=', ff2(27));
end;
```

Мы просто напечатали возвращаемые значения, но вы можете вычислить, например, их сумму.



Однако вспомним, что изначально процедуры *Factorial* и *Fibonacci* возвращали вычисленное значение, а в этом проекте мы сделали их «невозвращенцами». Здесь мы имеем ту же проблему, что и с параметрами. Групповая переменная может вернуть только одно значение – это значение последней подпрограммы в списке вызовов, все остальные значения будут проигнорированы. Если не печатать результаты на экране, то все подпрограммы, кроме последней, отработают зря.

Мы можем не только добавлять новые методы в список вызовов, но и **удалять** из него любой метод с помощью метода **Remove** класса *Delegate*.

Так как в списке вызовов подпрограммы могут повторяться, то из него будет удалено **последнее** вхождение списка заданной подпрограммы. Метод *Remove* вернёт новую переменную:

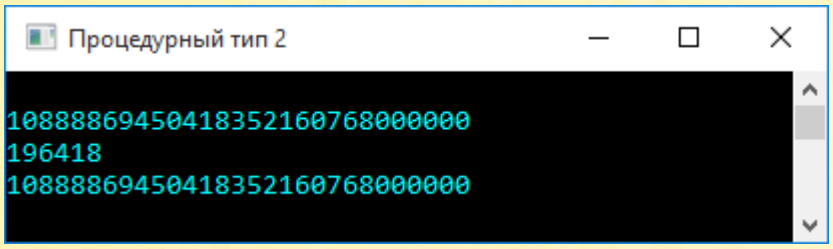
- С укороченным списком вызовов, если совпадение обнаружилось
- Ту же переменную, если совпадения не было
- *nil*, если в списке вызовов не осталось ни одного метода

Удалим из переменной **fifa** последние списки вызовов переменных **fibo** и **fact**:

```
Println;
Console.ForegroundColor := ConsoleColor.Cyan;
```

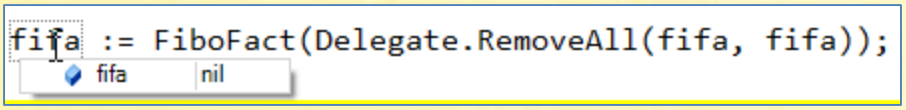
```
fifa := FiboFact(Delegate.Remove(fifa, fibo));
fifa := FiboFact(Delegate.Remove(fifa, fact));
fifa(27);
```

На рисунке видно, что удаление прошло успешно:



Метод RemoveAll удаляет *все* вхождения списка вызовов заданной переменной из списка вызовов множественной переменной.

Например, если из списка вызовов переменной fifa удалить весь его список вызовов, то она прекратит своё существование.



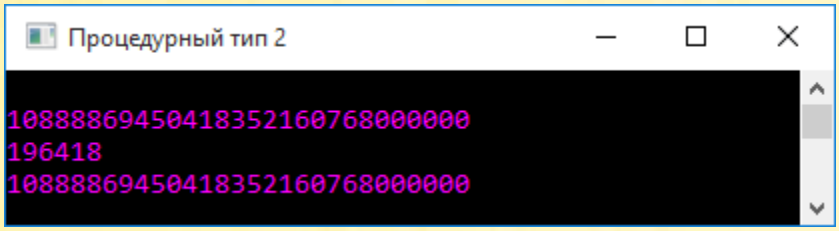
Групповую переменную можно получить и с помощью оператора +, который применяется к процедурным переменным одного типа.

Пример с этим оператором вы уже видели выше:

```
fifa := fact + fibo + fibo + fact + fact;
```

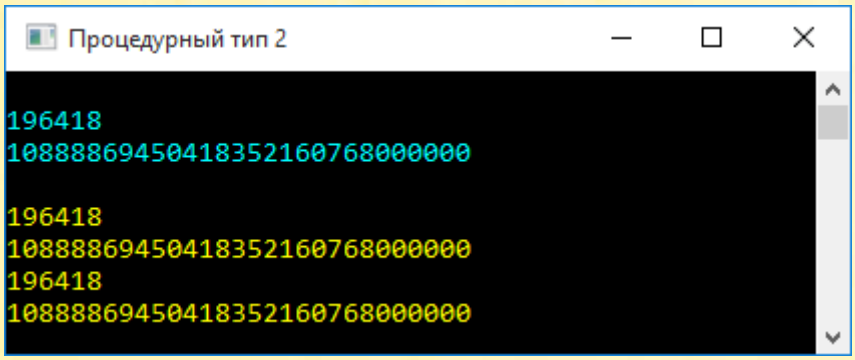
Оператор – удаляет списки вызовов указанных делегатов:

```
Println;
Console.ForegroundColor := ConsoleColor.Magenta;
fifa := fact + fibo + fibo + fact + fact;
fifa := fifa - fact - fibo;
fifa(27);
```



Оператор += позволяет добавлять к списку вызовов одной переменной списки вызовов других переменных или непосредственно однотипные функции/процедуры/методы:

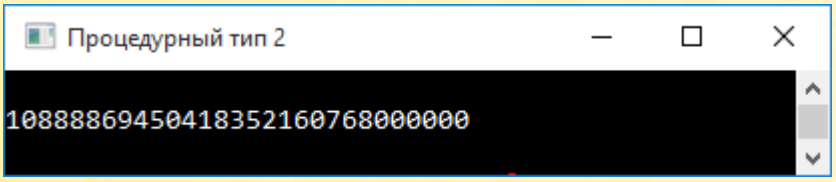
```
Println;
Console.ForegroundColor := ConsoleColor.Cyan;
fifa := nil;
fifa += Fibonacci;
fifa += Factorial;
Println;
fifa(27);

Println;
Console.ForegroundColor := ConsoleColor.Yellow;
fifa += fifa;
fifa(27);
```



Оператор -= удаляет из множественной переменной списки вызовов других переменных или вызовы отдельных методов:

```
Println;
Console.ForegroundColor := ConsoleColor.White;
fifa2 := Factorial2;
```

```
fifa2 += Fibonacci2;
fifa2 -= Fibonacci2;
Println(fifa2(27));
```



Обобщённые процедурные переменные

Мы можем объявить процедурный тип так, чтобы он имел возвращаемое значение и/или параметры произвольных типов:

```
type
    //объявление универсального типа-делегата:
    GetLength<T> = function(t: T): integer;
    GetLength<T, TResult> = function(t: T): TResult;
```

Такие процедурные типы называются обобщёнными.

Первый процедурный тип всегда возвращает значение типа *integer*, но может принимать 1 параметр любого типа. Второй процедурный тип отличается от первого только тем, что и тип возвращаемого значения может быть любым.

Благодаря обобщённым процедурным типам-делегатам мы получили в своё распоряжение огромное число типов функций, которые можно с ними использовать.

Начните новый проект и напишите несколько функций, возвращающих значение и получающих 1 аргумент. Например, такие:

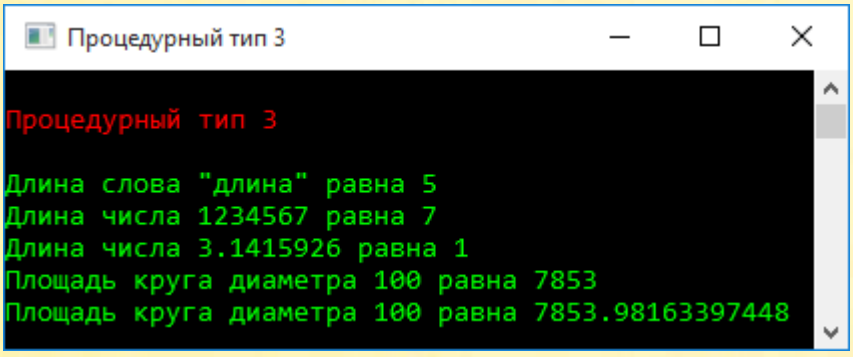
```
function GetStringLength(str: string): integer;
begin
    Result:= str.Length;
end;
```

```
function GetIntLength(num: integer): integer;
begin
    if (num = 0) then
    begin
       Result:= 1;
       exit;
    end;

    var log := Log10(Abs(num));
    Result:= Trunc(log) + 1;
end;

function GetDoubleLength(num: double): integer;
begin
    if (num = 0) then
    begin
       Result:= 1;
       exit;
    end;

   var log := Log10(Abs(Trunc(num)));
   var res := Trunc(log);
   Result:= res + 1;
end;

function GetCircleArea(d: double): integer;
begin
   var area := PI * d * d / 4;
   var res := Trunc(area);
   Result:= res;
end;

function GetCircleAreaD(d: double): double;
begin
    var res := PI * d * d / 4;
   Result:= res;
end;
```

Теперь перейдём в главный блок, где объявим и инициализируем процедурные переменные для наших функций. Обобщённые переменные создаются точно так же, как и обычные:

```
begin
    //заголовок окна:
    Console.Title := 'Процедурный тип 3';
    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Процедурный тип 3');
    Console.ForegroundColor := ConsoleColor.Green;
    Println;

    var getLength := GetStringLength;
    Console.WriteLine('Длина слова "длина" равна ' +
                      getLength('длина'));

    var getLength2 := GetIntLength;
    Console.WriteLine('Длина числа 1234567 равна ' +
                      getLength2(1234567));

    var getLength3 := GetDoubleLength;
    Console.WriteLine('Длина числа 3.1415926 равна ' +
                      getLength3(3.1415926));

    getLength3 := GetCircleArea;
    Console.WriteLine('Площадь круга диаметра 100 равна ' +
                      getLength3(100));

    var getLength4 := GetCircleAreaD;
    Console.WriteLine('Площадь круга диаметра 100 равна ' +
                      getLength4(100));

    Println;
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Результаты работы процедурных переменных вы можете видеть на рисунке:



Обобщённые процедурные типы *Action<T>* и *Func<T>*

Всем хороши обобщённые процедурные типы, но всё-таки их нужно объявлять. Однако в языке *паскаль* имеются готовые обобщённые процедурные типы Action<T> и Func<T>, которые могут избавить нас от лишних усилий.

Обобщённые процедурные типы Action<T> ссылаются на *процедуры*, а универсальные процедурные типы Func<T> - на *функции*, возвращающие значение произвольного типа.

Имеется целый набор процедурных типов Action<T> и Func<T> с различным числом параметров – *от 0 до 16*:

```
Action()
Action<T>
Action<T1, T2>
. . .

Func<TResult>(
Func<T, TResult>
Func<T1, T2, TResult>
. . .
```

T1, T2, … - тип параметра
TResult – тип возвращаемого значения

Сравним наши усилия в предыдущем проекте с меньшими трудозатратами в новом. Скопируйте в него все функции из проекта *Процедурные типы 3* и добавьте ещё одну процедуру, чтобы можно было задействовать обобщённый процедурный тип Action<> (обобщённый процедурный тип Func<> также можно использовать для ссылки на процедуры, но такая практика не рекомендуется):

```
procedure GetCircleAreaA(d: double);
begin
   var area := Math.Round((PI * d * d / 4),2);
   Console.WriteLine('Площадь круга диаметра {0}100 равна {1}', d,
area);
end;
```

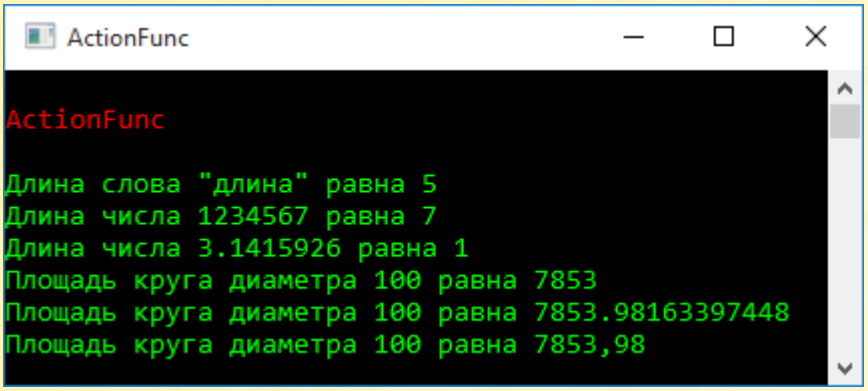
В главном блоке мы просто описываем процедурные переменные, соответствующие универсальным типам *Func* и Action:

```
uses System;


 function GetStringLength(str: string): integer;
 begin
     Result:= str.Length;
 end;

 function GetIntLength(num: integer): integer;
 begin
     if (num = 0) then
     begin
        Result:= 1;
        exit;
     end;

     var log := Log10(Abs(num));
     Result:= Trunc(log) + 1;
 end;

 function GetDoubleLength(num: double): integer;
 begin
     if (num = 0) then
     begin
        Result:= 1;
        exit;
     end;

    var log := Log10(Abs(Trunc(num)));
    var res := Trunc(log);
    Result:= res + 1;
 end;
```

```
function GetCircleArea(d: double): integer;
begin
   var area := PI * d * d / 4;
   var res := Trunc(area);
   Result:= res;
end;

function GetCircleAreaD(d: double): double;
begin
    var res := PI * d * d / 4;
   Result:= res;
end;

procedure GetCircleAreaA(d: double);
begin
  var area := Math.Round((PI * d * d / 4),2);
  Console.WriteLine('Площадь круга диаметра {0}100 равна {1}', d,
area);
end;



begin
    //заголовок окна:
    Console.Title := 'ActionFunc';
    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('ActionFunc');
    Console.ForegroundColor := ConsoleColor.Green;
    Println;

    var getLength := GetStringLength;
    Console.WriteLine('Длина слова "длина" равна ' +
                      getLength('длина'));

    var getLength2 := GetIntLength;

    Console.WriteLine('Длина числа 1234567 равна ' +
                      getLength2(1234567));

    var getLength3 := GetDoubleLength;
    Console.WriteLine('Длина числа 3.1415926 равна ' +
                      getLength3(3.1415926));
```

```
    getLength3 := GetCircleArea;
    Console.WriteLine('Площадь круга диаметра 100 равна ' +
                      getLength3(100));

    var getLength4 := GetCircleAreaD;
    Console.WriteLine('Площадь круга диаметра 100 равна ' +
                      getLength4(100));

    var CircleArea := GetCircleAreaA;
    CircleArea(100);


    Println;
    Console.ForegroundColor := ConsoleColor.Red;
end.
.
```

Рисунок показывает, что замена типов прошла успешно!



Обобщённые процедурные типы также могут быть множественными.

Добавьте к проекту ещё одну процедуру того же типа *Action<double>*, что и *GetCircleAreaA*:

```
procedure GetQuadratAreaA(d: double);
begin
   var area := Math.Round((d * d),2);
   Console.WriteLine('Площадь квадрата со стороной {0} равна {1}', d,
area);
end;
```

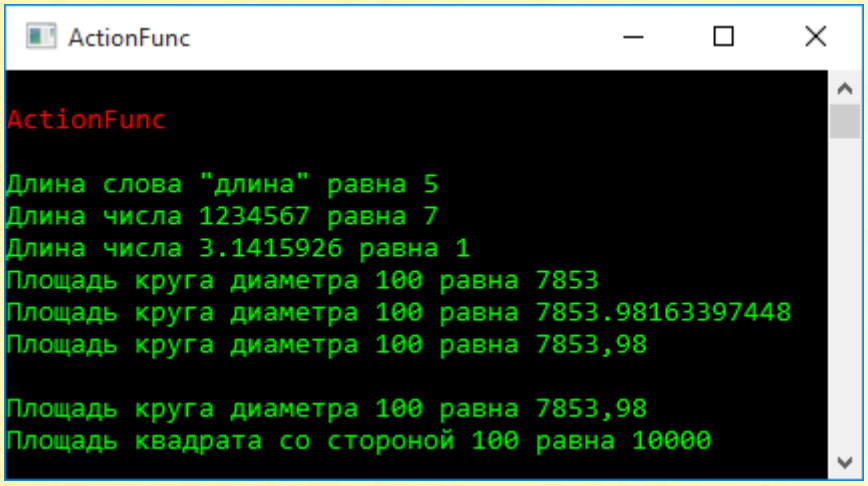
Только вместо площади круга она вычисляет и печатает **площадь квадрата**.

Допишите в **главный блок** ещё несколько строк:

```
Println;
Console.ForegroundColor := ConsoleColor.Green;
CircleArea += GetQuadratAreaA;
CircleArea(100);
```

Здесь мы создаём множественную переменную **CircleArea** со списком вызовов, состоящим из двух процедур – *GetCircleAreaA* и *GetQuadratAreaA*.

Запускаем приложение – **зелёные** строчки на рисунке подтверждают наше предположение: переменная **CircleArea** последовательно вызывает обе процедуры:



Итак, обобщённые процедурные типы **Func<T>** и **Action<T>** позволяют нам не объявлять новые типы, что очень удобно.

Синонимы для процедурных типов

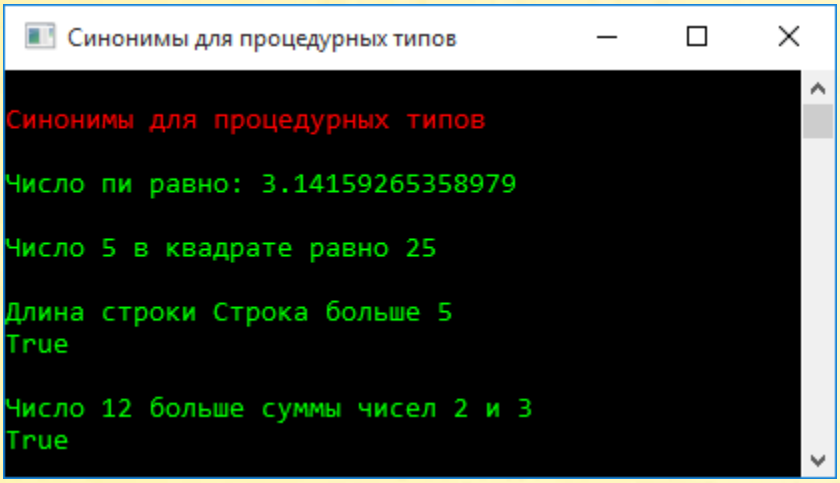
В *паскале* для некоторых процедурных типов введены **синонимы**, облегчающие их использование:

```
Action0 → Action
Action<T> → Action<T>
Action2<T1,T2> → Action<T1,T2>
Action3<T1,T2,T3> → Action<T1,T2,T3>

Func0<Res> → Func<Res>
Func<T,Res> → Func<T,Res>
Func2<T1,T2,Res> → Func<T1,T2,Res>
Func3<T1,T2,T3,Res> → Func<T1,T2,T3,Res>

IntFunc → Func<integer,integer>
RealFunc → Func<real,real>
StringFunc → Func<string,string>
Predicate<T> → Predicate<T>
Predicate2<T1,T2> → Predicate<T1,T2>
Predicate3<T1,T2,T3> → Predicate<T1,T2,T3>
```

Парочка примеров:

```
var fres: Func0<double> := () -> PI;
Console.WriteLine('Число пи равно: ' + fres);
Println;

var fint: IntFunc := n ->  n*n;
var n:= 5;
Console.WriteLine('Число {0} в квадрате равно {1}', n, fint(n));
Println;


var pr2: Predicate2<String, integer>:= (s, n) -> s.Length > n;
var s:= 'Строка';
Console.WriteLine('Длина строки {0} больше {1}', s, 5);
Println(pr2(s, 5));
Println;
```

```
var pr3: Predicate3<integer, integer, integer>:= (n1,n2,n3) ->
                                                    n1 > n2 + n3;
Console.WriteLine('Число {0} больше суммы чисел {1} и {2}',
                  12, 2, 3);
Println(pr3(12,2,3));
```



Процедурные переменные как параметры

Процедурные переменные можно использовать как **параметры** процедур и функций - точно так же, как и другие объекты ссылочных типов.

Функции, которые передаются как аргументы, называются **функциями обратного вызова** (*callback*). Процедурные переменные служат прекрасным средством для организации таких функций в языке *паскаль*.

Начните новый проект, скопируйте в него функции *Factorial* и Fibonacci, а затем объявите процедурный тип для них:

```
type
   //объявление процедурного типа:
   FiboFact = function(n: integer): decimal;
```

Теперь опишем новую функцию, которая принимает функцию ff типа *FiboFact* и целое число n:

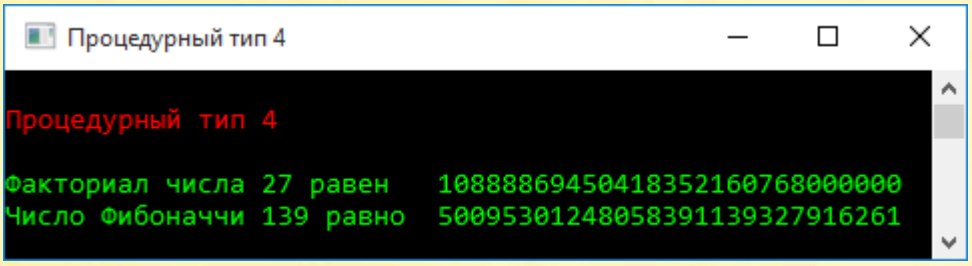
```
function GetFF(ff: FiboFact; n: integer): decimal;
begin
    Result:= ff(n);
end;
```

Целое число мы используем для передачи в функцию, на которую ссылается переменная *ff*, для вычисления факториала или числа Фибоначчи.

В главном блоке мы передаём функции *GetFF* ссылку на нужную функцию, а также аргумент для неё:

```
Println('Факториал числа 27 равен  ', GetFF(Factorial, 27));
Println('Число Фибоначчи 139 равно ', GetFF(Fibonacci, 139));
```

В первом случае выполняется функция Factorial с аргументом 27, во втором – функция Fibonacci с аргументом 139:



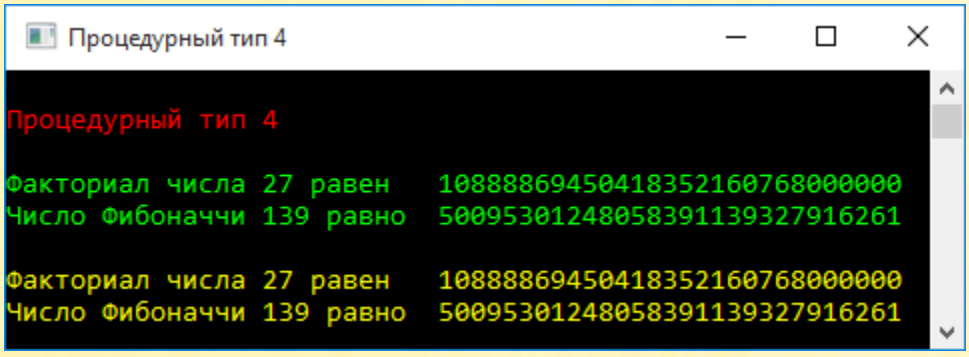
Мы можем обойтись и без объявления процедурного типа, а для типа параметра указать процедурный тип Func:

```
function GetFFF(ff: Func<integer, decimal>; n: integer): decimal;
begin
    Result:= ff(n);
end;
```

Не забывайте, что тип **последнего** параметра в угловых скобках процедурного типа *Func* – это тип **возвращаемого значения**. Все остальные типы определяют сигнатуру делегируемого метода (процедуры или функции).

В **главном блоке** новая функция *GetFFF* вызывается точно так же:

```
Println;
Console.ForegroundColor := ConsoleColor.Yellow;
Println('Факториал числа 27 равен  ', GetFFF(Factorial, 27));
Println('Число Фибоначчи 139 равно ', GetFFF(Fibonacci, 139));
```



Проект *Пузырьковая сортировка*

Метод пузырьковой сортировки медленный, но очень простой, поэтому применяется исключительно в учебных целях. У нас цели именно такие, поэтому в новом приложении мы будем сортировать данные любых типов. Нас вполне удовлетворят строковые и числовые данные, что, впрочем, ничуть не мешает вам отсортировать и что-нибудь другое.

Итак, функция пузырьковой сортировки должна быть **универсальной** – это очевидно, поскольку она должна уметь сортировать списки объектов *разных типов*. Менее очевиден вопрос: как именно следует сортировать списки? – Обычно числа сортируют в порядке увеличения их значений, а слова – в лексикографическом порядке. Но, возможно, вам потребуется список, в котором порядок следования элементов противоположный – от б*о*льших значений к меньшим.

Например, в каких-нибудь статистических исследованиях или при составлении обратного словаря русского языка. В некоторых случаях очень удобен словарь, в котором слова располагаются по длине. При разгадывании кроссвордов он был бы очень кстати! Короче говоря, было бы совсем неплохо, если бы одна и та же функция умела сортировать списки по-разному. Научить её этому можно с помощью процедурной переменной, которую мы передадим функции сортировки при её вызове:

```
// ФУНКЦИЯ СОРТИРОВКИ
function BubbleSort<T>(lst:List<T>; comp: Func<T, T, boolean>):
List<T>;
begin
   var res := new List<T>(lst);
   var n := lst.Count - 1;
   var sorted := true;
   while (sorted) do
   begin
      sorted := false;
      for var i := 0 to n-1 do
      begin
         if (comp(res[i], res[i+1])) then
         begin
            (res[i], res[i+1]) := (res[i+1], res[i]);
            sorted := true;
         end;
      end;
   end;
   Result:= res;
end;
```

Процедурная переменная **comp** сравнивает два соседних элемента списка *lst* типа *T* (точнее, его копии **res**) и возвращает *true*, если они нарушают правильный порядок следования элементов в списке. Если это так, то функция сортировки переставляет их местами.

В качестве аргумента можно передавать функции сортировки любые функции сравнения, которые совпадают с типом *Func<T, T, boolean>*. Это значит, что мы можем изменять поведение функции сортировки с помощью соответствующих функций сравнения.

В простейшем случае можно сортировать элементы так, как это принято для их типа *T*:

```
function Compare<T>(t1, t2: T): boolean;
begin
   var res := (IComparable(t1)).CompareTo(t2) > 0;
   Result:= res;
end;
```

Но для начала нам нужно создать **списки** элементов числового и строкового типа. Здесь важно знать меру, чтобы не потерять веру в метод пузырьковой сортировки. Я думаю, что сотни элементов в каждом списке вполне достаточно, чтобы ожидание ещё не стало утомительным.

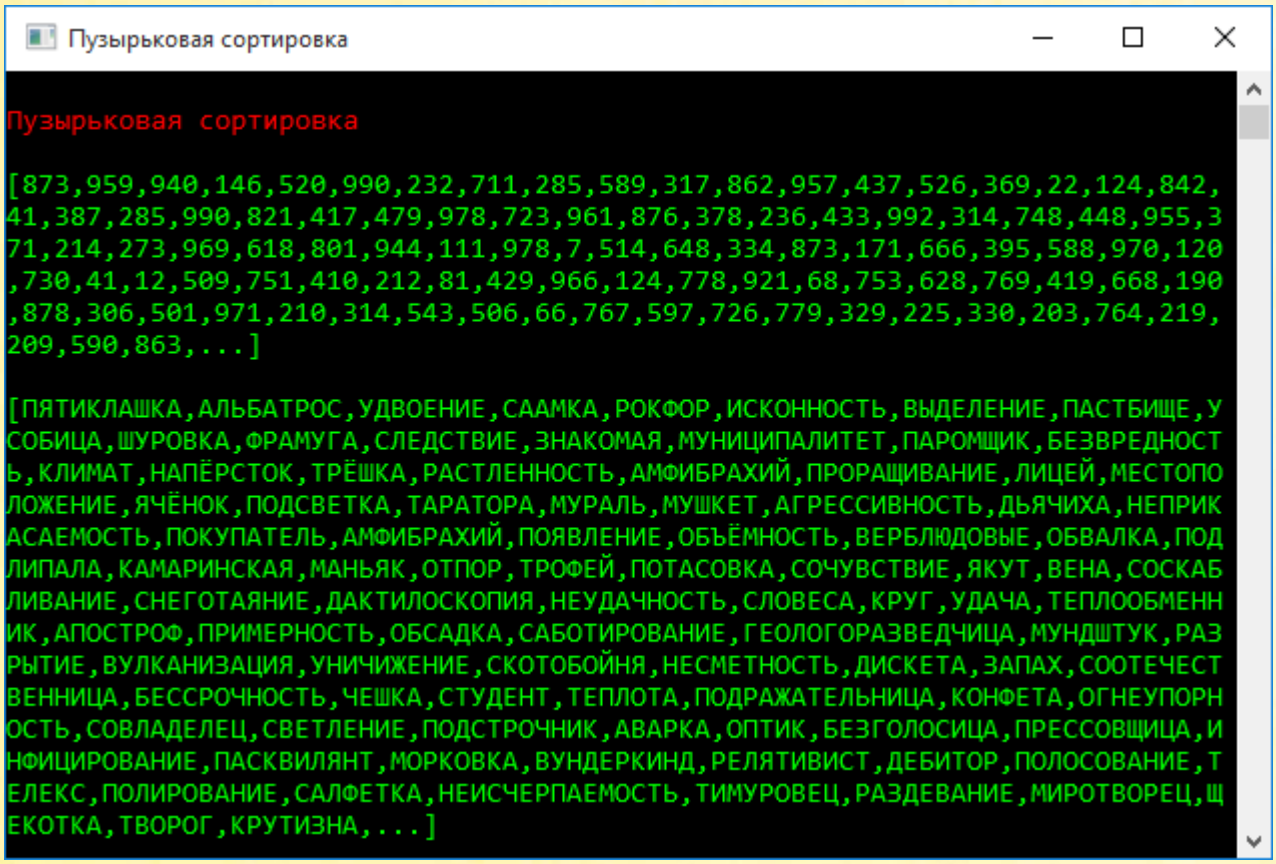
Сотню случайных чисел мы легко получим от функции *SeqRandom*, а вот со словами труднее. Мы загрузим словарь Ожегова и случайно выберем из него сотню слов:

```
begin
    //заголовок окна:
    Console.Title := 'Пузырьковая сортировка';
    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Пузырьковая сортировка');
    Console.ForegroundColor := ConsoleColor.Green;
    Println;

    var lstNum := SeqRandom(100, 1, 1000).ToList;
    Println(lstNum);
    Println;

    var lstStrAll := ReadAllLines('OSH-W97.txt').ToList();
    var lstStr := new List<string>();
    for var i := 0 to 100-1 do
    begin
        var ind := PABCSystem.Random(0, lstStrAll.Count-1);
        lstStr.Add(lstStrAll[ind]);
    end;
    Println(lstStr);
```

```
    Println;
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Полученные списки мы печатаем в консольном окне.

Глядя на рисунок, трудно не заметить, что и числа, и слова располагаются в списке именно в случайном порядке, чего мы и добивались:

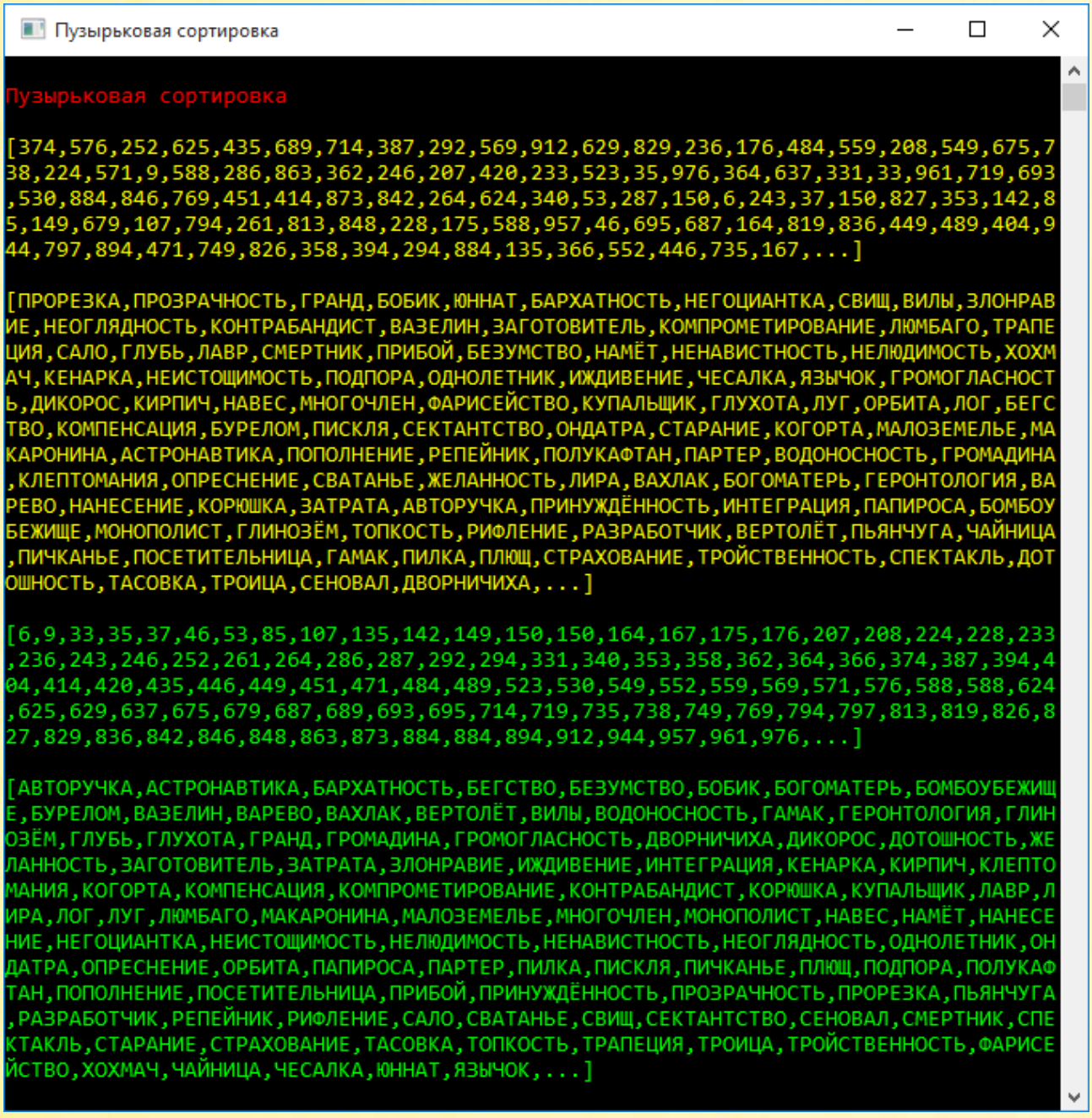


Пришло время отсортировать оба списка в общепринятом порядке. Для этого мы вызываем функцию сортировки *BubbleSort*, передавая ей в качестве аргументов списки и ссылку на функцию сравнения Compare<T>:

```
Console.ForegroundColor := ConsoleColor.Green;
Println;
Console.ForegroundColor := ConsoleColor.Green;
var lstInt := BubbleSort(lstNum, Compare&<integer>);
Println(lstInt);
```

```
Println;
var lstStr2 := BubbleSort(lstStr, Compare&<string>);
Println(lstStr2);
```

На рисунке жёлтым цветом показано содержимое списков до сортировки, а зелёным – после. Убеждаемся, что числа располагаются согласно величине, а слова – согласно алфавиту:



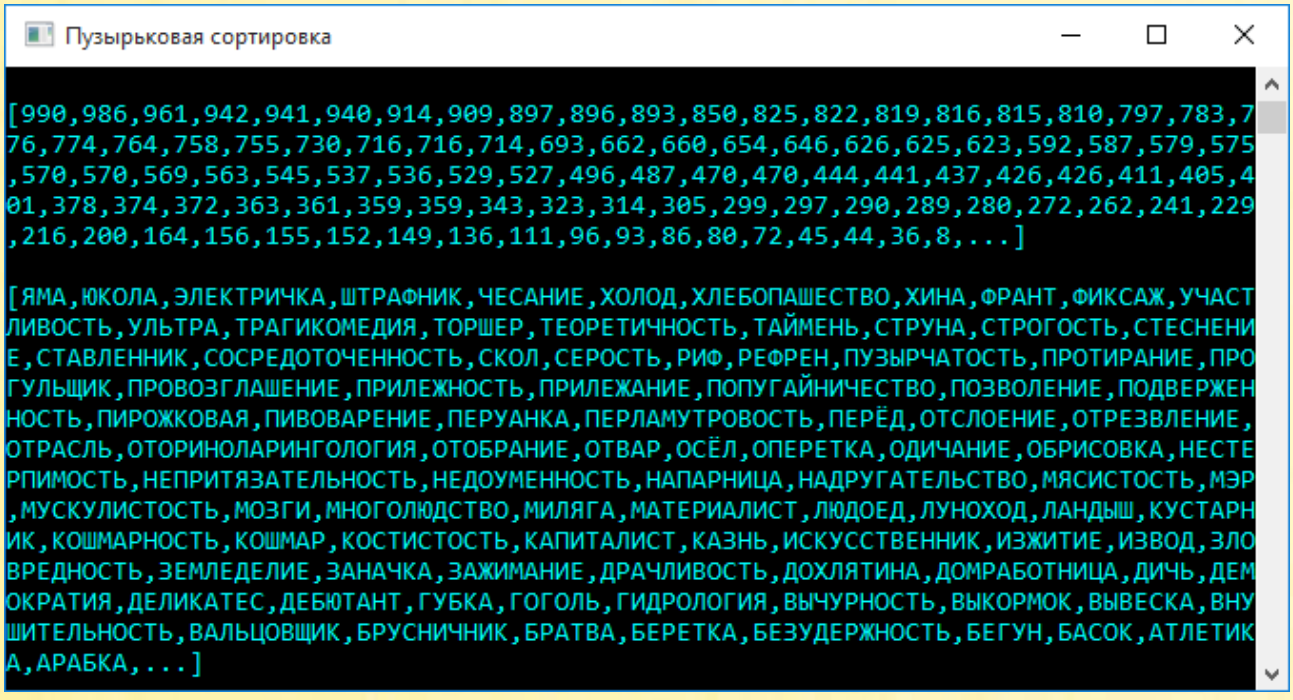
Функция для **обратной сортировки** отличается от функции прямой сортировки только тем, что мы изменили оператор сравнения:

```
// ОБРАТНАЯ СОРТИРОВКА
function CompareReversed<T>(t1, t2: T): boolean;
begin
   var res := (IComparable(t1)).CompareTo(t2) < 0;
   Result:= res;
end;
```

В **главном блоке** на этот раз мы сортируем списки шиворот-навыворот:

```
// обратная сортировка:
Println;
Console.ForegroundColor := ConsoleColor.Cyan;
lstInt := BubbleSort(lstNum, CompareReversed&<integer>);
Println(lstInt);

Println;
lstStr2 := BubbleSort(lstStr, CompareReversed&<string>);
Println(lstStr2);
```

Списки отсортированы:



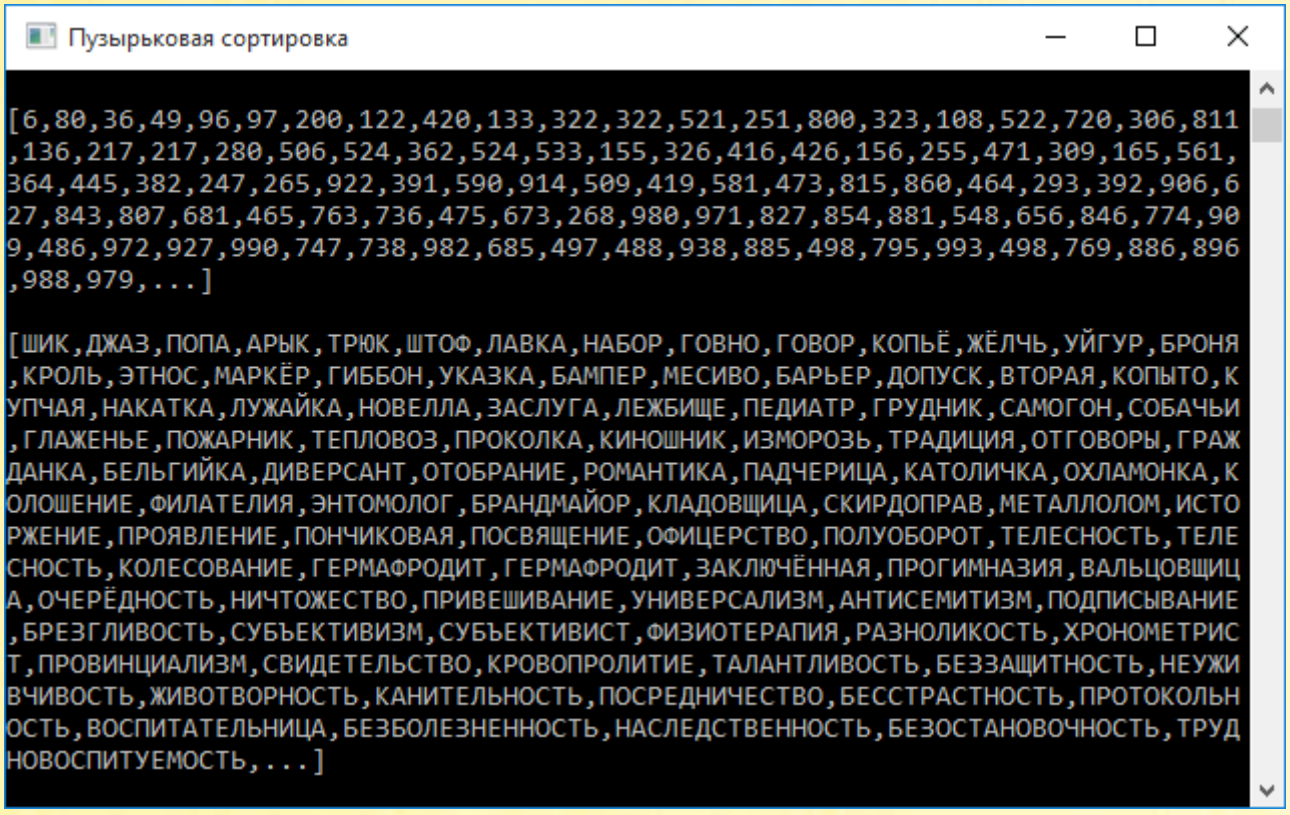
Теперь отсортируем элементы списков согласно **сумме их символов**, то есть числа - по сумме их цифр, а слова – по сумме кодов их букв:

```
// СОРТИРОВКА ПО СУММЕ
function CompareSum<T>(t1, t2: T): boolean;
begin
     var arr := t1.ToString().ToCharArray();
     var sum1 := 0;
     foreach var i in arr do
         sum1 += ord(i);

     arr := t2.ToString().ToCharArray();
     var sum2 := 0;
     foreach var i in arr do
        sum2 += ord(i);

     var res := sum1 > sum2;
     Result:= res;
end;
```

В **главный блок** дописываем новые строки:

```
// сортировка по сумме:
Console.ForegroundColor := ConsoleColor.Gray;
Println;
lstInt := BubbleSort(lstNum, CompareSum&<integer>);
Println(lstInt);

Println;
lstStr2 := BubbleSort(lstStr, CompareSum&<string>);
Println(lstStr2);
```

И получаем на выходе причудливо **отсортированные списки**:



> Такая сортировка на первый взгляд кажется несколько искусственной, но, например, в нумерологии и в теории чисел часто оперируют суммой цифр чисел, а сумма букв, составляющих слова, иногда используется в словесных задачах.

И наконец, мы отсортируем списки по **длине элементов**. Для чисел эта сортировка бесполезна, так как они и без того сортируются именно так, а для слов она вполне уместна и разумна.

Здесь важно учесть, что слова равной длины должны располагаться в алфавитном порядке:

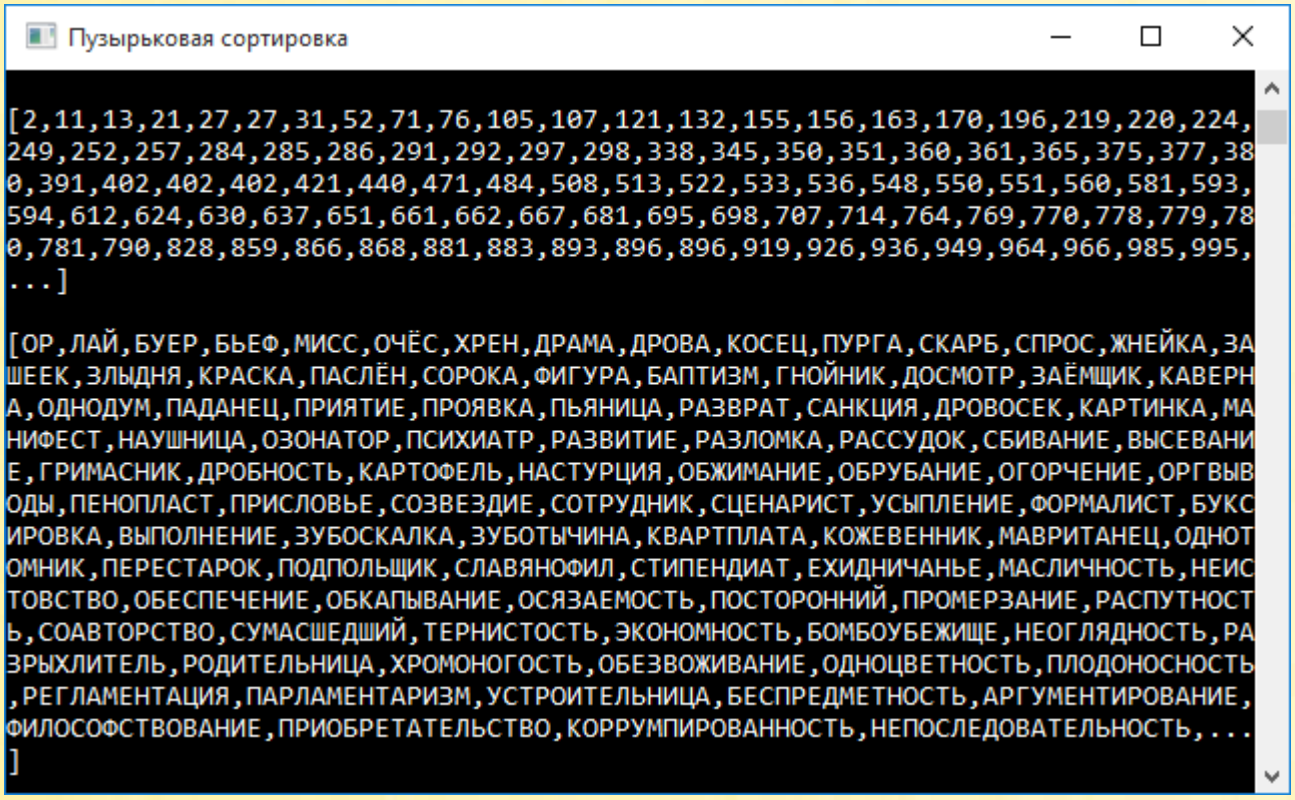
```
// СОРТИРОВКА ПО ДЛИНЕ
function CompareLen<T>(t1, t2: T): boolean;
begin
   var len1 := t1.ToString().Length;
   var len2 := t2.ToString().Length;

   var res:= true;
```

```
    if (len1 = len2) then
        res := (IComparable(t1)).CompareTo(t2) > 0
    else
        res := len1 > len2;

      Result:= res;
end;
```

Возвращаемся в главный блок и пишем последнюю порцию кода:

```
// сортировка по длине:
Console.ForegroundColor := ConsoleColor.White;
Println;
lstInt := BubbleSort(lstNum, CompareLen&<integer>);
Println(lstInt);

Println;
lstStr2 := BubbleSort(lstStr, CompareLen&<string>);
Println(lstStr2);
```

И опять оба списка отсортировались успешно:



И так, отправляя методу сортировки ссылку на метод сравнения, вы можете располагать элементы списка и вдоль, и поперёк. И при этом метод сортировки остаётся неизменным. Этот приём широко используется во многих коллекциях для сортировки элементов.

Анонимные функции

Вы, должно быть, обратили внимание, что все функции сравнения очень короткие и не используются самостоятельно – они нужны только для передачи в функцию сортировки. Например, *функция для прямой сортировки*, состоит, по сути, из единственной строки:

```
// ЕСТЕСТВЕННАЯ СОРТИРОВКА
function Compare<T>(t1, t2: T): boolean;
begin
   var res := (IComparable(t1)).CompareTo(t2) > 0;
   Result:= res;
end;
```

В *паскале* можно создавать **анонимные функции**. У них нет имени. Это просто блок кода, который, как обычно, заключается в операторные скобки, а ему предшествует ключевое слово *function*, после которого в скобках перечисляются формальные параметры.

Продолжим пузырьковую сортировку в новом проекте. В него нужно скопировать весь код из предыдущего проекта, за исключением кода главного блока после создания списков чисел и слов.

Начнём с естественной сортировки списков. Для этого мы передавали функции **BubbleSort<T>** ссылку на функцию *Compare<T>:*

```
var lstInt := BubbleSort(lstNum, Compare&<integer>);
var lstStr2 := BubbleSort(lstStr, Compare&<string>);
```

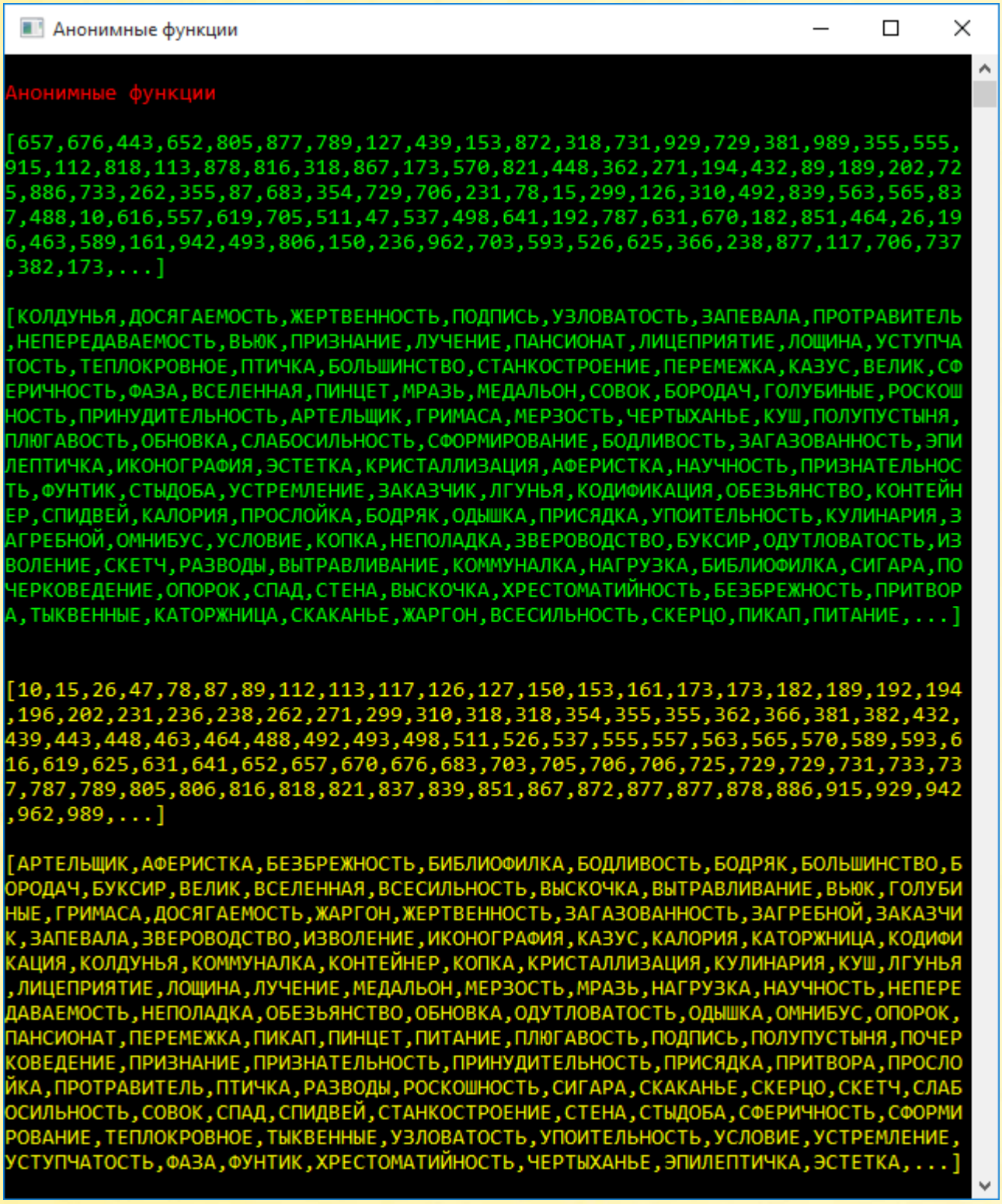
Поскольку эта функция одноразовая, то мы можем не писать новую функцию, а создать процедурную переменную нужного типа непосредственно в главном блоке, присвоив ей ссылку на встроенный блок кода – анонимную функцию.

Так мы создаём две процедурные переменные – compi и comps – из анонимных функций, а затем передаём их функции сортировки как аргументы:

```
//естественная сортировка:
Println;
Console.ForegroundColor := ConsoleColor.Yellow;
var compi: Func<integer, integer, boolean>  := function(n1, n2) ->
                                               begin
                                                  Result:= n1 > n2;
                                               end;
var lstInt := BubbleSort&<integer>(lstNum, compi);
Println(lstInt);

Println;
var comps: Func<string, string, boolean>  := function(s1, s2) ->
                                             begin
                                                Result:=
                            (IComparable(s1)).CompareTo(s2) > 0;
                                             end;

var lstStr2 := BubbleSort&<string>(lstStr, comps);
Println(lstStr2);
```

Тип возвращаемого значения указывать не нужно, поскольку он однозначно определяется компилятором по типу последнего параметра в процедурном типе *Func* или по типу возвращаемого значения в объявлении процедурного типа.

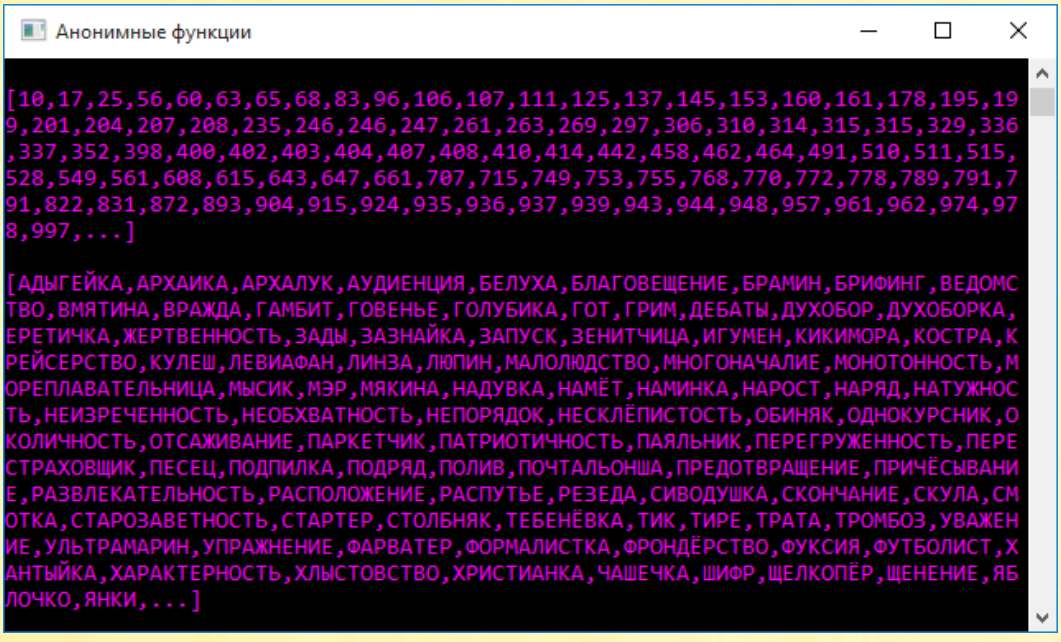
На рисунке отчётливо видно, что процедурные переменные исполняют встроенный код, который действует точно так же, как и именованные функции.


Анонимные функции
Анонимные функции
[657,676,443,652,805,877,789,127,439,153,872,318,731,929,729,381,989,355,555,
915,112,818,113,878,816,318,867,173,570,821,448,362,271,194,432,89,189,202,72
5,886,733,262,355,87,683,354,729,706,231,78,15,299,126,310,492,839,563,565,83
7,488,10,616,557,619,705,511,47,537,498,641,192,787,631,670,182,851,464,26,19
6,463,589,161,942,493,806,150,236,962,703,593,526,625,366,238,877,117,706,737
,382,173,...]
[КОЛДУНЬЯ,ДОСЯГАЕМОСТЬ,ЖЕРТВЕННОСТЬ,ПОДПИСЬ,УЗЛОВАТОСТЬ,ЗАПЕВАЛА,ПРОТРАВИТЕЛЬ
,НЕПЕРЕДАВАЕМОСТЬ,ВЬЮК,ПРИЗНАНИЕ,ЛУЧЕНИЕ,ПАНСИОНАТ,ЛИЦЕПРИЯТИЕ,ЛОЩИНА,УСТУПЧА
ТОСТЬ,ТЕПЛОКРОВНОЕ,ПТИЧКА,БОЛЬШИНСТВО,СТАНКОСТРОЕНИЕ,ПЕРЕМЕЖКА,КАЗУС,ВЕЛИК,СФ
ЕРИЧНОСТЬ,ФАЗА,ВСЕЛЕННАЯ,ПИНЦЕТ,МРАЗЬ,МЕДАЛЬОН,СОВОК,БОРОДАЧ,ГОЛУБИНЫЕ,РОСКОШ
НОСТЬ,ПРИНУДИТЕЛЬНОСТЬ,АРТЕЛЬЩИК,ГРИМАСА,МЕРЗОСТЬ,ЧЕРТЫХАНЬЕ,КУШ,ПОЛУПУСТЫНЯ,
ПЛЮГАВОСТЬ,ОБНОВКА,СЛАБОСИЛЬНОСТЬ,СФОРМИРОВАНИЕ,БОДЛИВОСТЬ,ЗАГАЗОВАННОСТЬ,ЭПИ
ЛЕПТИЧКА,ИКОНОГРАФИЯ,ЭСТЕТКА,КРИСТАЛЛИЗАЦИЯ,АФЕРИСТКА,НАУЧНОСТЬ,ПРИЗНАТЕЛЬНОС
ТЬ,ФУНТИК,СТЫДОБА,УСТРЕМЛЕНИЕ,ЗАКАЗЧИК,ЛГУНЬЯ,КОДИФИКАЦИЯ,ОБЕЗЬЯНСТВО,КОНТЕЙН
ЕР,СПИДВЕЙ,КАЛОРИЯ,ПРОСЛОЙКА,БОДРЯК,ОДЫШКА,ПРИСЯДКА,УПОИТЕЛЬНОСТЬ,КУЛИНАРИЯ,З
АГРЕБНОЙ,ОМНИБУС,УСЛОВИЕ,КОПКА,НЕПОЛАДКА,ЗВЕРОВОДСТВО,БУКСИР,ОДУТЛОВАТОСТЬ,ИЗ
ВОЛЕНИЕ,СКЕТЧ,РАЗВОДЫ,ВЫТРАВЛИВАНИЕ,КОММУНАЛКА,НАГРУЗКА,БИБЛИОФИЛКА,СИГАРА,ПО
ЧЕРКОВЕДЕНИЕ,ОПОРОК,СПАД,СТЕНА,ВЫСКОЧКА,ХРЕСТОМАТИЙНОСТЬ,БЕЗБРЕЖНОСТЬ,ПРИТВОР
А,ТЫКВЕННЫЕ,КАТОРЖНИЦА,СКАКАНЬЕ,ЖАРГОН,ВСЕСИЛЬНОСТЬ,СКЕРЦО,ПИКАП,ПИТАНИЕ,...]
[10,15,26,47,78,87,89,112,113,117,126,127,150,153,161,173,173,182,189,192,194
,196,202,231,236,238,262,271,299,310,318,318,354,355,355,362,366,381,382,432,
439,443,448,463,464,488,492,493,498,511,526,537,555,557,563,565,570,589,593,6
16,619,625,631,641,652,657,670,676,683,703,705,706,706,725,729,729,731,733,73
7,787,789,805,806,816,818,821,837,839,851,867,872,877,877,878,886,915,929,942
,962,989,...]
[АРТЕЛЬЩИК,АФЕРИСТКА,БЕЗБРЕЖНОСТЬ,БИБЛИОФИЛКА,БОДЛИВОСТЬ,БОДРЯК,БОЛЬШИНСТВО,Б
ОРОДАЧ,БУКСИР,ВЕЛИК,ВСЕЛЕННАЯ,ВСЕСИЛЬНОСТЬ,ВЫСКОЧКА,ВЫТРАВЛИВАНИЕ,ВЬЮК,ГОЛУБИ
НЫЕ,ГРИМАСА,ДОСЯГАЕМОСТЬ,ЖАРГОН,ЖЕРТВЕННОСТЬ,ЗАГАЗОВАННОСТЬ,ЗАГРЕБНОЙ,ЗАКАЗЧИ
К,ЗАПЕВАЛА,ЗВЕРОВОДСТВО,ИЗВОЛЕНИЕ,ИКОНОГРАФИЯ,КАЗУС,КАЛОРИЯ,КАТОРЖНИЦА,КОДИФИ
КАЦИЯ,КОЛДУНЬЯ,КОММУНАЛКА,КОНТЕЙНЕР,КОПКА,КРИСТАЛЛИЗАЦИЯ,КУЛИНАРИЯ,КУШ,ЛГУНЬЯ
,ЛИЦЕПРИЯТИЕ,ЛОЩИНА,ЛУЧЕНИЕ,МЕДАЛЬОН,МЕРЗОСТЬ,МРАЗЬ,НАГРУЗКА,НАУЧНОСТЬ,НЕПЕРЕ
ДАВАЕМОСТЬ,НЕПОЛАДКА,ОБЕЗЬЯНСТВО,ОБНОВКА,ОДУТЛОВАТОСТЬ,ОДЫШКА,ОМНИБУС,ОПОРОК,
ПАНСИОНАТ,ПЕРЕМЕЖКА,ПИКАП,ПИНЦЕТ,ПИТАНИЕ,ПЛЮГАВОСТЬ,ПОДПИСЬ,ПОЛУПУСТЫНЯ,ПОЧЕР
КОВЕДЕНИЕ,ПРИЗНАНИЕ,ПРИЗНАТЕЛЬНОСТЬ,ПРИНУДИТЕЛЬНОСТЬ,ПРИСЯДКА,ПРИТВОРА,ПРОСЛО
ЙКА,ПРОТРАВИТЕЛЬ,ПТИЧКА,РАЗВОДЫ,РОСКОШНОСТЬ,СИГАРА,СКАКАНЬЕ,СКЕРЦО,СКЕТЧ,СЛАБ
ОСИЛЬНОСТЬ,СОВОК,СПАД,СПИДВЕЙ,СТАНКОСТРОЕНИЕ,СТЕНА,СТЫДОБА,СФЕРИЧНОСТЬ,СФОРМИ
РОВАНИЕ,ТЕПЛОКРОВНОЕ,ТЫКВЕННЫЕ,УЗЛОВАТОСТЬ,УПОИТЕЛЬНОСТЬ,УСЛОВИЕ,УСТРЕМЛЕНИЕ,
УСТУПЧАТОСТЬ,ФАЗА,ФУНТИК,ХРЕСТОМАТИЙНОСТЬ,ЧЕРТЫХАНЬЕ,ЭПИЛЕПТИЧКА,ЭСТЕТКА,...]

Таким образом, благодаря обобщённым процедурным типам *Func* нам не пришлось объявлять новые процедурные типы, а анонимные функции избавили нас от написания полноценных именованных функций.

Но не возникло ли у вас ощущения, что и процедурные переменные здесь излишни, ведь они нужны только для того, чтобы передать анонимную функцию в функцию сортировки? – Действительно, мы можем отправить в функцию сортировки непосредственно анонимную функцию:

```
Console.ForegroundColor := ConsoleColor.Magenta;
Println;
lstInt := BubbleSort&<integer>(lstNum, function(n1, n2) ->
                                       begin
                                          Result := n1 > n2;
                                       end );
Println(lstInt);

Println;
lstStr2 := BubbleSort&<string>(lstStr, function(s1, s2) ->
                                       begin
                                          Result :=
                          (IComparable(s1)).CompareTo(s2) > 0;
                                       end );
Println(lstStr2);
```



Заменив прежний код исправленным, мы получим правильные результаты, избежав создания не только именованных функций, но и процедурных переменных для хранения их адресов.

Как вы видите, анонимные функции облегчают написание исходного кода программы и делают разработку программ более удобной. Однако для компилятора это дополнительная работа, поскольку он создаёт из наших анонимных функций именованные функции, которые используются в программе так, как и раньше. Такие послабления программистам называют *синтаксическим сахаром*, с которым мы встретимся ещё при изучении лямбда-выражений.

И тут самое время вспомнить, что обобщённый класс *List<T>* имеет метод сортировки Sort:

```
procedure Sort ( comparison: Comparison<T>);
```

В отличие от нашей пузырьковой функции он сортирует **исходный** список, а не возвращает новый, отсортированный, но зато работает так быстро, что мы можем сортировать практически любые списки!

В качестве арнумента ему следует передать процедурную переменную **comparison** типа *Comparison<T>:*

```
function Comparison<T>( t1, t2 : T): integer;
```

Обратите внимание, что она возвращает значение типа *integer*, а не *boolean*, как наши методы сравнения!

Если **t1 < t2**, то возвращаемое значение меньше нуля
Если **t1 = t2**, то возвращаемое значение равно нулю
Если **t1 > t2**, то возвращаемое значение больше нуля.

Мы опять можем создать пару процедурных переменных типа **Comparison<T>** из анонимных функций:

```
var compi: Comparison<integer> :=   function(n1, n2) ->
                                    begin
```

```
                                        Result := n1 - n2;
                                    end;


var comps: Comparison<string> :=    function(s1, s2) ->
                                    begin
                                        Result :=
                            (IComparable(s1)).CompareTo(s2);
                                    end;
```

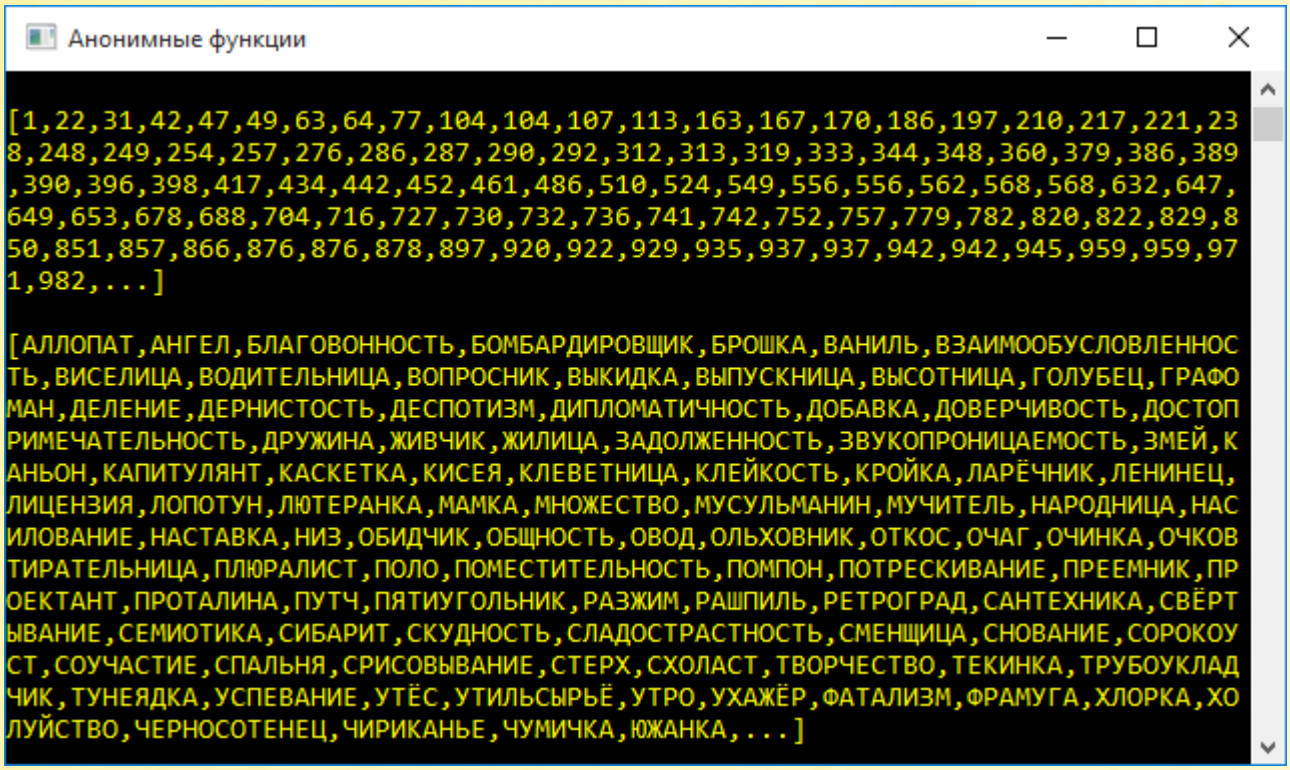
И передать их методу сортировки:

```
Println;
lstNum.Sort(compi);
Println(lstNum);

Println;
lstStr.Sort(comps);
Println(lstStr);
```

Либо отказаться от написания лишнего кода и передать в метод сортировки непосредственно анонимные функции:

```
Console.ForegroundColor := ConsoleColor.Yellow;
Println;
lstNum.Sort(function(n1, n2) ->
            begin
               Result := n1 - n2;
            end
           );
Println(lstNum);

Println;
lstStr.Sort(function(s1, s2) ->
            begin
               Result := (IComparable(s1)).CompareTo(s2);
            end
           );
Println(lstStr);
```

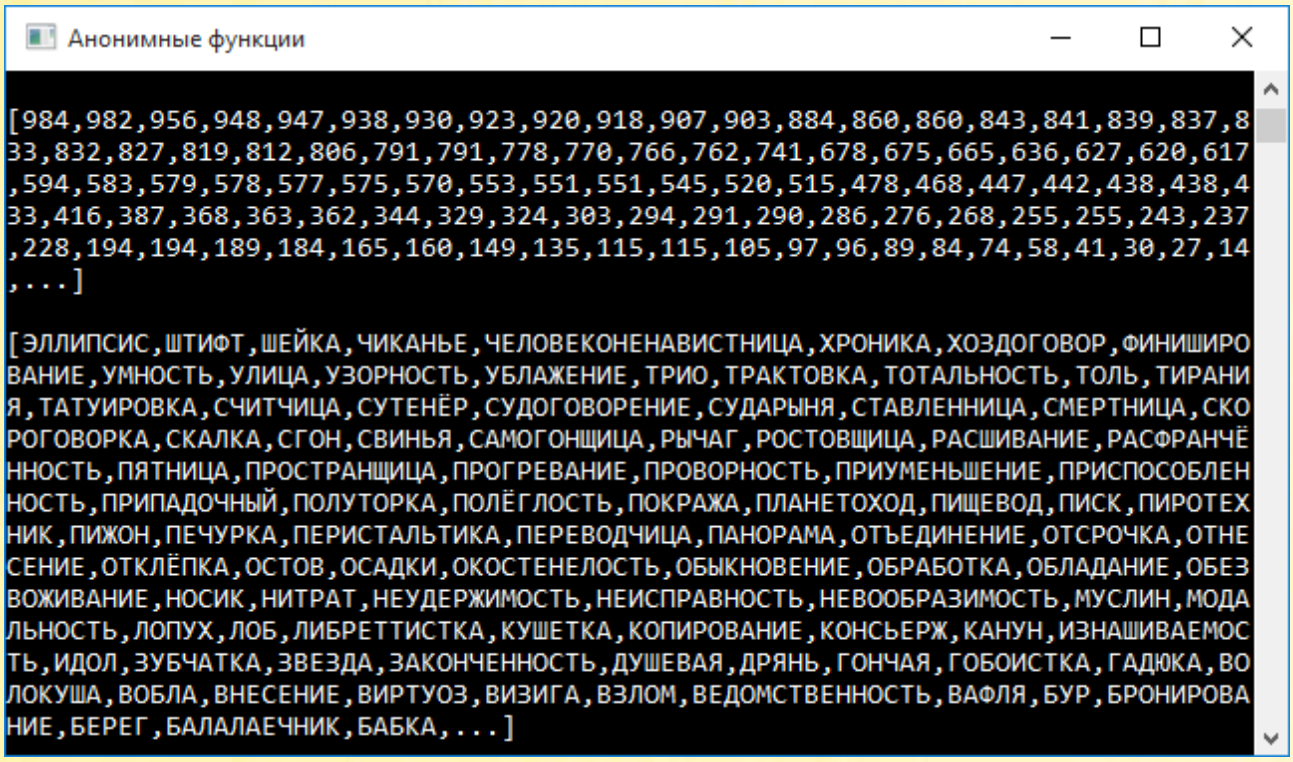
Оба способа хороши, но второй значительно короче. А результат сортировки мы получим одинаковый:



Столь же просто мы отсортируем элементы списков в обратном порядке. Достаточно поставить знак минус:

```
//обратная сортировка:
Println;
Console.ForegroundColor := ConsoleColor.White;
lstNum.Sort(function(n1, n2) ->
            begin
               Result := -(n1 - n2);
            end
           );
Println(lstNum);

Println;
lstStr.Sort(function(s1, s2) ->
            begin
               Result := -(IComparable(s1)).CompareTo(s2);
            end
           );
```

```
Println(lstStr);
```

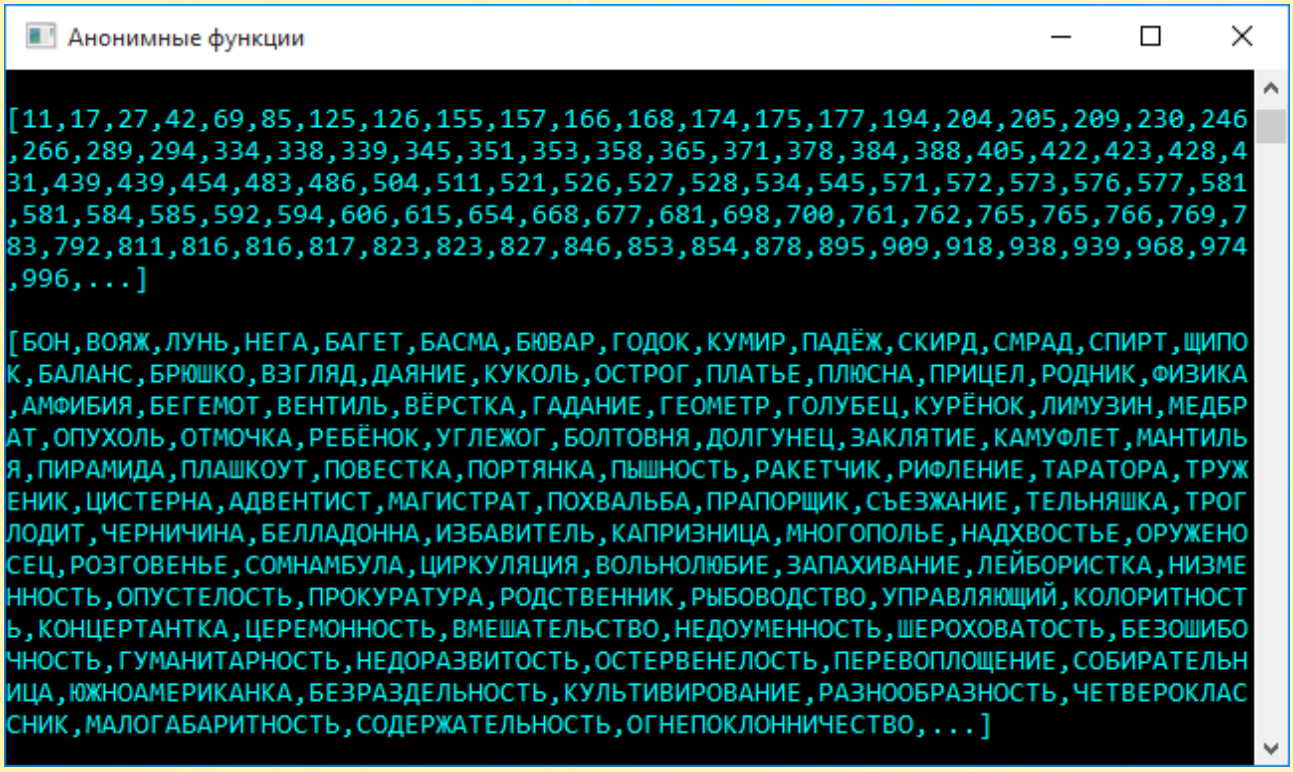


И отсортируем элементы списков по длине:

```
//сортировка по длине:
Println;
Console.ForegroundColor := ConsoleColor.Cyan;
lstNum.Sort(function(n1, n2) ->
            begin
               var len1 := n1.ToString().Length;
               var len2 := n2.ToString().Length;
               var res: integer;
               if (len1 = len2) then
                  res := n1 - n2
               else
                  res := len1 - len2;
               Result := res;
            end
           );
Println(lstNum);

Println;
```

```
lstStr.Sort(function(s1, s2) ->
            begin
               var len1 := s1.Length;
               var len2 := s2.Length;
               var res: integer;
               if (len1 = len2) then
                res := (IComparable(s1)).CompareTo(s2)
            else
                res := len1 - len2;
            Result := res;
            end
           );
Println(lstStr);
```

На этот раз анонимные функции получились **многострочными**, но никаких сложностей в их написании мы не встретили:



> Считается, что анонимные функции должны состоять не более чем из 3-5 строк. Иначе лучше определить **именованную функцию**.

При сортировке по длине обобщённая именованная функция *CompareLen<T>* была бы более уместна, чем аналогичная анонимная функция.

Как вы видите, анонимные функции очень похожи на именованные, но описываются внутри функции, не имеют идентификатора и для них не нужно явно указывать тип возвращаемого значения.

Как обычно, параметры анонимной функции и её локальные переменные видны только внутри тела функции. Однако анонимная функция может использовать *локальные* переменные (а также *значимые* параметры) функции, в которой она находится (но не локальные переменные, объявленные в блоке кода внутри функции – например, в теле операторов *if* или *for* – если сама анонимная функция не находится в теле этих операторов). Такие переменные называются **внешними** (*outer variables*). Говорят, что анонимная функция **захватывает** внешнюю переменную, и тогда та теряет статус локальной переменной и продолжает существовать до тех пор, пока анонимная функция не будет уничтожена сборщиком мусора. Такой способ использования внешних переменных в анонимной функции называют **замыканием** (*closure*). В этом случае компилятор создаёт новый класс для анонимной функции, который имеет открытое поле с захваченной внешней переменной. В ряде случаев это может привести к изменению поведения таких переменных, так что пользуйтесь замыканиями с осторожностью.

Множественные процедурные переменные могут ссылать на несколько именованных и анонимных функций. Функции **добавляются** в список вызовов с помощью операторов + и +=. Функции можно **удалять** из списка вызовов так, как это было описано раньше.

Лямбда-выражения

Лямбда-выражения – это следующий шаг по упрощению синтаксиса и очередной синтаксический сахар, так как они всё равно преобразуются компилятором в анонимные функции. В современном программировании лямбда-выражения практически полностью вытеснили анонимные функции.

Поскольку лямбда-выражения это всего лишь другая форма записи анонимных функций, то их используют совершенно аналогично. Вполне разумно перевести наши анонимные функции в лямбда-выражения, прежде чем заняться их детальным изучением.

Начните новое приложение и скопируйте в него код из предыдущего проекта, исключая сортировку списков.

Сравним процесс создания процедурных переменных из анонимных функций:

```
var compi: Func<integer, integer, boolean>  := function(n1, n2) ->
begin
    Result := n1 > n2;
end;

var comps: Func<string, string, boolean>  := function(s1, s2) ->
begin
    Result := (IComparable(s1)).CompareTo(s2) > 0;
end;
```

И из лямбда-выражений:

```
var compi2: Comparison<integer> := (n1, n2) -> n1 - n2;
var comps2: Comparison<string>  := (s1, s2) ->
                                   (IComparable(s1)).CompareTo(s2);
```

Сразу бросается в глаза, что лямбда-выражения короче. Для создания процедурных переменных не нужно указывать ключевое слово *function* и типы параметров. Переменная *Result* также отсутствует, поскольку она предполагается после лямбда-оператора ->. При желании эту переменную можно указать, но тогда лямбда-выражение следует заключить в операторные скобки:

```
var comps3: Comparison<string>  := (s1, s2) -> begin
                                Result:= IComparable(s1).CompareTo(s2) end;
```

Запись

```
(список_параметров) -> выражение
```

называется **лямбда-выражением**.

Запись

```
(список_параметров) -> begin операторы; end;
```

называется **лямбда-оператором**. Лямбда-оператор отличается от лямбда-выражения тем, что имеет операторные скобки, в которые заключены операторы (или один оператор).

> То есть лямбда-выражение – это запись, которая состоит из единственного выражения, не заключена в операторные скобки и не заканчивается точкой с запятой.

Если список параметров состоит из **одного** элемента, то круглые скобки допускается не ставить. Если параметры отсутствуют или их больше одного, то круглые скобки необходимы. Если параметров несколько, то они перечисляются через запятую.

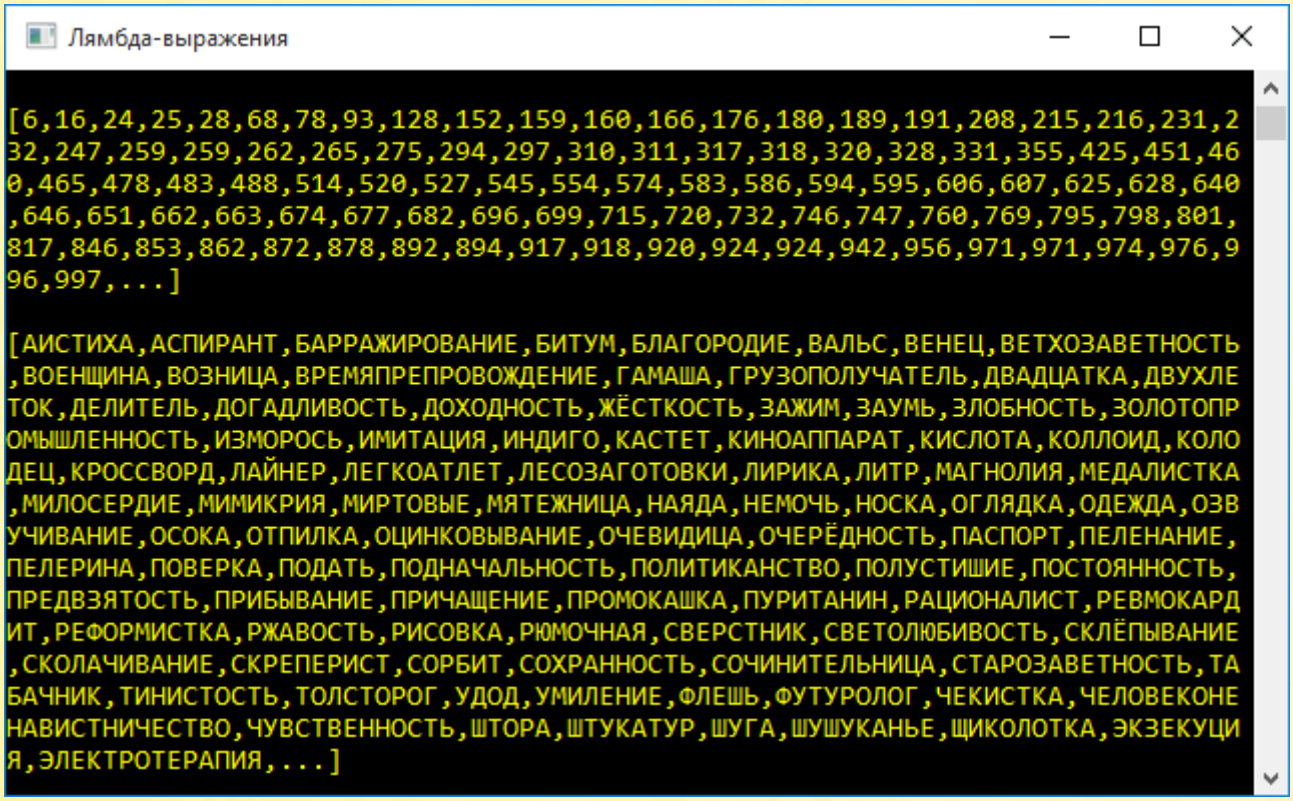
Если компилятор может самостоятельно вывести типы параметров, то их не нужно явно указывать в списке параметров. В противном случае тип параметров указывается, как обычно.

Оператор -> делит лямбда-выражение (или лямбда-оператор) на две части. **Слева** находится список параметров анонимной функции, а **справа** – выражение или операторы, которые могут возвращать какое-либо значение (или ничего не возвращать).

Создав процедурные переменные compi2 и comps2 с помощью лямбда-выражений, мы можем использовать их как параметры в методах сортировки:

```
// естественная сортировка:
Println;
Console.ForegroundColor := ConsoleColor.Yellow;
Println;
lstNum.Sort(compi2);
Println(lstNum);

Println;
lstStr.Sort(comps2);
Println(lstStr);
```

Этот код в точности повторяет код из предыдущего проекта, поскольку процедурным переменным всё равно, каким именно способом они были созданы. И методы сортировки действуют так же, как и раньше:



И опять так же, как и раньше, мы можем передать в метод сортировки непосредственно лямбда-оператор:

```
Console.ForegroundColor := ConsoleColor.Magenta;
Println;
        lstNum.Sort((n1, n2) ->
        begin
           Result:= n1 - n2;
        end);
Println(lstNum);

Println;
lstStr.Sort((s1, s2) ->
        begin
           Result:= (IComparable(s1)).CompareTo(s2);
        end);
Println(lstStr);
```

Результат сортировки ничем не отличается от того, что показан на рисунке выше. Для обратной сортировки необходимо передать в метод сортировки однострочные лямбда-выражения:

```
// обратная сортировка:
Println;
Console.ForegroundColor := ConsoleColor.White;
lstNum.Sort( (n1, n2) -> -(n1 - n2) );
Println(lstNum);

Println;
lstStr.Sort( (s1, s2) -> -(IComparable(s1)).CompareTo(s2) );
Println(lstStr);
```

Сортировка по длине опять выглядит тяжеловато из-за множества операторов в теле лямбда-оператора:

```
// сортировка по длине:
Println;
Console.ForegroundColor := ConsoleColor.Cyan;
lstNum.Sort((n1, n2) ->
```

```
            begin
               var len1 := n1.ToString().Length;
               var len2 := n2.ToString().Length;
               var res: integer;
               if (len1 = len2) then
                  res := n1 - n2
               else
                  res := len1 - len2;
               Result := res;
            end
           );
Println(lstNum);

Println;
lstStr.Sort((s1, s2) ->
            begin
               var len1 := s1.Length;
               var len2 := s2.Length;
               var res: integer;
               if (len1 = len2) then
                res := (IComparable(s1)).CompareTo(s2)
            else
                res := len1 - len2;
            Result := res;
            end
           );
Println(lstStr);
```

Но зато хорошо видно, что тело лямбда-оператора ничем не отличается от тела анонимной функции, так что лямбда-операторы можно легко получить из анонимных функций.

В программировании лямбда-выражения используются значительно чаще, чем лямбда-операторы. Например, мы хотим найти в списках числа или слова-**палиндромы**.

Пишем обобщённую функцию для определения палиндромности заданного объекта n:

```
function IsPalindrom<T>(n : T): boolean;
begin
   var str := n.ToString();
```

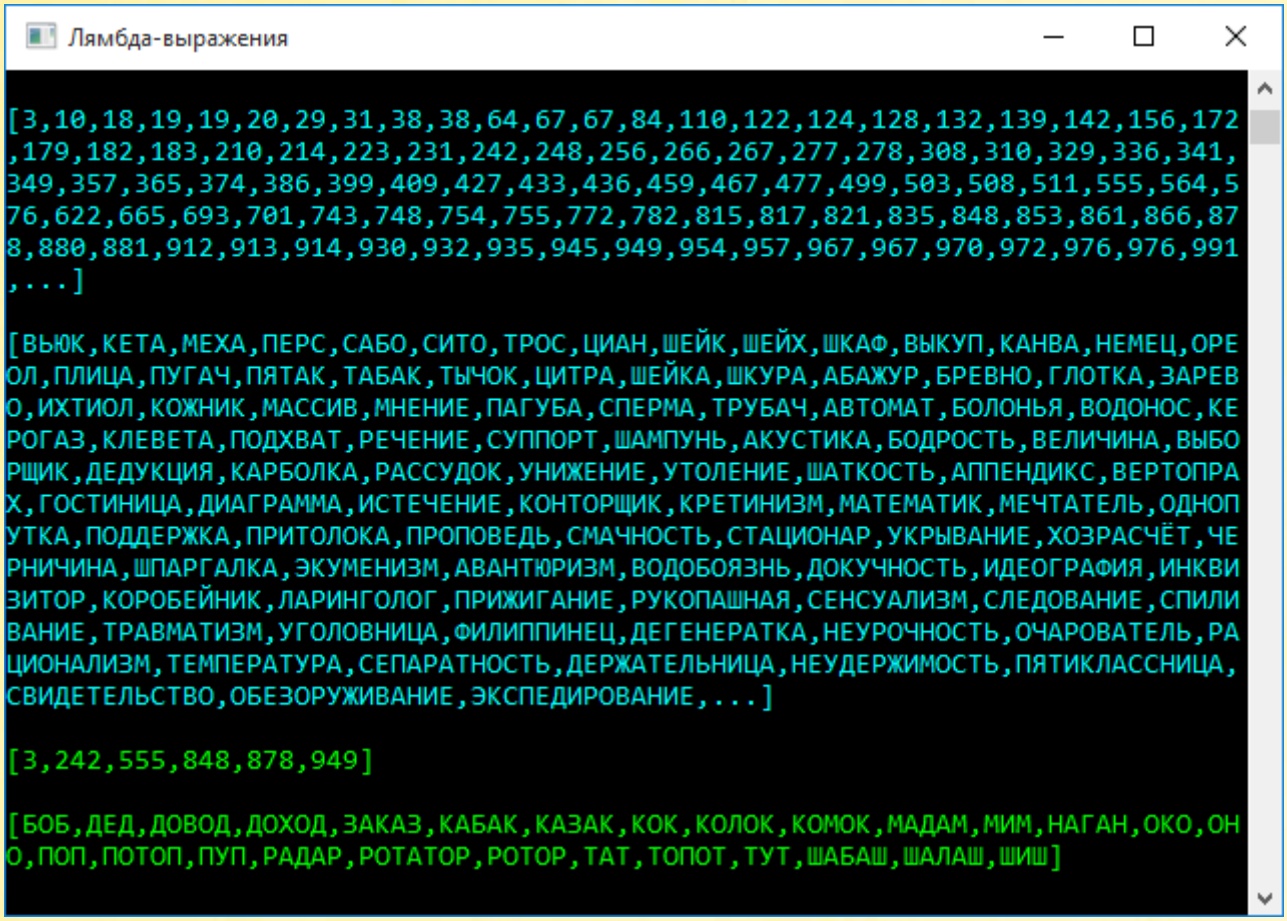
```
    var len := str.Length;
    // слово-палиндром?
    for var i := 1 to len div 2 do
    begin
       var ch1 := str[len - i + 1];
       var ch2 := str[i];
       if (ch1 <> ch2) then
       begin
          // не палиндром:
          Result:= false;
          exit;
       end;
    end;
    // палиндром:
    Result:= true;
end;
```

Теперь с помощью оператора запроса Where, который имеется у списков, мы легко составим новые коллекции – из чисел и слов:

```
// ищем палиндромы:
Console.ForegroundColor := ConsoleColor.Green;
Println;
var pali := lstNum.Where(n -> IsPalindrom&<integer>(n));
Println(pali);

Println;
var pals := lstStrAll.Where(s -> IsPalindrom&<string>(s));
Println(pals);
```

Методу расширения *Where* мы передаём простое лямбда-выражение, включающее вызов функции *IsPalindrom*.

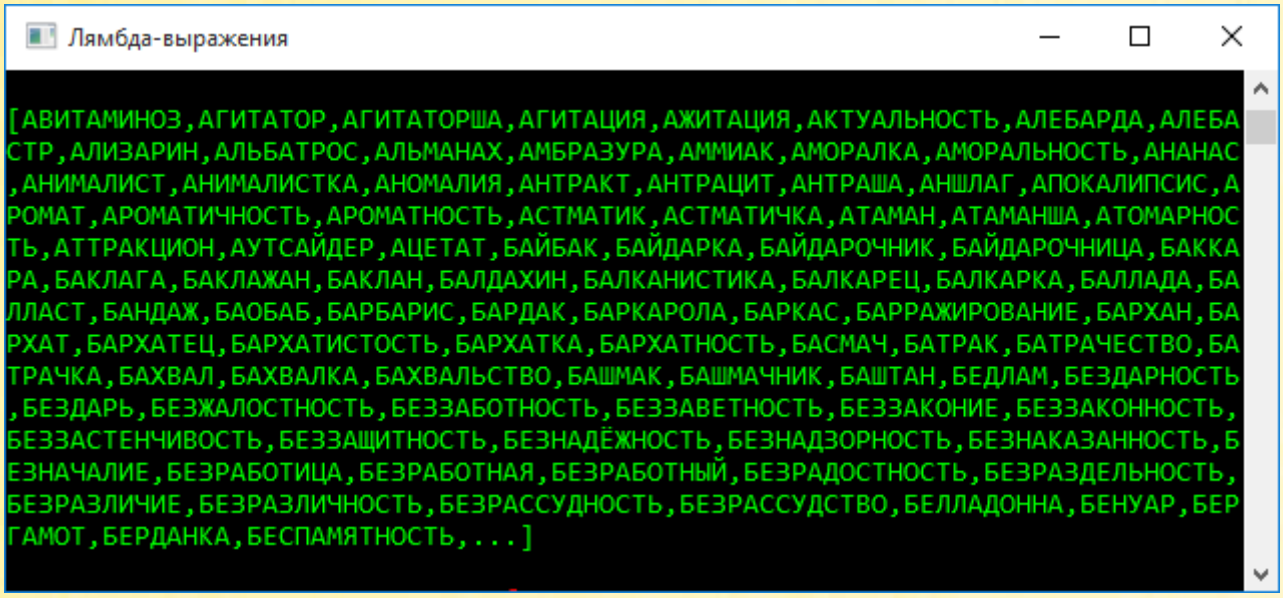
На рисунке зелёным цветом напечатаны искомые палиндромы:



Аналогично мы можем «отфильтровать» из списка строк такие, которые не короче, например, пяти букв, и пятая буква – А:

```
Console.WriteLine();
int npos = 5-1;
IEnumerable<string> pals = lstStrAll.Where(s => s.Length > npos &&
                                                 s[npos] == 'A');
Print<string>(pals.ToList());
```

Рисунок показывает, что метод расширения *Where* правильно нашёл слова:



Здесь мы воспользовались методом расширения *Where* интерфейса *IEnumerable*. Другие методы расширения мы рассмотрим дальше.

Последовательности

Последовательность – это упорядоченная коллекция однотипных элементов.

> По-английски *последовательность* называется *sequence*.

Хорошим примером последовательностей могут служить **арифметические и геометрические прогрессии,** которые изучают в школе.

Элементы последовательностей можно **только просматривать**. Изменить их значения нельзя. Каждый раз нужно создавать новую последовательность.

В *паскале* к последовательностям относятся также **специализированные коллекции**:

- одномерные динамические массивы *array of T*
- списки *List<T>*
- двусвязные списки *LinkedList<T>*
- множества *HashSet<T>* и *SortedSet<T>*

Язык интегрированных запросов *LINQ*

LINQ (*Language Integrated Query*) - *язык интегрированных запросов* - предназначен для обработки коллекций, реализующих интерфейс *IEnumerable<T>.* Это могут быть последовательности, массивы, списки, словари, базы данных и другие источники.

Чтобы получить информацию от источника данных, нужно составить **запрос** (*query*), под которым понимают выражение из **операторов запроса** (их называют также **стандартными операторами запроса** *LINQ*, или *LINQ Standard Query Operators, LSQO*).

Операторы запроса - это методы расширения класса **Enumerable**, который находится в пространстве имён *System.Linq*.

Запросы чаще всего получают и возвращают **последовательности** (*sequences*) типа *IEnumerable<T>,* состоящие из отдельных **элементов** типа *T*. Роль операторов состоит в генерировании выходной последовательности (результата запроса) путём преобразования входной последовательности, которая в запросе **никогда не изменяется**. Как правило, не изменяется и порядок следования элементов в выходной последовательности по сравнению с их исходным порядком: если какой-либо элемент входной последовательности стоит раньше другого, то и в выходной последовательности будет точно такой же порядок их следования.

Некоторые операторы возвращают не последовательности, а **отдельные элементы** (*поэлементные операторы*), **числовые** (операторы *агрегирования*) или **логические** значения (*квантификаторы*). Такие операторы занимают в запросе **последнее** место.

Операторы запросов *LINQ*

Для удобства пользования все многочисленные операторы запросов разбиты на группы:

Метод	Группа	Отложенный
Aggregate	Методы агрегирования	
All	Методы-квантификаторы	
Any	Методы-квантификаторы	
AsEnumerable	Методы преобразования	v
Average	Методы агрегирования	
Cast	Методы преобразования	v
Concat	Методы для множеств	v
Contains	Методы-квантификаторы	
Count	Методы агрегирования	
DefaultIfEmpty	Методы элементов	v
Distinct	Методы для множеств	v
ElementAt	Методы элементов	
ElementAtOrDefault	Методы элементов	
Empty	Методы генерирования	v
Except	Методы для множеств	v
First	Методы элементов	

FirstOrDefaul	Методы элементов	
GroupBy	Методы группирования	v
GroupJoin	Методы объединения	v
Intersect	Методы для множеств	v
Join	Методы объединения	v
Last	Методы элементов	
LastOrDefault	Методы элементов	
LongCount	Методы агрегирования	
Max	Методы агрегирования	
Min	Методы агрегирования	
OfType	Методы преобразования	v
OrderBy	Методы сортировки	v
OrderByDescending	Методы сортировки	v
Range	Методы генерирования	v
Repeat	Методы генерирования	v
Reverse	Методы сортировки	v
Select	Методы проецирования	v
SelectMany	Методы проецирования	v
SequenceEqual	Методы-квантификаторы	
Single	Методы элементов	

SingleOrDefault	Методы элементов	
Skip	Методы фильтрации	v
SkipWhile	Методы фильтрации	v
Sum	Методы агрегирования	
Take	Методы фильтрации	v
TakeWhile	Методы фильтрации	v
ThenBy	Методы сортировки	v
ThenByDescending	Методы сортировки	v
ToArray	Методы преобразования	
ToDictionary	Методы преобразования	
ToList	Методы преобразования	
ToLookup	Методы преобразования	v
Where	Методы фильтрации	v
Zip	Методы объединения	v

> Существует множество других способов группировки этих методов, поэтому больше ориентируйтесь этот алфавитный список, чем на конкретные группы.

Отложенные методы не выполняются до тех пор, пока не вызываются методы, использующие элементы выходной последовательности.

Генерирование последовательностей

Чтобы использовать последовательности при решении задач, их нужно сначала создать или генерировать. Для этого в *паскале* имеется несколько способов.

Функция Range

Основной способ генерирования последовательности натуральных чисел – применение функции Range:

```
function Range(a,b: integer): sequence of integer;
```

Она возвращает последовательность целых чисел типа *integer*.

a – первый элемент последовательности
b – последний элемент

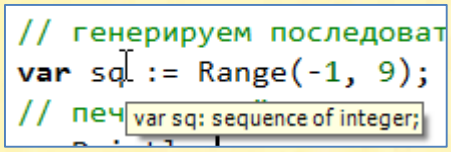
Если a = b, то в последовательности окажется 1 элемент.
Если a < b, то последовательность будет пустой.

Для изучения этой важной функции пишем небольшую программу:

```
uses System;


begin
   //заголовок окна:
   Console.Title := 'Функция Range';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Функция Range');
   Console.ForegroundColor := ConsoleColor.Green;
   Println;

   // генерируем последовательность натуральных чисел:
   var sq := Range(-1, 9);
   // печатаем её:
```

```
    sq.Println;

    Println;
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Функция **Range(-1, 9)** возвращает последовательность целых чисел в заданном диапазоне от минус единицы до девяти. Чтобы узнать тип результата, подведите к имени переменной курсор мышки:



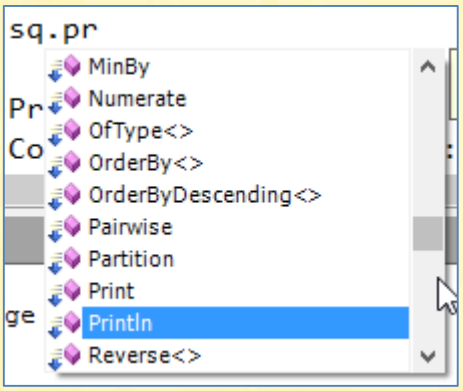
На самом деле функция *Range* ничего не возвращает, пока последовательность не будет использована в программе. Такие функции называют «ленивыми».

Но нам, конечно, хочется посмотреть, что же сгенерировала наша функция.

У последовательностей имеются методы расширения **Print** и **Println**. Второй метод от первого отличается только тем, что переводит печать информации на другую строку.

Так как переменная **sq** имеет тип *sequence of integer*, то мы можем применить к ней один из методов печати (обычно это **Println**). Для этого после имени переменной нужно поставить точку, а затем напечатать имя метода.

В *Редакторе кода* имеется **интеллектуальная подсказка**, которая по мере ввода имени метода показывает возможные продолжения. Переведите клавишами со стрелками выделение на нужную строку и нажмите клавишу ВВОД:



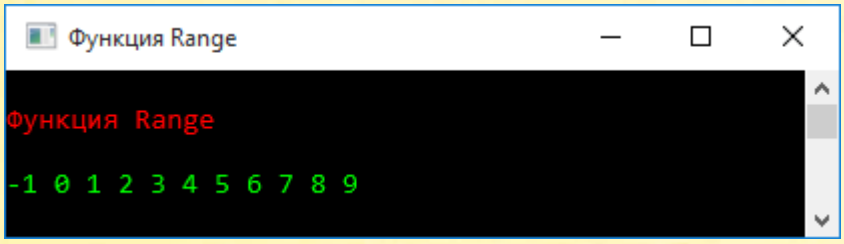
Всё слово целиком будет напечатано после точки.

Вместо клавиш со стрелками вы можете просто щёлкнуть мышкой по нужной строке.

Теперь у нас получилась такая «программа»:

```
// генерируем последовательность натуральных чисел:
var sq := Range(-1, 9);
// печатаем её:
sq.Println;
```

Она **распечатывает** последовательность на экране:



Так как функция *Range* возвращает последовательность, то прямо к ней можно применить метод расширения. Тогда код получится короче:

```
// генерируем последовательность натуральных чисел
// и печатаем её:
var sq := Range(-1, 9).Println;
```

Этот способ печати используется очень часто.

К последовательностям можно применять **цепочкой** и несколько методов. Если строка получается длинная, то её можно переносить по точке. Так:

```
// генерируем последовательность натуральных чисел
// и печатаем её:
var sq := Range(-1, 9).
          Println;
```

Или так:

```
var sq := Range(-1, 9)
         .Println;
```

Но обязательно подписывайте идентификаторы один под другим, чтобы код был ясным и понятным.

Методы расширения *Println* и *Print*

Без **методов печати** не обходится ни одна программа, поэтому разберём их более подробно.

По умолчанию все числа или элементы последовательности других типов разделяются **пробелом**, что вы и видели на экране. В этом случае вызывается такой метод расширения:

```
function Println<T>(Self: sequence of T): sequence of T; extensionmethod;
```

Его объявление выглядит довольно страшно, но использовать его очень просто.

Если вы хотите разделить элементы другим символом или строкой, то используйте другой метод:

```
function Println<T>(Self: sequence of T; delim: string): sequence of T;
extensionmethod;
```

Для разделения элементов последовательности используется строка **delim**.

Этот метод возвращает исходную последовательность.

Перейдём к **примерам**.

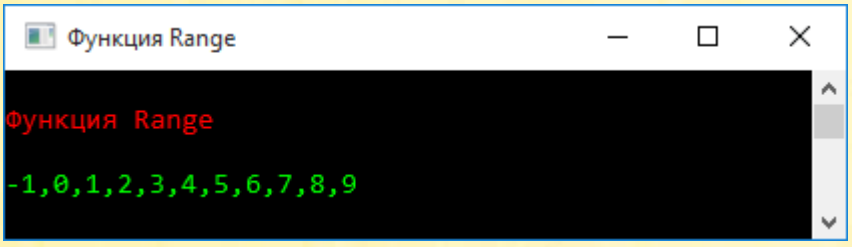
Если в качестве разделителя задать **пустую строку**:

```
var sq := Range(-1, 9)
          .Println('');
```

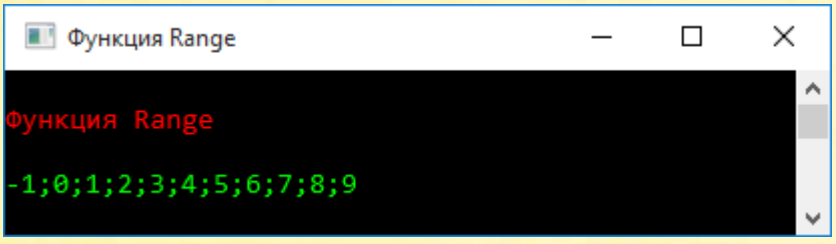
То все элементы будут напечатаны **вплотную** друг за другом:

Часто разделителем служит **запятая**:

```
var sq := Range(-1, 9)
          .Println(',');
```

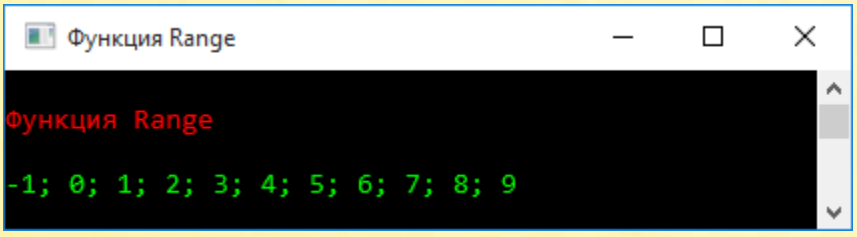


Или **точка с запятой**:



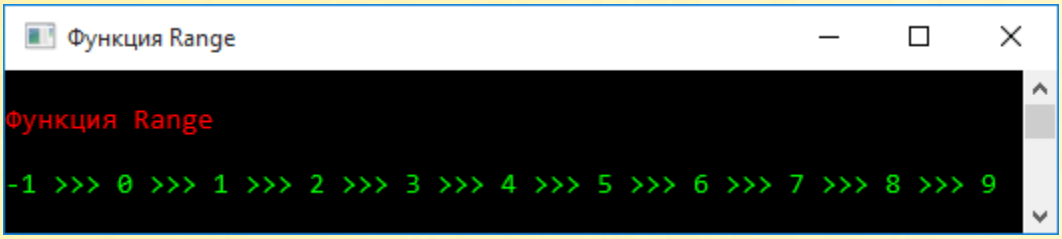
Дополнительно можно поставить пробел, чтобы числа лучше читались на экране:

```
var sq := Range(-1, 9)
          .Println('; ');
```



Разделителем может быть **любая строка**:

```
var sq := Range(-1, 9)
          .Println(' >>> ');
```

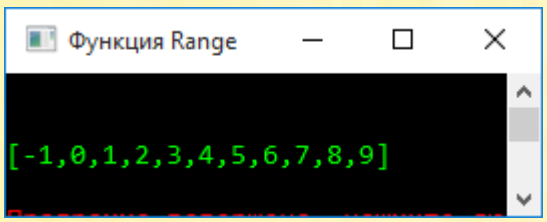


Метод расширения **Print** мы разбирать не будем. Он действует аналогично, только *не переводит* вывод на следующую строку. По этой причине используется значительно реже.

Последовательности можно выводить и обычными процедурами печати **Print** и **Println**, если передать им последовательность в качестве аргумента:

```
Println(sq)
```

Но такой способ печати менее удобен. К тому же элементы последовательности всегда разделяются запятыми:



Точно также печатают последовательности процедуры **Write** и **Writeln**.

Функция Range (продолжение)

Функция **Range** с двумя аргументами возвращает арифметическую последовательность с *разностью 1*. Но ведь могут потребоваться последовательности и с другой разностью! Тогда нужно воспользоваться функцией **Range** с **тремя параметрами**:

```
function Range(a,b,step: integer): sequence of integer;
```

Она возвращает арифметическую последовательность, в которой:

a – первый член
b – максимальное значение элементов последовательности (возможно, недостижимое)
step – разность

Рассмотрим случай, когда b кратно step:

```
var sq3 := Range(0, 15, 3)
           .Println;
```

Первый член арифметической последовательности равен 0, последний – 15.

Добавляем к максимальному значению 1:

```
var sq3 := Range(0, 16, 3)
           .Println;
```

Напечатаны те же самые числа:

Действительно, после числа **15** в последовательности должно следовать число **18**, а оно больше заданного максимального. Будьте внимательны, используя эту функцию!

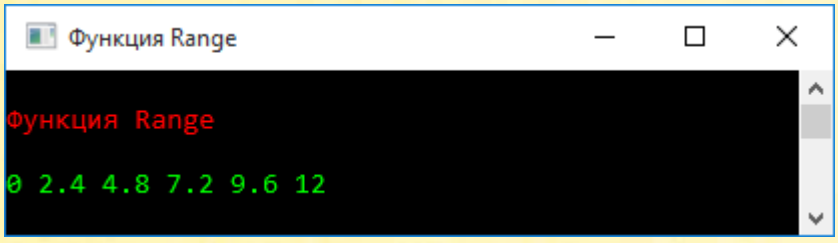
Для **вещественных чисел** имеется подобная функция с тремя аргументами:

```
function Range(a,b: real; n: integer): sequence of real
```

Но она действует иначе!

Функция **Range** с тремя параметрами для вещественных чисел возвращает последовательность действительных чисел в диапазоне **a**, **b**, разделённом на **n** частей:

```
var sqr := Range(0.0, 12.0, 5)
          .Println;
```



Не используйте эту функцию в своих проектах! Чтобы избежать путаницы, она заменена функцией *Partition* (см. дальше).

Функция **Range** может принимать и аргументы типа **char**. Тогда возвращаемая последовательность состоит из символов *Юникода* в диапазоне *c1..c2*:

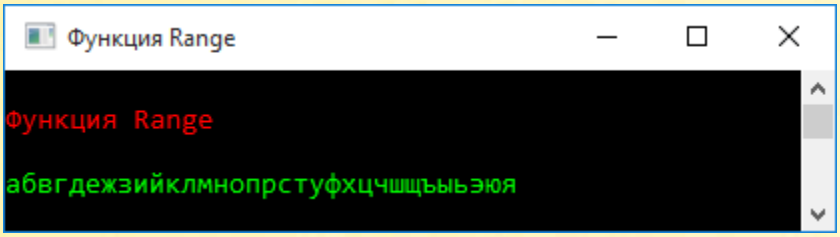
```
function Range(c1,c2: char): sequence of char;
```

Создаём последовательность букв русского алфавита:

```
var sqrus := Range('а','я')
```

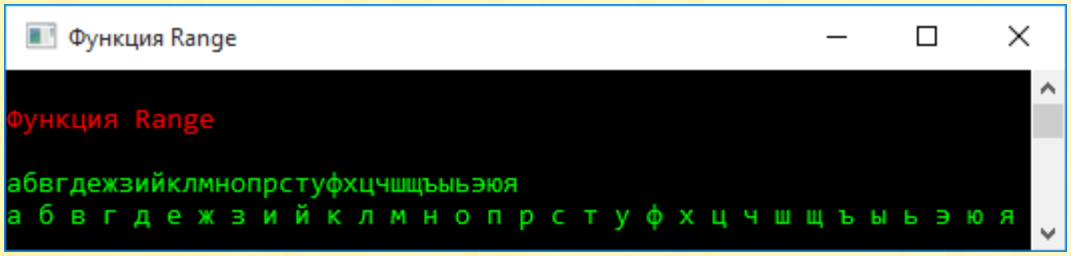
```
                .Println;
```

Обратите внимание, что при печати между символами разделитель не вставляется:

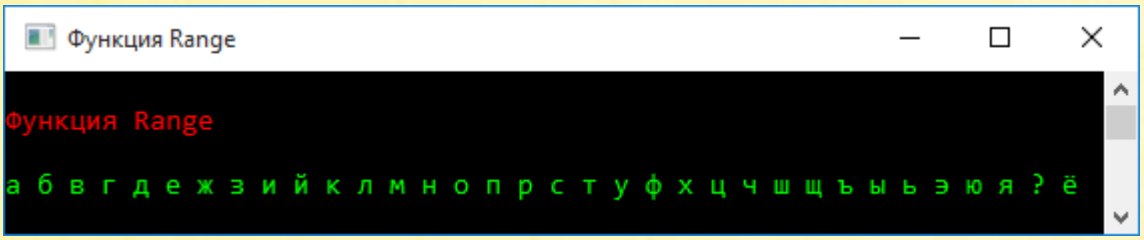


Но вы можете указать нужный разделитель в методе **Println**:

```
var sqrus2 := Range('а','я')
              .Println(' ');
```



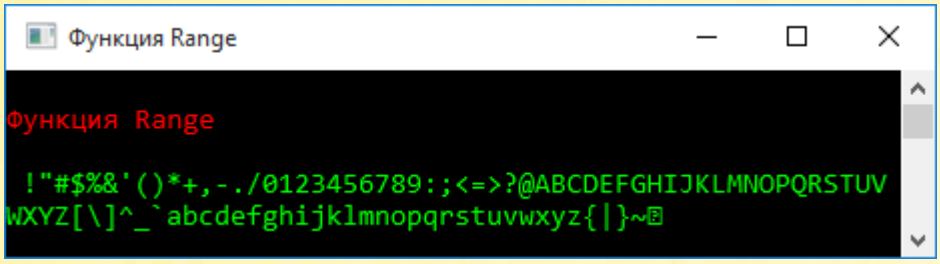
В этой последовательности отсутствует буква **ё**. Если вместо буквы *я* поставить букву *ё*, то она появится, но с ненужной буквой *è*, которая на рисунке заменена вопросительным знаком:



С буквой *ё* проблемы начались давно, и так до сих пор и не решены.

Можно напечатать последовательность всех символов **ASCII**, задавая их коды:

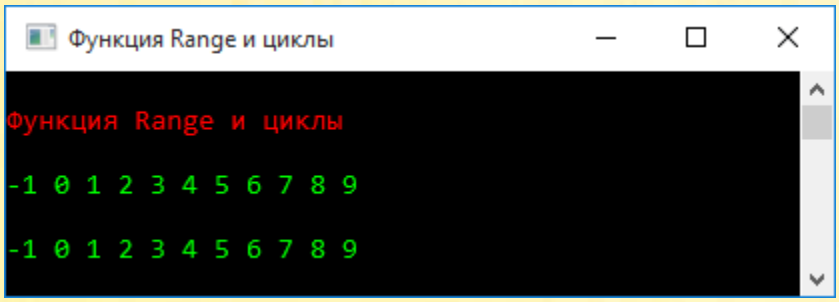
```
var sqascii:= Range(#32,#127)
              .Println;
```



Функция Range и циклы

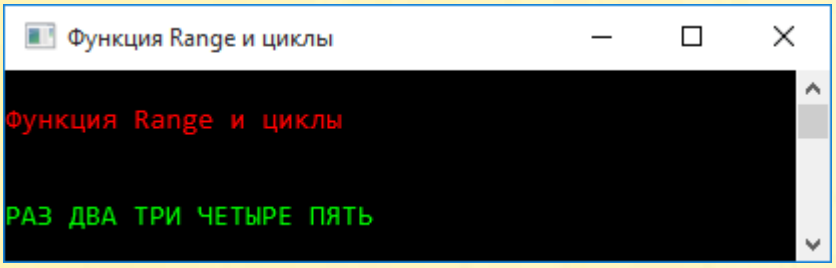
Функцию Range можно использовать совместно с оператором цикла foreach как альтернативу циклу *for*:

```
// генерируем последовательность натуральных чисел
// и печатаем её:
var sq := Range(-1, 9).
          Println;

Println;
foreach var i in sq do
   Print(i);
Println;
```



В этом примере мы просто распечатали значения элементов последовательности, но их можно использовать иначе. Например, функция Range может генерировать индексы в массиве:

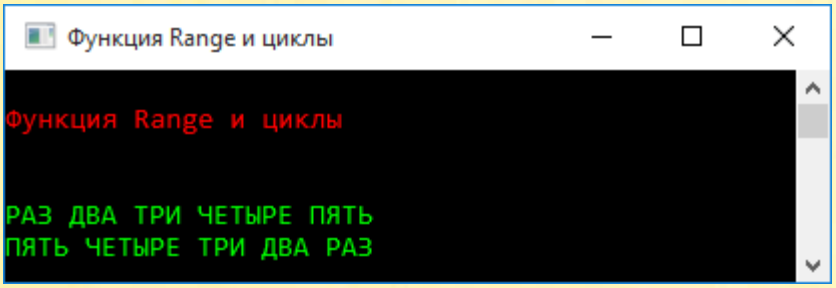
```
var words : array of string := ('РАЗ', 'ДВА', 'ТРИ', 'ЧЕТЫРЕ', 'ПЯТЬ');

foreach var i in Range(0, words.Length-1) do
begin
   Print(words[i]);
end;
```



В цикле foreach переменная цикла имеет тот же тип, что и элементы последовательности. Важно помнить, что в этом цикле нельзя изменить значения элементов последовательности. Можно только перебрать их.

Если использовать функцию Range с тремя параметрами, то можно просмотреть массив в обратном порядке:

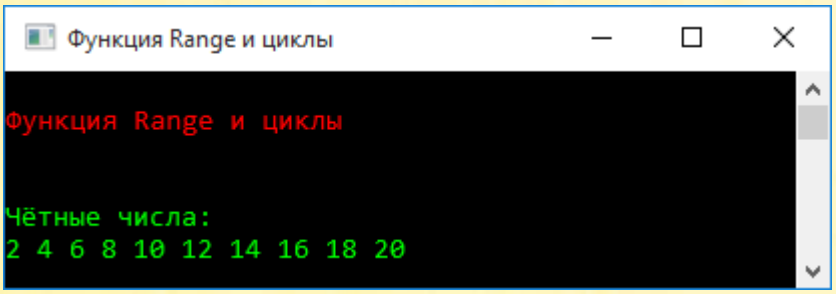
```
foreach var i in Range(words.Length-1, 0, -1) do
begin
  Print(words[i]);
end;
```



Задавая параметру **step** разные значения, вы можете получить более гибкие циклы, чем цикл *for*!

Напечатаем **чётные числа** в заданном диапазоне:

```
Println('Чётные числа:');
foreach var i in Range(2, 20, 2) do
begin
  Print(i);
end;
```



С обычным циклом *for* так просто не получится!

Методы расширения типа integer

Тип *integer* имеет несколько методов расширения для генерирования последовательностей. Иногда ими пользоваться удобнее, чем функцией *Range*.

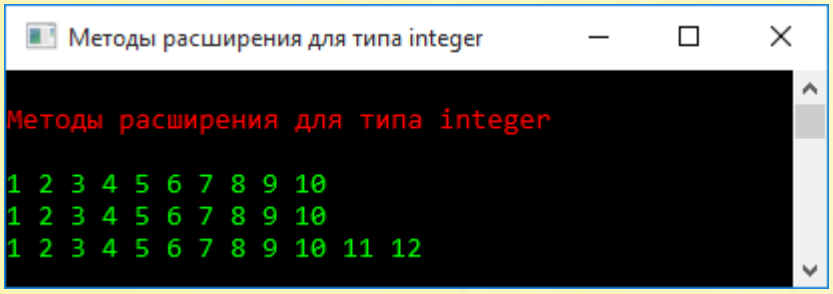
Первый метод расширения также называется **Range**, но он не имеет параметров:

```
function Range(Self: integer): sequence of integer; extensionmethod;
```

Он действует так же, как функция *Range(1,n)*.

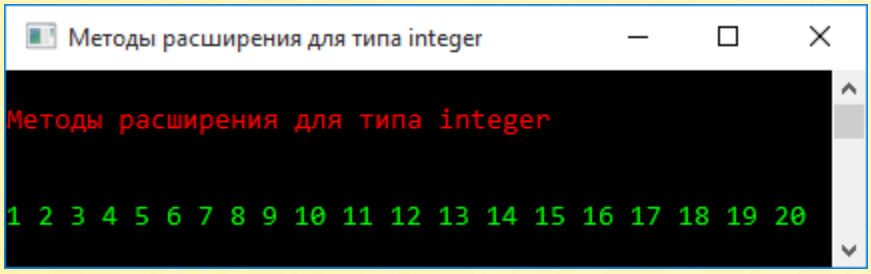
Последовательность всегда начинается с **1** и заканчивается числом **n**. Этот метод можно применять к переменным, литералам и выражениям, имеющим тип *integer*:

```
// переменная типа integer:
var n := 10;
n.Range().Println;

// литерал типа integer:
10.Range().Println;

// выражение типа integer:
(n + 2).Range().Println;
```



Вот так просто можно выполнить цикл 20 раз:

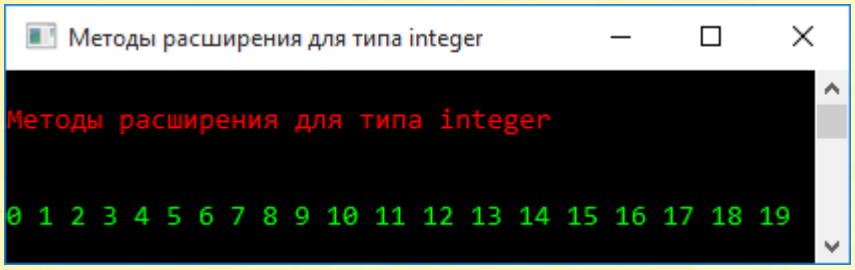
```
// использование в цикле:
foreach var i in 20.Range do
   Print(i);
```



Так элементы последовательностей нумеруются с нуля, то нужна последовательность, которая выдаёт нужное число элементов, также начиная с нуля. Метод расширения Times умеет это делать:

```
// использование в цикле:
foreach var i in 20.Times do
```

```
  Print(i);
```



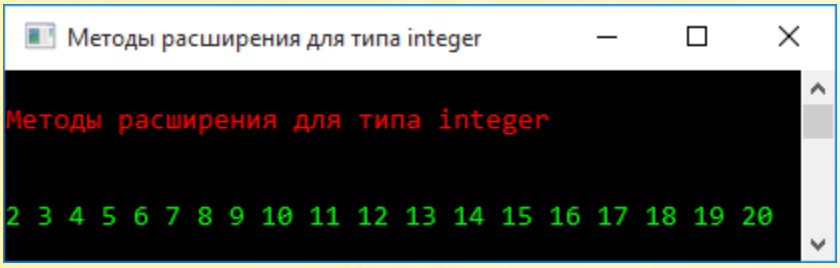
Метод расширения To аналогичен функции *Range(a, b)*:

```
function &To(Self: integer; n: integer): sequence of integer;
                                          extensionmethod;
```

Он генерирует последовательность от заданного значения до значения в скобках.

В этом примере метод To выдаёт последовательность от 2 до 20, что очень удобно использовать в цикле:

```
// использование в цикле:
foreach var i in 2.To(20) do
   Print(i);
```



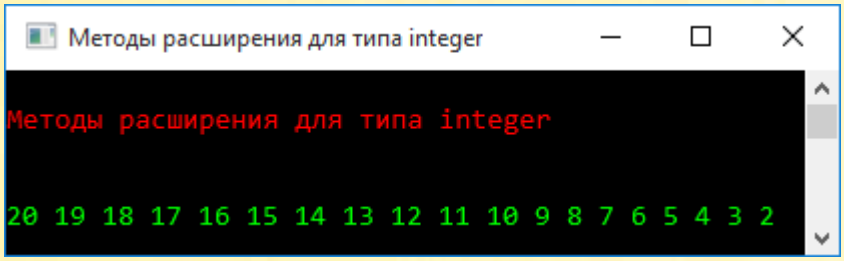
Метод расширения Downto аналогичен функции *Range(a, b, -1)*:

```
function &Downto(Self: integer; n: integer): sequence of integer;
                                              extensionmethod;
```

То есть он выдаёт **убывающую** последовательность.

Посмотрим, как его можно использовать **в цикле**:

```
// использование в цикле:
foreach var i in 20.Downto(2) do
   Print(i);
```



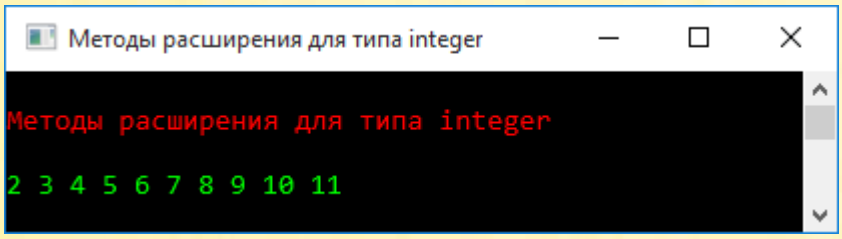
Отличный получился цикл!

Метод расширения Step генерирует **бесконечную** последовательность целых чисел, начиная с заданного значения:

```
function Step(Self: integer): sequence of integer; extensionmethod;
```

С бесконечными последовательностями и циклами нужно быть очень осторожными и всегда предусматривать условия их завершения. Мы применим к последовательности метод **Take**, который рассмотрим дальше. Он пропускает только заданное число элементов и завершает дальнейшее их «производство»:

```
// метод расширения Step:
2.Step()
 .Take(10)
 .Println;
```



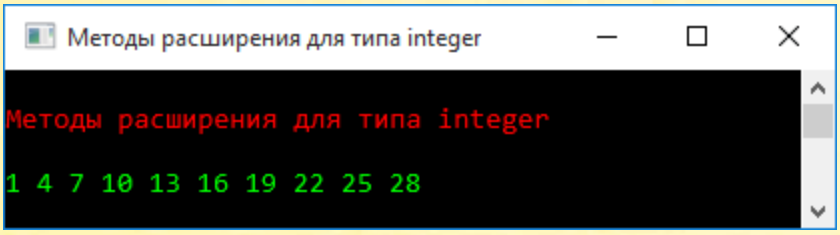
На рисунке видно, что последовательность начинается с двойки, а каждый следующий элемент больше предыдущего на 1.

Мы ограничились первым десятком элементов, иначе последовательность продолжалась бы и дальше.

Метод **Step** может заменить бесконечный цикл, когда число итераций заранее не известно.

Метод Step с параметром генерирует бесконечную последовательность целых чисел, начиная с заданного, с шагом *step*:

```
1.Step(3)
 .Take(10)
 .Println;
```

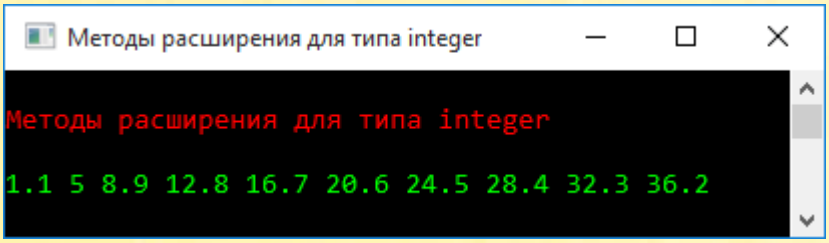


Такой же метод расширения имеется и у типа **real/double**:

```
function Step(Self: real; step: real): sequence of real;
                                       extensionmethod;
```

Он возвращает бесконечную последовательность **вещественных** чисел:

```
1.1.Step(3.9)
    .Take(10)
    .Println;
```



Функции Seq…

Функции SeqRandom и SeqRandomInteger абсолютно одинаковы. Они возвращают последовательность из n случайных целых чисел в диапазоне a..b:

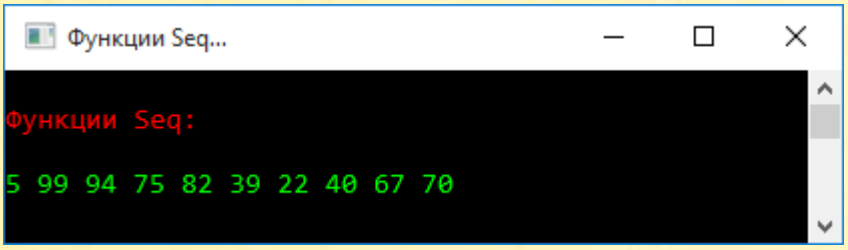
```
function SeqRandom(n: integer := 10;
                   a: integer := 0;
                   b: integer := 100): sequence of integer;
```

```
function SeqRandomInteger(n: integer := 10;
                          a: integer := 0;
                          b: integer := 100): sequence of integer;
```

По умолчанию n равно 10, a равно 0, а b равно 100.

При вызове функции без параметров она выдаст 10 случайных целых чисел в диапазоне 0..99:

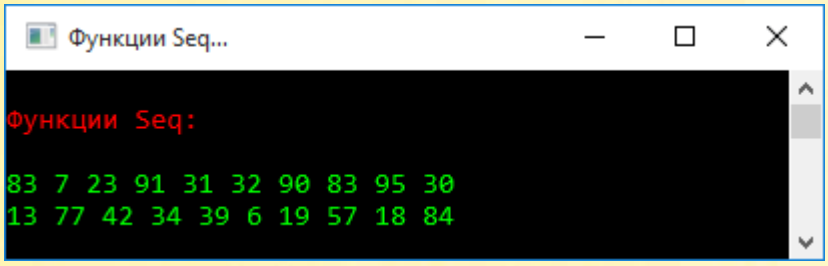
```
var sq := SeqRandom
          .Println;
```



А теперь вопрос: что напечатает метод *Println* во второй раз?

```
var sq := SeqRandom
          .Println;

sq.Println;
```

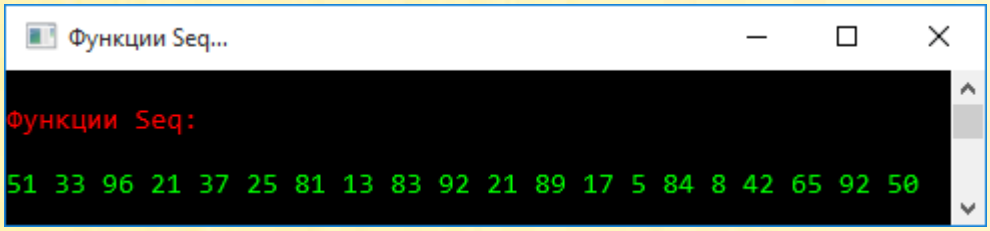
Точно такую же последовательность, что и первый метод *Println*? – А вот и нет! Второй метод *Println* напечатает **новую** случайную последовательность:



Это значит, что функции *SeqRandom* и *SeqRandomInteger* ленивые и при каждом обращении к ним генерируют новую последовательность. Это нужно иметь в виду!

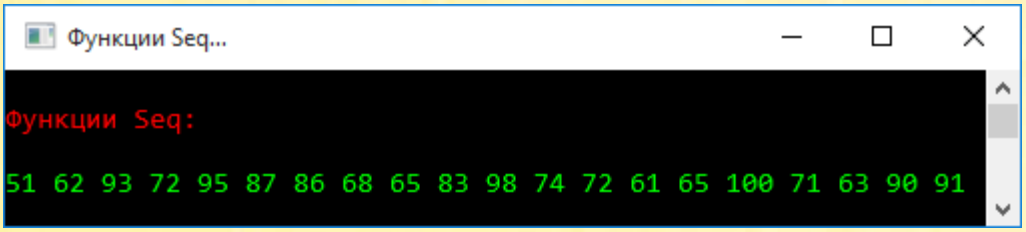
Если вы хотите получить не 10, а, например, 20 чисел в диапазоне по умолчанию, то укажите новое число в скобках:

```
var sq := SeqRandom(20)
          .Println;
```



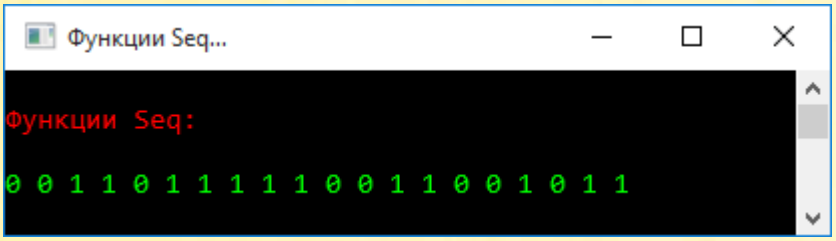
При вызове функции **SeqRandom** с двумя аргументами второй аргумент означает нижнюю границу диапазона. А верхней останется по умолчанию:

```
var sq := SeqRandom(20, 50)
          .Println;
```



И наконец, вызов функции с **тремя** параметрами позволяет задавать число элементов, верхнюю и нижнюю границы диапазона. Так мы можем моделировать подбрасывание монетки:

```
var sq := SeqRandom(20, 0, 1)
          .Println;
```

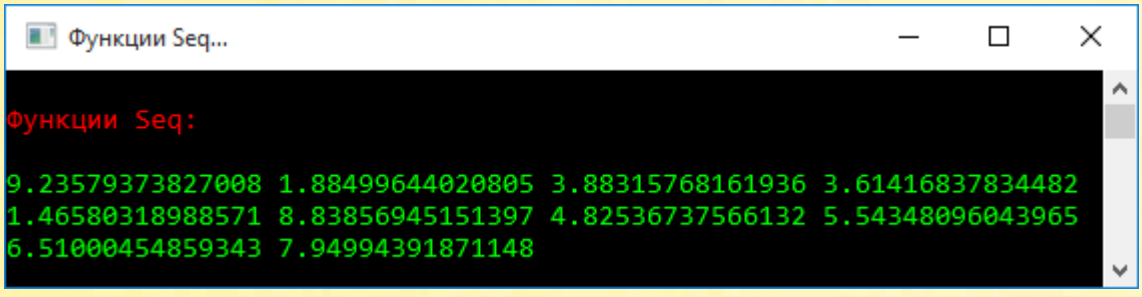


Аналогичная функция **SeqRandomReal** имеется и для *вещественных* чисел:

```
function SeqRandomReal(n: integer;
                       a: real; b: real): sequence of real;
```

У неё нет параметров по умолчанию, поэтому всегда нужно задавать 3 аргумента:

```
var sq := SeqRandomReal(10, 0, 10)
          .Println;
```



Функция SeqGen возвращает последовательность элементов типа *T*:

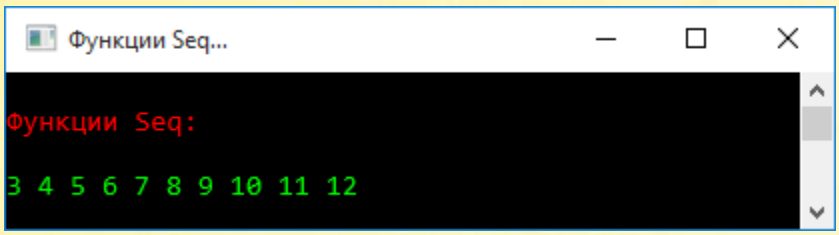
```
function SeqGen<T>(count: integer; f: integer -> T): sequence of T;
```

count – число элементов
f – функция для вычисления элементов последовательности

Она объединяет функцию *Range* и метод расширения *Select*, о котором речь пойдёт дальше.

С помощью этой функции легко создать любую **арифметическую последовательность**:

```
var sq := SeqGen(10, i -> i + 3)
          .Println;
```



Здесь:

10 – число элементов в последовательности
i -> i + 3 – функция, вычисляющая значения элементов. Начальное значение переменной *i* равно 0, поэтому первый элемент *i + 3* равен 3. Затем *i* получает значение 1, а элемент *i + 3* = 1 + 3 = 4. И так далее.

Если начальное значение переменной *i* (название может быть любым) должно отличаться от нуля, то используйте функцию **SeqGen с тремя параметрами**:

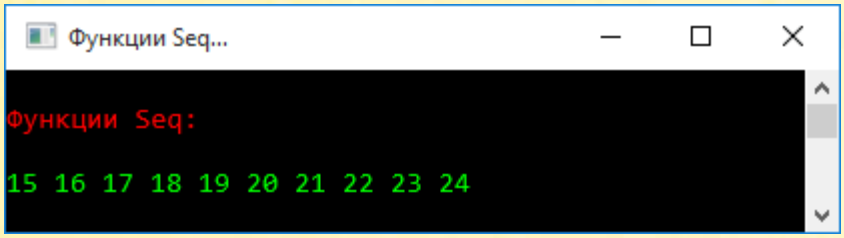
```
function SeqGen<T>(count: integer;
                   f: integer -> T; from: integer): sequence of T;
```

count – число элементов
f – функция для вычисления элементов последовательности
from – это начальное значение переменной

Пусть начальное значение переменной равно 10:

```
var sq := SeqGen(10, i -> i + 5, 10)
          .Println;
```

Тогда первый элемент последовательности будет равен 10 + 5 = 15:



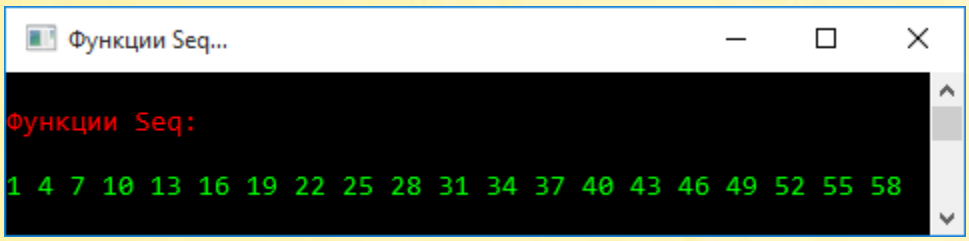
Другая **функция SeqGen с тремя параметрами** возвращает последовательность элементов типа *T*:

```
function SeqGen<T>(count: integer;
                   first: T; next: T -> T): sequence of T;
```

count – число элементов
first – первый элемент последовательности
next – функция для вычисления элементов последовательности. По значению предыдущего элемента последовательности вычисляется следующий.

С её помощью также легко получить **арифметическую прогрессию**:

```
var sq := SeqGen(20, 1, a -> a + 3)
          .Println;
```



Здесь:

20 – число элементов в последовательности
1 – первый элемент
a -> a + 3 – функция для вычисления следующих элементов

Эта функция действует более понятно, чем предыдущая.

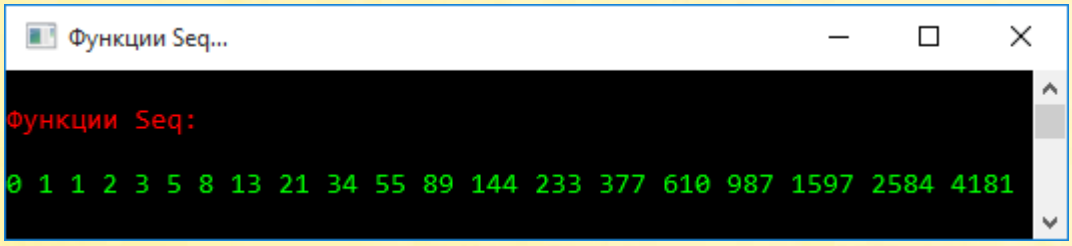
И последняя функция SeqGen - с четырьмя параметрами:

```
function SeqGen<T>(count: integer;
                   first,second: T; next: (T,T) -> T): sequence of T;
```

count – число элементов в последовательности
first – первый элемент последовательности
second – второй элемент последовательности
next – функция для вычисления элементов последовательности. По значениям двух предыдущих элементов последовательности вычисляется следующий.

С помощью этой функции легко получить последовательность чисел Фибоначчи:

```
var sq := SeqGen(20, 0, 1, (a,b) -> a + b)
          .Println;
```



Здесь:

20 – число элементов в последовательности
0 – первый элемент
1 – второй элемент
(a,b) -> a + b – функция для вычисления следующих элементов

a + b = 1 при начальных значениях. Последовательность: 0, 1, 1. Теперь a = b = 1. Дальше:

(1,1) -> 1 + 1 = 2. Последовательность: 0, 1, 1, 2. Теперь a = 1, b = 2.

(1,2) -> 1 + 2 = 3. Последовательность: 0, 1, 1, 2, 3. Теперь a = 2, b = 3.

И так далее.

Функция SeqWhile возвращает последовательность элементов типа *T*:

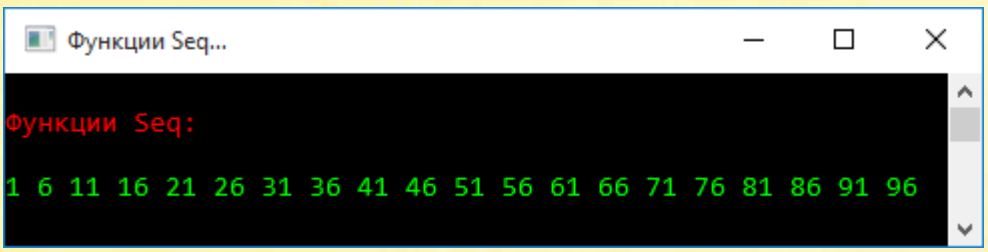
```
function SeqWhile<T>(first: T;
                     next: T -> T; pred: T -> boolean): sequence of T;
```

first – первый элемент последовательности
next – функция для вычисления элементов последовательности
pred – условие прекращения вычислений

Эта функция удобна для создания последовательностей, число элементов которых заранее неизвестно, а определяется заданным условием.

Например, создадим арифметическую последовательность с первым членом 1 и разностью 5. Наибольший член последовательности должен быть меньше 100:

```
var sq := SeqWhile(1, n -> n + 5, n -> n < 100)
          .Println;
```



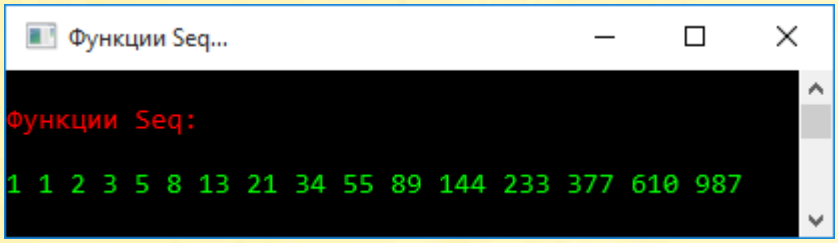
Функция SeqWhile с 4 параметрами полезна при генерировании последовательностей типа чисел Фибоначчи, когда общее количество членов заранее неизвестно:

```
function SeqWhile<T>(first,second: T;
                     next: (T,T) -> T;
                     pred: T -> boolean): sequence of T;
```

first – первый элемент последовательности
second – второй элемент последовательности
next – функция для вычисления следующего элемента последовательности по двум предыдущим
pred – условие прекращения вычислений

Следующий код напечатает числа Фибоначчи, не превышающие 1000:

```
var sq := SeqWhile(1, 1, (n1, n2) -> n1 + n2, n -> n < 1000)
          .Println;
```

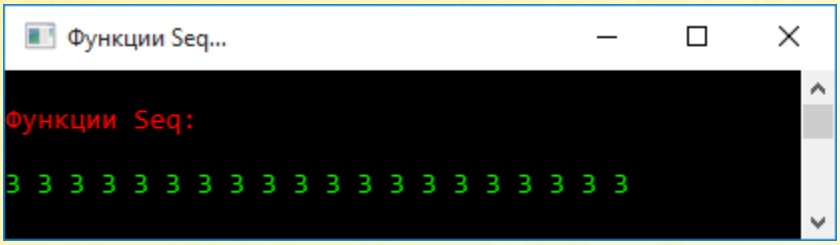


Функция SeqFill возвращает последовательность из *count* одинаковых элементов типа T:

```
function SeqFill<T>(count: integer; x: T): sequence of T;
```

x – значение элемента
count – число элементов

```
SeqFill(20, 3).Println
```



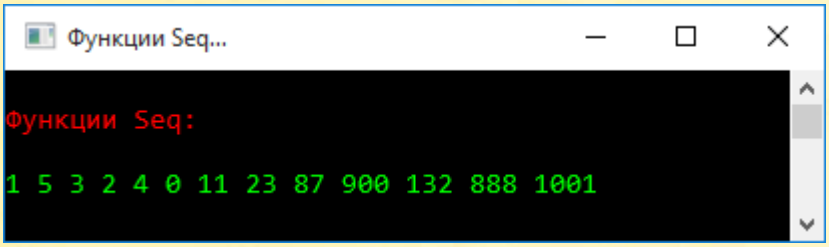
Здесь:

20 – число элементов
3 – значение каждого элемента

Мы рассмотрели методы, которые самостоятельно генерируют последовательности, подчиняющиеся какому-то правилу. Иногда необходимо создать последовательность **произвольных** элементов. Тогда их нужно перечислить в скобках при вызове функции **Seq**:

```
function Seq<T>(params a: array of T): sequence of T;
```

```
Seq(1,5,3,2,4, 0, 11, 23, 87, 900, 132, 888, 1001).Println;
```

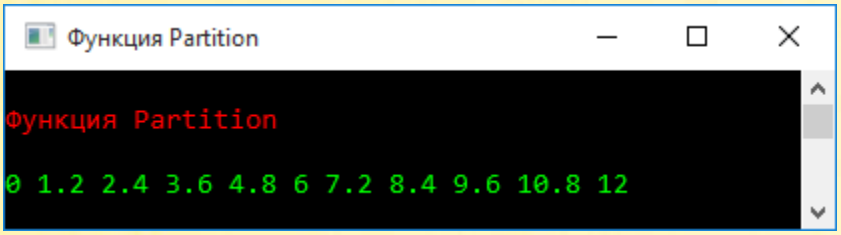


Функция Partition

Функция Partition возвращает последовательность вещественных чисел в диапазоне **a**,..**b**, разделённом на **n** частей:

```
function Partition(a,b: real; n: integer): sequence of real;
```

```
var sqr := Partition(0.0, 12.0, 10)
           .Println;
```



По сути, это арифметическая последовательность, в которой:

a – первый член
n + 1 – число членов
(a – b) / n – разность

Ввод последовательностей с клавиатуры

Для ввода последовательностей из *n* целочисленных элементов с клавиатуры служат **функции ReadSeqInteger**:

```
function ReadSeqInteger(n: integer): sequence of integer;
```

```
function ReadSeqInteger(const prompt: string; n: integer):
                        sequence of integer;
```

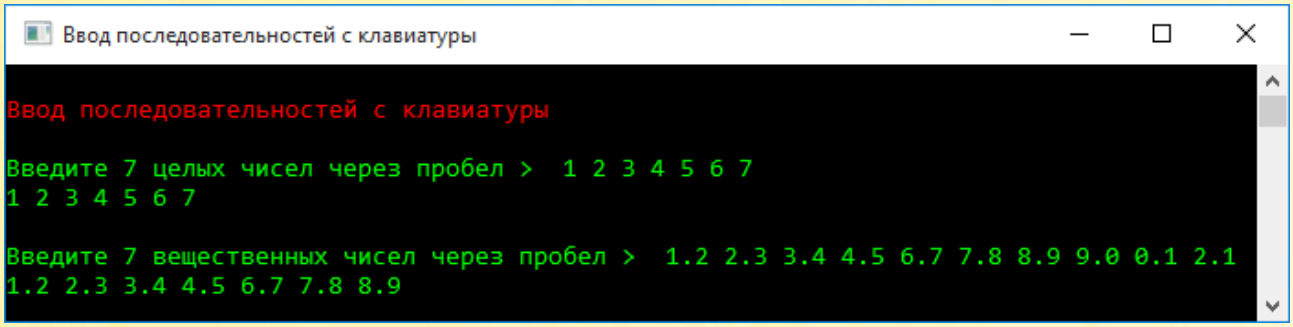
Вторая функция дополнительно печатает на экране поясняющую строку **prompt**.

Аналогичные функции для **вещественных** чисел:

```
function ReadSeqReal(n: integer): sequence of real;
```

```
function ReadSeqReal(const prompt: string; n: integer):
                     sequence of real;
```

```
var sq := ReadSeqInteger('Введите 7 целых чисел через
                          пробел > ', 7);
sq.Println;
Println;

var sqr := ReadSeqReal('Введите 7 вещественных чисел через
                        пробел > ', 7);
sqr.Println;
Println;
```



Если чисел введено больше, чем указано в вызове функции, то лишние данные отбрасываются.

Функции для строк:

```
function ReadSeqString(n: integer): sequence of string;
```

```
function ReadSeqString(const prompt: string; n: integer):
                       sequence of string;
```

Если число элементов в последовательности заранее неизвестно, то используйте следующие функции, которые завершают ввод данных, когда не выполнится условие cond:

```
function ReadSeqIntegerWhile(cond: integer -> boolean):
                              sequence of integer;

function ReadSeqRealWhile(cond: real -> boolean):
                            sequence of real;

function ReadSeqStringWhile(cond: string -> boolean):
                             sequence of string;

function ReadSeqIntegerWhile(const prompt: string;
                              cond: integer -> boolean):
                              sequence of integer;

function ReadSeqRealWhile(const prompt: string;
                            cond: real -> boolean):
                            sequence of real;

function ReadSeqStringWhile(const prompt: string;
                             cond: string -> boolean):
                             sequence of string;
```

Не советую пользоваться этими функциями в своих программах.

Операции над последовательностями

К последовательностям можно применять различные операции.

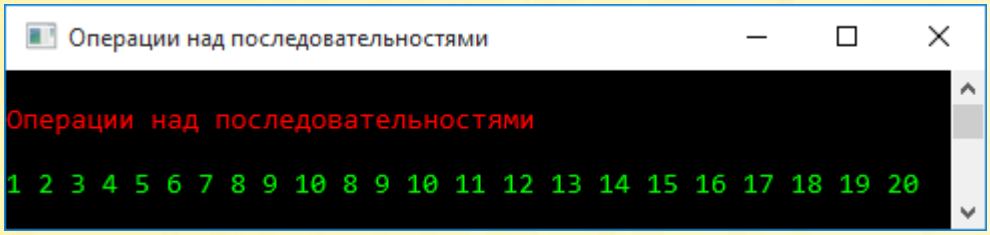
Оператор + и метод *Concat*

Оператор + возвращает последовательность, образованную объединением двух (или больше) последовательностей:

```
function IEnumerable<T>.operator+(a,b: sequence of T): sequence of T;
```

При этом элементы второй последовательности следуют за элементами первой последовательности:

```
var sq1 := Range(1, 10);
var sq2 := Range(8, 20);

var res := sq1 + sq2;
res.Println;
```

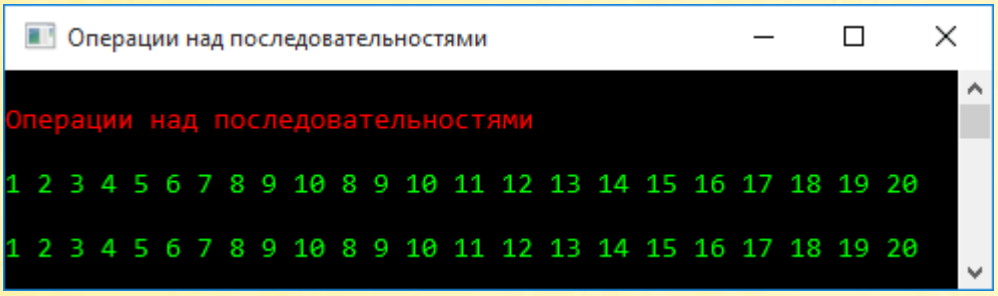


Число элементов в результирующей последовательности равно общему числу элементов в последовательностях-слагаемых.

Эта операция называется **конкатенацией**. Объединить две последовательности можно и с помощью метода расширения Concat, который действует точно так же:

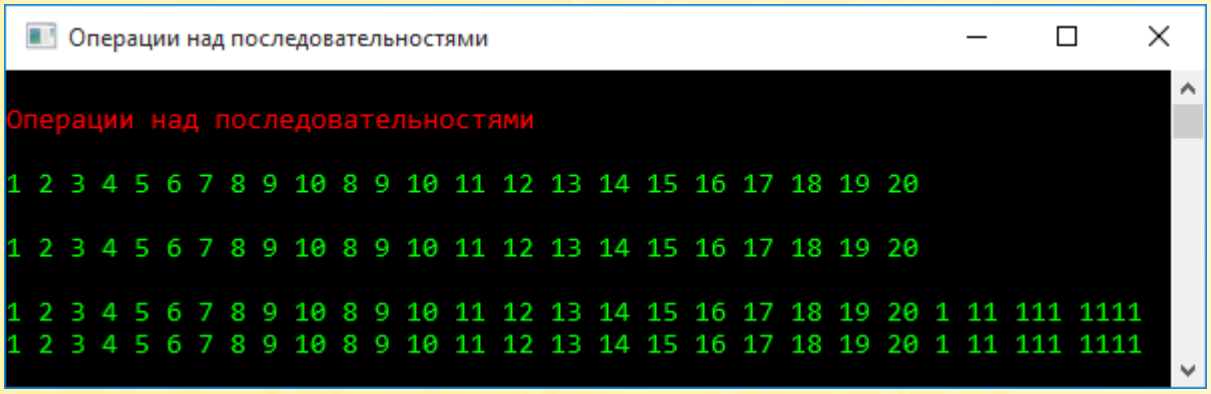
```
function Concat(second: sequence of T): sequence of T;
```

```
var sq1 := Range(1, 10);
var sq2 := Range(8, 20);

var res := sq1 + sq2;
res.Println;

Println;
sq1.Concat(sq2)
   .Println;
```



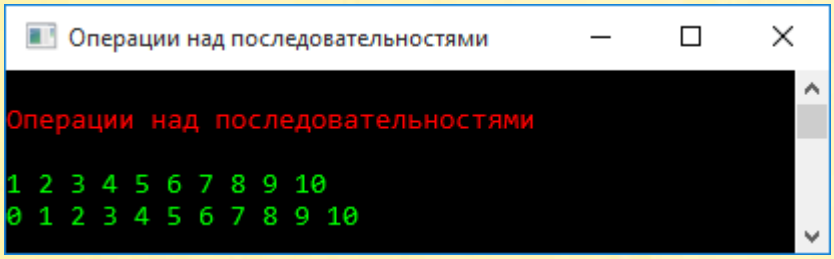
Примеры конкатенации трёх последовательностей:

```
Println;
var sq3 := Seq(1,11,111,1111);
(sq1 + sq2 + sq3).Println;
sq1.Concat(sq2).Concat(sq3).Println;
```



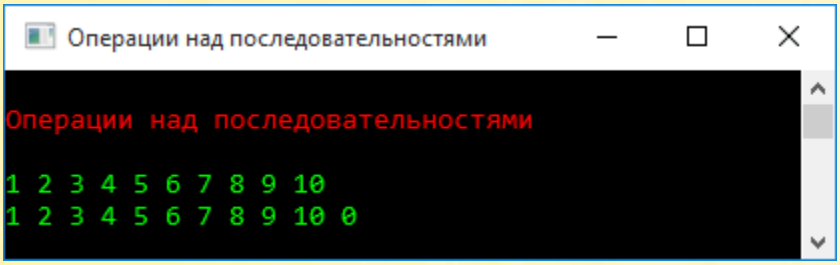
К последовательности можно добавить элемент совместимого типа спереди:

```
var sq1 := Range(1, 10).Println;
var res:= 0 + sq1;
res.Println;
```



Или сзади:

```
var sq1 := Range(1, 10).Println;
var res:= sq1 + 0;
res.Println;
```



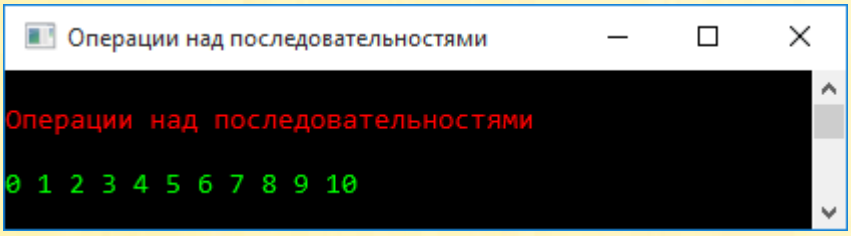
Оператор *

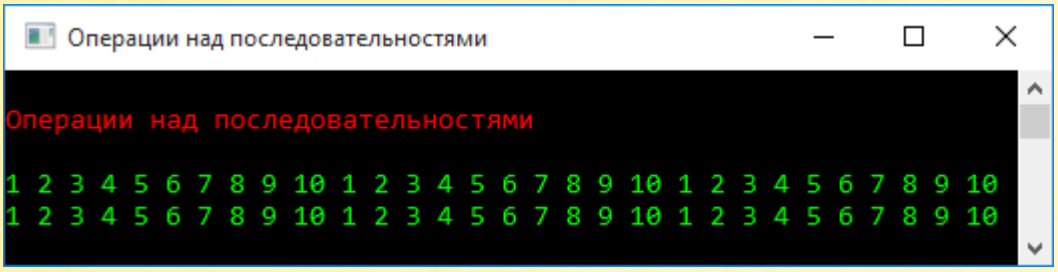
Оператор * возвращает последовательность, образованную повторением последовательности заданное число раз:

```
function IEnumerable<T>.operator*(n: integer; a: sequence of T):
                                        sequence of T;
```

Числовой множитель может занимать место и первое, и второе место в выражении:

```
var sq1 := Range(1,
10);
var res:= 3 * sq1;
res.Println;

res:= sq1 * 3;
res.Println;
```



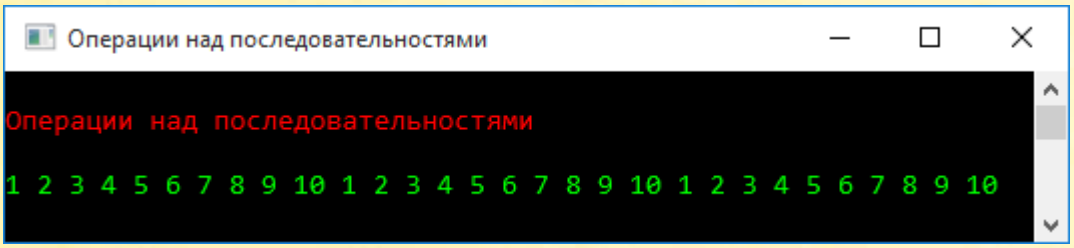


Метод расширения Cycle повторяет заданную последовательность бесконечно:

```
function Cycle<T>(Self: sequence of T): sequence of T;
                  extensionmethod;
```

Так как бесконечную последовательность напечатать невозможно, то мы ограничимся первыми 30-ю элементами:

```
var sq1 := Range(1, 10);
sq1.Cycle.Take(30).Println;
```



Встроенные методы расширения для последовательностей

Последовательности нужны для того, чтобы извлекать из них полезную информацию. И тут нам не обойтись без **методов расширения**.

Метод *Where*

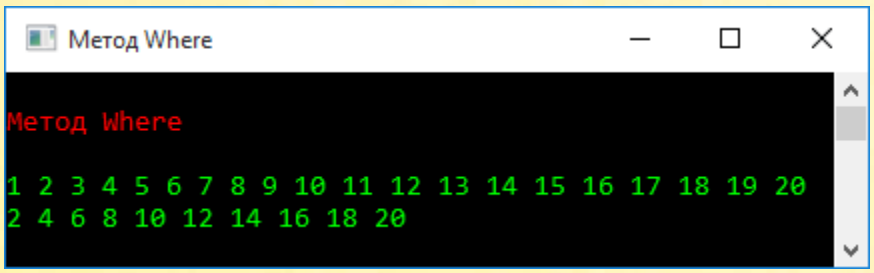
Метод Where отбирает элементы в соответствии с условием, заданным функцией *predicate*:

```
function Where(predicate: T->boolean): sequence of T;
```

Иначе говоря, он пропускает дальше только те элементы исходной последовательности, которые удовлетворяют условию.

Создадим последовательность из 20 натуральных чисел и отфильтруем **чётные**:

```
var sq := Range(1,20).Println;
sq.Where(n -> n mod 2 = 0).Println;
```

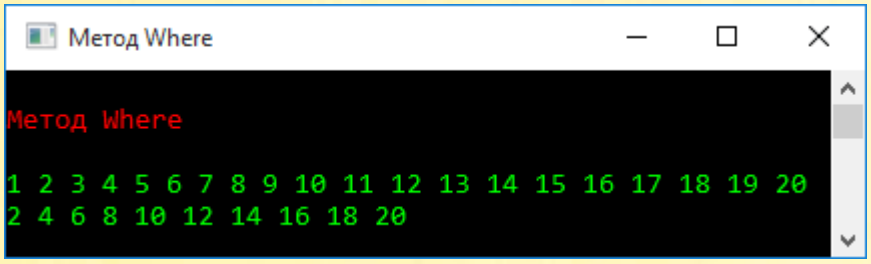


Второй вариант **метода Where** также фильтрует элементы в соответствии с условием, заданным функцией *predicate*, но в ней можно использовать *индекс* элемента в исходной последовательности:

```
function Where(predicate: (T,integer)->boolean): sequence of T;
```

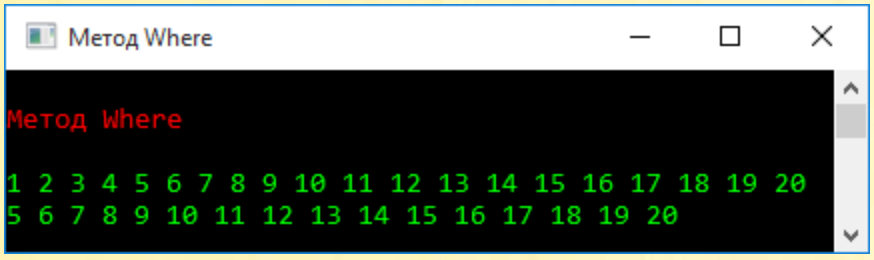
Если индекс элементов не используется, то второй метод *Where* вернёт ту же самую последовательность чётных чисел:

```
var sq := Range(1,20).Println;
sq.Where((n, i) -> n mod 2 = 0).Println;
```



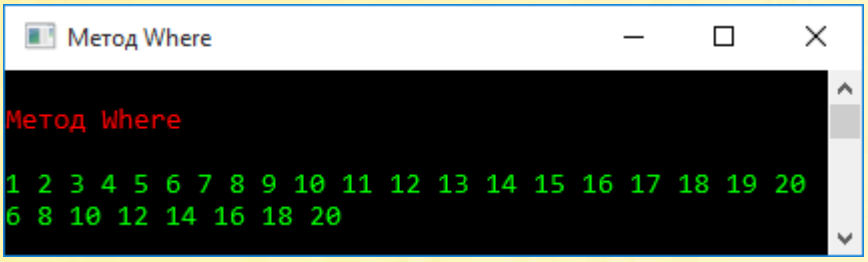
Можно отфильтровать элементы только по их *индексам*:

```
var sq := Range(1,20).Println;
sq.Where((n, i) -> i > 3).Println;
```



И наконец, в условие могут входить как индексы, так и значения элементов:

```
var sq := Range(1,20).Println;
sq.Where((n, i) -> (i > 3) and (n mod 2 = 0)).Println;
```



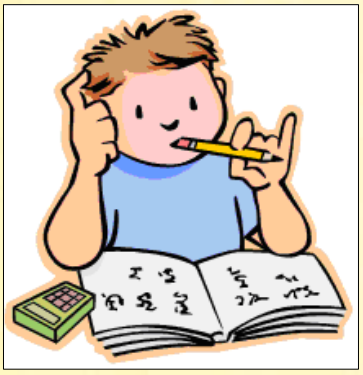
В этом примере мы отфильтровали чётные числа, индексы которых больше трёх.

Метод **Where** очень часто используется при решении задач, в том числе и занимательных.

Трёхзначное число (Range.Where)

В книге *600 задач на сообразительность*, на странице 112 напечатана **задача 41**:

> *Трёхзначное число*
> *Если от трёхзначного числа отнять 7, то оно разделится на 7; если отнять от него 8, то оно разделится на 8; если отнять от него 9, то оно разделится на 9. Какое это число?*



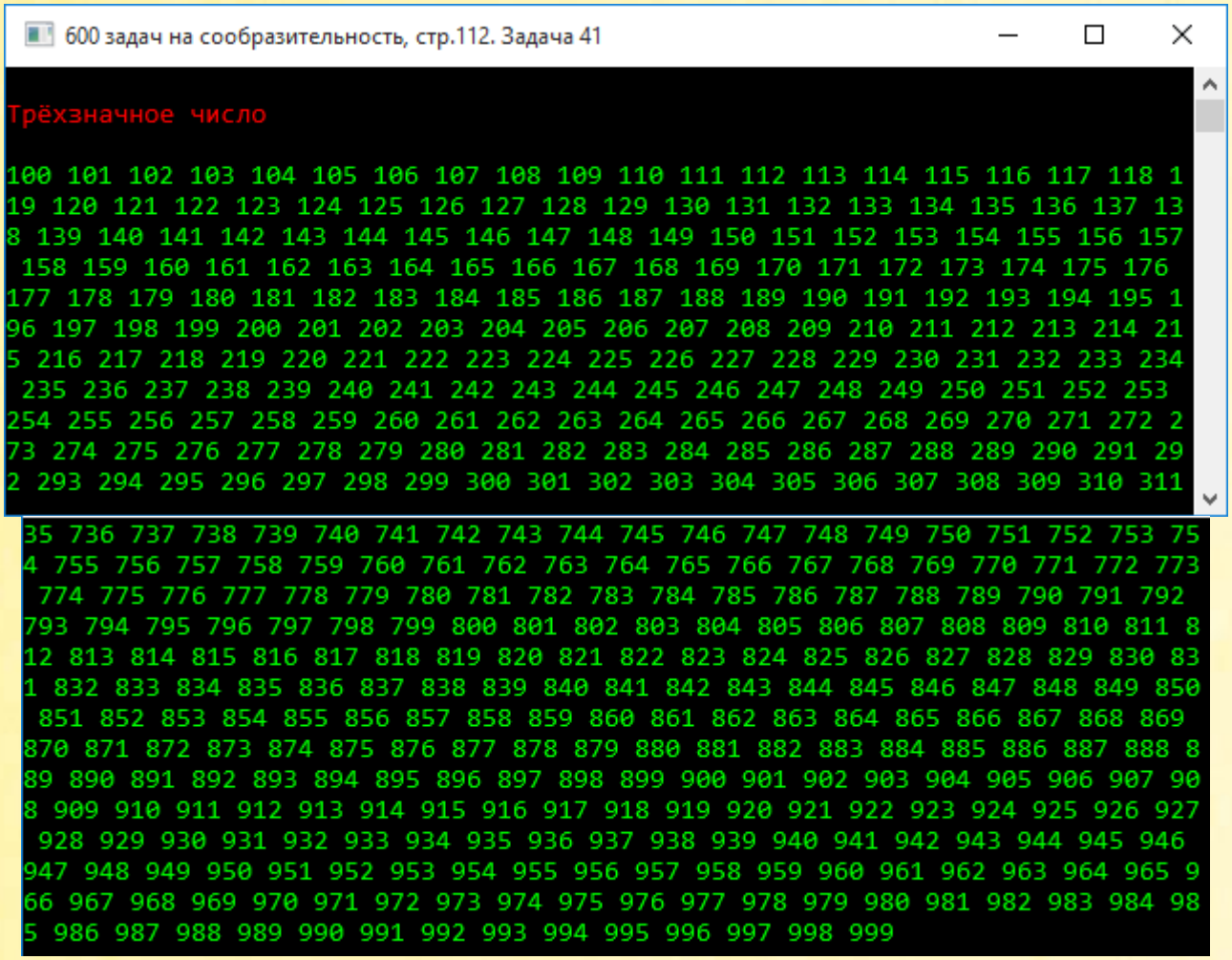
Задача очень простая, и легко решается в уме. Но и для тренировки сгодится!

Из условия задачи следует, что искомое число – **трёхзначное**. Мы легко получим последовательность всех трёхзначных чисел с помощью функции **Range**:

```
Range(100,999)
```

В этом легко убедиться, если **распечатать** последовательность на экране:

```
Range(100,999).Println;
```



Итак, мы получили все трёхзначные числа и теперь должны выбрать такое число (или, возможно, числа, если решение не единственное), которое удовлетворяет условию задачи.

Условие довольно длинное, но простое, поэтому мы можем применить к последовательности метод расширения Where, которому следует передать лямбда-выражение, то есть функцию-предикат, которая «отфильтрует» из последовательности только те элементы, которые удовлетворяют заданному условию. Если мы обозначим буквой *n* текущий элемент последовательности, то условие можно записать так:

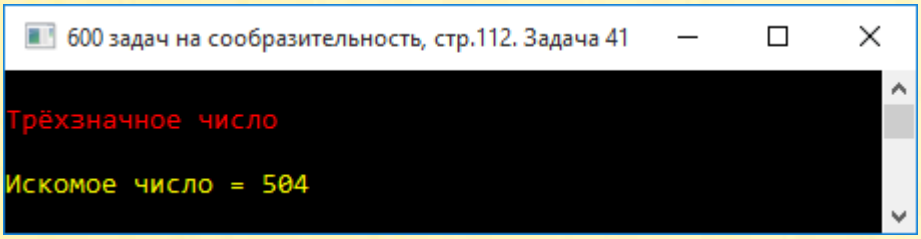
```
((n - 7) mod 7 = 0) and
((n - 8) mod 8 = 0) and
((n - 9) mod 9 = 0))
```

То есть одновременно должны быть верными 3 логических выражения, что и требуется в задаче.

Всё решение укладывается в одну строку. Но строка получается длинная, поэтому для удобства её разбивают на более короткие логические части. Так одна длинная физическая строка превращается в 4 физические:

```
uses
   System;

// 600 задач на сообразительность, стр.112. Задача 41


begin
   // заголовок окна:
   Console.Title := '600 задач на сообразительность,
                     стр.112. Задача 41';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Трёхзначное число');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   Range(100,999).Where(n -> ((n - 7) mod 7 = 0) and
                             ((n - 8) mod 8 = 0) and
                             ((n - 9) mod 9 = 0))
                 .Println;

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Ответ мы, естественно, получим тот же самый, что при решении «классическим» способом:



Пятизначное число (Range.Where)

Задача 440 из книги *Математическая шкатулка* [Нагибин88]:

> Мне было задано пятизначное число. К этому числу нужно было прибавить 200 000 и сумму умножить на 3. Вместо этого я приписал к заданному мне числу в конце его справа цифру 2 и получил правильный результат.
>
> Какое число было мне задано?

Пятизначных чисел совсем немного, поэтому мы легко выберем их из последовательности, генерируемой методом Range.

По условию задачи, к заданному пятизначному числу *i* следовало прибавить 200 000, а получившуюся сумму – умножить на 3. Вместо этого нерадивый ученик приписал в конце числа *i* двойку. А мы должны умножить заданное число на 10 и добавить 2.

Записываем условие задачи в методе Where:

```
Where(i ->  (i + 200000) * 3 = (i * 10 + 2))
```

Когда эти два числа совпадут, задача будет решена:

```
uses
   System;

//Нагибин 440


begin
   //заголовок окна:
   Console.Title := 'Нагибин 440';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Пятизначное число');
```

```
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    // решаем задачу:
    Range(10000, 99999)
    .Where(i ->  (i + 200000) * 3 = (i * 10 + 2))
    .Println;

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

На всякий случай мы на этом не останавливаемся и проверяем пятизначные числа до полного их исчерпания. Впрочем, как показывает рисунок, задача имеет *единственное* решение, а искомое число – 85714:

Шестизначный перенос (Range.Where)

Задача 557 из книги *Математическая шкатулка* [Нагибин88], страница 94:

Первая слева цифра шестизначного числа – 1. Если её перенести с первого места в конец числа, сохранив порядок остальных цифр, то вновь полученное число будет втрое больше первоначального.

Восстановите первоначальное число.

В этой задача нужно перенести **первую** цифру шестизначного числа. Дело хлопотное, поэтому мы пишем 3 функции:

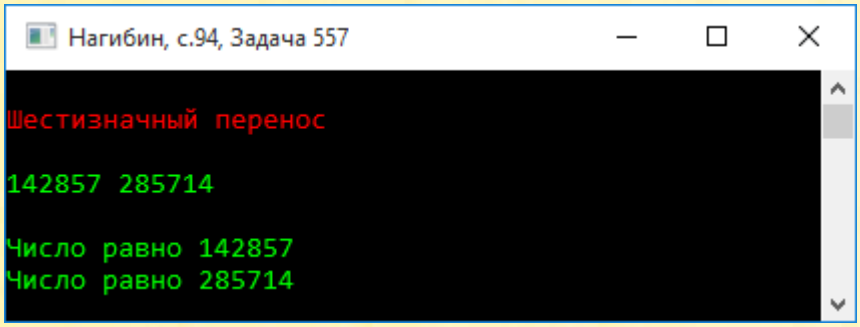
```
// первая цифра числа:
var start: integer -> integer := x -> x div 100000;
// 5-значное число из последних 5 цифр числа x:
var end5: integer -> integer := x -> x mod 100000;
// новое 6-значное число:
var newnum: integer -> integer := x -> end5(x) * 10 + start(x);
```

Зато теперь условие в методе **Where** стало простым и понятным:

```
uses
   System;

//Нагибин, с.94, Задача 557


//мин. и макс. 6-значные числа:
const
   MIN6 = 100000;
   MAX6 = 999999;

begin
   // заголовок окна:
   Console.Title := 'Нагибин, с.94, Задача 557';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Шестизначный перенос');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу -->

   // первая цифра числа:
   var start: integer -> integer := x -> x div 100000;
   // 5-значное число из последних 5 цифр числа x:
   var end5: integer -> integer := x -> x mod 100000;
   // новое 6-значное число:
   var newnum: integer -> integer := x -> end5(x) * 10 + start(x);
```

```
    var res:= Range(MIN6, MAX6)
              .Where(n -> newnum(n) = 3 * n)
              .Println;

    // печатаем ответ:
    Console.WriteLine();
    Console.WriteLine('Число равно ' + res.ElementAt(0));
    Console.WriteLine('Число равно ' + res.ElementAt(1));

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Среди 6-значных чисел, начинающихся с единицы, первоначальное число – *единственное*. Есть ещё одно подобное число, но оно начинается с двойки:

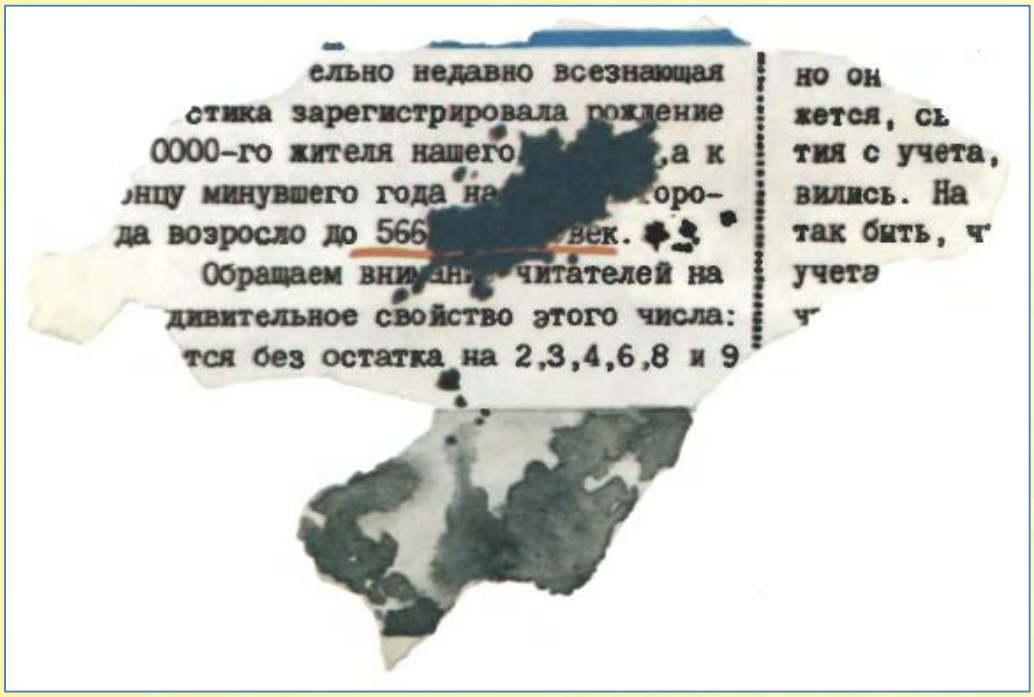


Всезнающая статистика (Range.Where)

Задача 14 (16) из книги *Удивительный мир чисел* [КА86], страницы 86-87:

Попал как-то мне в руки обрывок прошлогодней газеты. Моё внимание привлекло чернильное пятно, закрывшее последние три цифры шестизначного числа (Рис. 2.8). По сохранившемуся кусочку текста я вспомнил: это была заметка, в которой сообщалось, что к концу минувшего года население нашего города возросло до этого числа. В заметке говорилось также о том, что это шестизначное число уникально среди шестизначных: оно делится на 2, 3, 4, 6, 7, 8 и 9. Ого! Не правда ли?

Чтобы восстановить все цифры этого числа, нет нужды обращаться в реставрационную лабораторию. Собственная сообразительность подскажет вам математический метод быстрого решения этой задачи.



Эта задача очень простая, поскольку нам предстоит проверить всего одну тысячу чисел. Действительно, на обрывке газеты видно, что первые три цифры 6-значного числа – 566. Последние три – от 000 до 999:

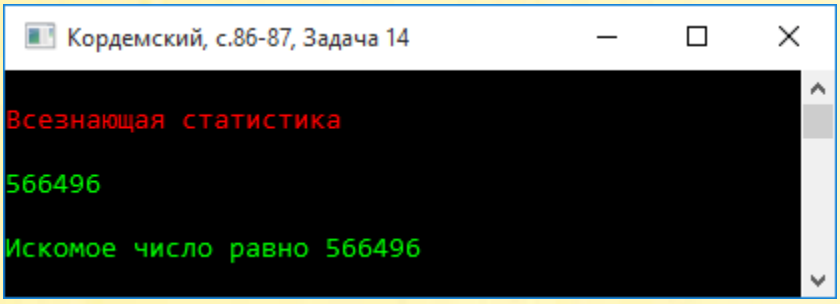
```
const
   //мин. и макс. 6-значные числа:
   MIN6 = 566000;
   MAX6 = 566999;
```

Метод Range генерирует нужные нам числа, а метод Where проверяет их - они должны нацело делиться на числа 2,3,4,6,7,8 и 9:

```
uses
   System;

//Кордемский, с.86-87, Задача 14

const
   //мин. и макс. 6-значные числа:

   MIN6 = 566000;
   MAX6 = 566999;

begin
   //заголовок окна:
   Console.Title := 'Кордемский, с.86-87, Задача 14';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Всезнающая статистика');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

    // решаем задачу:
    var res:= Range(MIN6, MAX6)
             .Where(num ->  (num mod 2 = 0) and
                            (num mod 3 = 0) and
                            (num mod 4 = 0) and
                            (num mod 6 = 0) and
                            (num mod 7 = 0) and
                            (num mod 8 = 0) and
                            (num mod 9 = 0))
             .Println;
```

```
      // печатаем ответ:
      Println;
      Console.WriteLine('Искомое число равно ' + res.First);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Ответ вы можете видеть на рисунке:



Восстановите потерянную цифру (Range.Where)

Задача 7 из книги *Удивительный мир чисел* [КА86], страница 85:

Числа вида $M_p = 2^p - 1$, где p - простое число, называются *числами Мерсенна*. При некоторых значениях *p* M_p - простое число. Так, первые одиннадцать простых чисел Мерсенна получаются при значениях *p* = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107.

Двенадцатое простое число Мерсенна равно $M_{127} = 2_{127} - 1$.

В его десятичной записи одна цифра стёрлась. Мы пока заменили её буквой *x*. Получилась такая запись:

170 141 183 460 469 231 *x*31 687 303 715 884 105 727.

Восстановите цифру, замещенную буквой *x*, если известно, что $M_{127} + 3$ делится на 13.

Задача решается очень просто, если вспомнить признак делимости чисел на 13. Цифра, обозначенная буквой ***x***, может принимать значения от 0 до 9. Мы их получим в методе Range:

```
Range(0, 9)
```

Для каждого значения *x* мы находим арифметическую сумму трёхзначных чисел – как и предписывает нам признак делимости на 13:

```
var summa: integer -> integer := x -> 170 - 141 + 183
                                    - 460 + 469 - 231
                                    + x * 100 + 31 - 687
                                    + 303 - 715 + 884
                                    - 105 + 727 + 3;
```

К получившейся сумме нужно добавить тройку. Если при этом функция summa возвращает число, которое делится на 13, то задача решена:

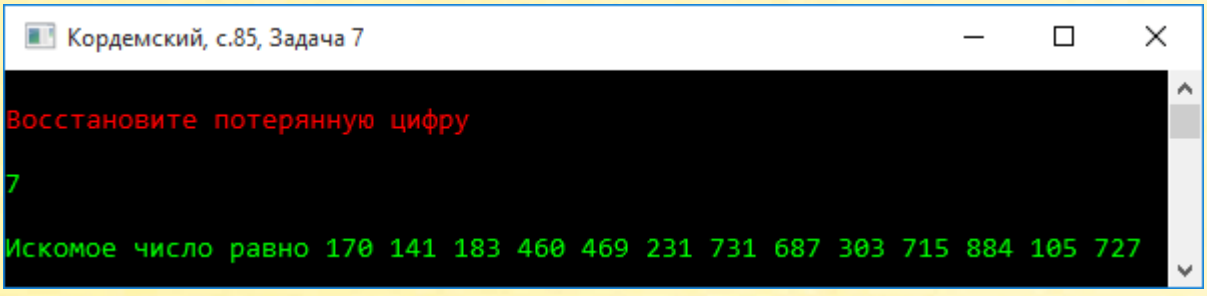
```
Where(n ->  summa(n) mod 13 = 0)
```

Вся программа целиком:

```
uses
   System;

// Кордемский, с.85, Задача 7


begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.85, Задача 7';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Восстановите потерянную цифру');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();
```

```
    var summa: integer -> integer := x -> 170 - 141 + 183
                                         - 460 + 469 - 231
                                         + x * 100 + 31 - 687
                                         + 303 - 715 + 884
                                         - 105 + 727 + 3;
    // решаем задачу:
    var x:= Range(0, 9)
           .Where(n ->  summa(n) mod 13 = 0)
           .Println;
    Println;
    // печатаем искомое число:
    Console.WriteLine('Искомое число равно 170 141 183 460 469
                           231 {0}31 687 303 715 884 105 727', x.First);

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

На рисунке вы видите, что x = 7:



Снимите маску с одной цифры (Range.Where)

Задача 6 из книги *Удивительный мир чисел* [КА86], страница 84:

> В записи знаменитого «шахматного» числа
>
> M_{64}= 1y446 744 073 709 551 615
>
> на его вторую цифру накинута маска *y*. Сняв маску, **расшифруйте значение y**, зная, что достаточно увеличить заданное число на 3 единицы, как оно становится кратным числу 19.
>
> **Примечание**. По легенде именно такое число пшеничных зёрен следовало выдать в награду изобретателю шахмат, попросившему положить всего одно зерно на первую клетку шахматной доски, а на каждую следующую клетку вдвое большее число зёрен, чем на предыдущую.

Эта задача сродни предыдущей, но признак делимости чисел на 19 не очень удобен. Поэтому в методе **Where** мы просто находим остаток от деления числа *n + 3* на 19:

```
Where(n -> num(n) mod 19 = 0)
```

Однако мы должны учесть, что искомое число очень велико, поэтому параметр функции **num** должен иметь тип *decimal*:

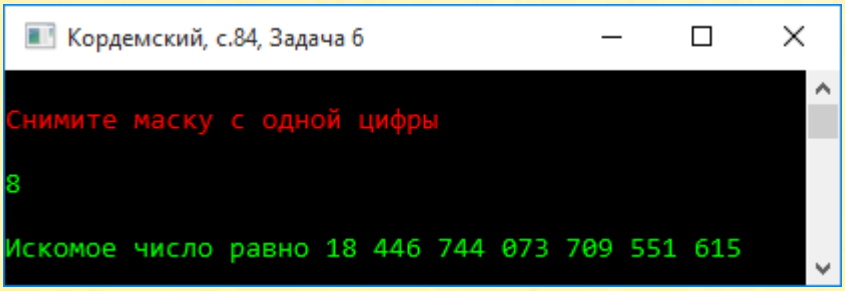
```
var num: decimal-> decimal := y -> (10 + y) * 100000000000000000 +
                                              446744073709551615 + 3;
```

Других сложностей в задаче нет:

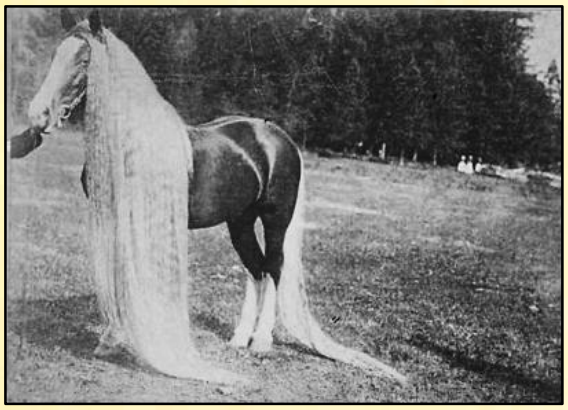
```
uses
   System;
```

```
// Кордемский, с.84, Задача 6


begin
    // заголовок окна:
    Console.Title := 'Кордемский, с.84, Задача 6';
    Console.WriteLine('');
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Снимите маску с одной цифры');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    var num: decimal-> decimal := y -> (10 + y) *
                      1000000000000000000 +
                      446744073709551615 + 3;

    // решаем задачу:
    var y := Range(0, 9)
            .Where(n -> num(n) mod 19 = 0)
            .Println;
    Println;
    // печатаем искомое число:
    Console.WriteLine('Искомое число равно 1{0}
                      446 744 073 709 551 615', y.First);

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

И мы быстро находим, что y = 8:



И «хвост», и «грива» (Range.Where)



Задача 8 из книги *Удивительный мир чисел* [КА86], страница 72:

> Если есть четырёхзначные числа, первые две и последние две цифры каждого из которых совпадают соответственно с первыми двумя и последними двумя цифрами квадрата и куба искомого числа, то какое из них наименьшее и какое - наибольшее? Числа, оканчивающиеся единицей и двумя нулями, исключаем.
>
> В зашифрованном виде:
>
> $xyzt^2 = xy...zt$ и
>
> $xyzt^3 = xy...zt$.
>
> В правой и левой частях этих равенств буквой ***x*** зашифрована одна и та же цифра, буквой ***y*** - также, буквой ***z*** - также и буквой ***t*** - также, но эти цифры могут быть и одинаковыми.

Чтобы получить из квадратов и кубов две первые и две последние цифры (точнее, двузначные числа, составленные из этих цифр), мы напишем вспомогательные функции.

Две последние цифры легко получить как остаток от деления исходного числа на сотню:

```
var Last2: integer -> integer := num -> num mod 100;
```

Так как кубы четырёхзначных чисел слишком велики для типа *integer*, то для квадратов и кубов необходимо написать отдельные функции:

```
var Last2Quadrat: integer -> integer := num -> num * num mod 100;
var Last2Cube: int64 -> int64 := num -> num * num * num mod 100;
```

С первыми двумя цифрами сложнее: мы не знаем «длину» числа, поэтому и не сможем сразу разделить его на нужную степень числа 10. В этом случае нужно последовательно делить заданное число на 10, пока оно не станет двузначным:

```
function First2(num: int64) : int64;
begin
    while (num >= 100) do
        num := num div 10;
    Result:= num;
end;
```

Для квадратов и кубов опять необходимы функции:

```
var First2Quadrat: int64 -> int64 := num -> First2(num*num);
var First2Cube: int64 -> int64 := num -> First2(num*num*num);
```

От метода Range мы получаем все четырёхзначные числа:

```
Range(1000,9999)
```

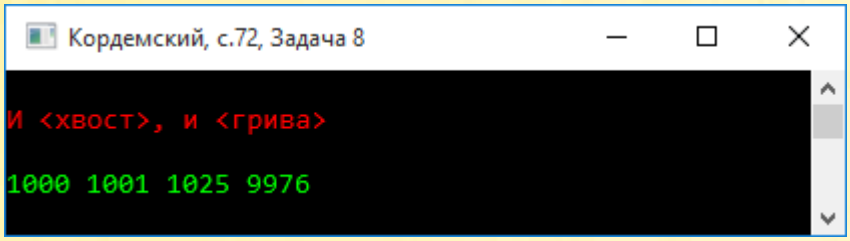
Метод Where пропускает только такие числа, которые удовлетворяют условиям задачи:

```
Where(n -> (First2(n) = First2Quadrat(n)) and
           (First2(n) = First2Cube(n))    and
           (Last2(n) = Last2Quadrat(n))   and
           (Last2(n) = Last2Cube(n)))
```

А метод Println печатает все профильтрованные числа:

```
uses
   System;

// Кордемский, с.72, Задача 8


    function First2(num: int64) : int64;
    begin
        while (num >= 100) do
            num := num div 10;
        Result:= num;
    end;

    var First2Quadrat: int64 -> int64 := num -> First2(num*num);
    var First2Cube: int64 -> int64 := num -> First2(num*num*num);

    var Last2: integer -> integer := num -> num mod 100;
    var Last2Quadrat: integer -> integer := num -> num * num mod 100;
    var Last2Cube: int64 -> int64 := num -> num * num * num mod 100;

begin
    // заголовок окна:
    Console.Title := 'Кордемский, с.72, Задача 8';
    Console.WriteLine('');
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('И «хвост», и «грива»');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    Range(1000,9999).Where(n -> (First2(n) = First2Quadrat(n)) and
                                (First2(n) = First2Cube(n))    and
                                (Last2(n) = Last2Quadrat(n))   and
                                (Last2(n) = Last2Cube(n)))
                    .Println;

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Запустив программу, мы тут же получаем числа:



Первые два числа нарушают условия задачи, поэтому остаются 2 последних числа. Первое из них – 1025 – наименьшее из возможных, а второе – 9976 – наибольшее.

Ж-Ж-Ж! (Range.Where)

Задача 3 из книги *Удивительный мир чисел* [КА86], страница 71:

$$(Ж-1)^5=\overline{Ж\ Ж\ Ж(Ж-1)}$$

Число в левой части равенства не меньше 1, поэтому значение цифры Ж изменяется от 2 до 9. Метод Range выдаёт нам однозначные числа:

```
Range(2,9)
```

Метод Where проверяет их на соответствие условию задачи:

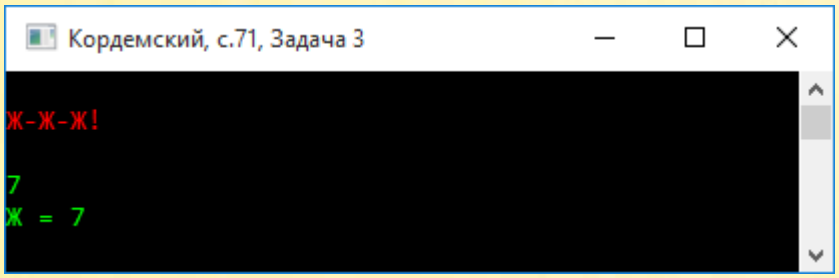
```
Where(n -> (n-1)*(n-1)*(n-1)*(n-1)*(n-1) =
            n * 1000 + n * 100 + n * 10 + n - 1)
```

Метод Println печатает все подходящие значения. От него мы узнаём, что решение *единственное*. Печатаем его на экране:

```
uses
   System;

// Кордемский, с.71, Задача 3


begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.71, Задача 3';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Ж-Ж-Ж!');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var res:= Range(2,9)
            .Where(n -> (n-1)*(n-1)*(n-1)*(n-1)*(n-1) =
                        n * 1000 + n * 100 + n * 10 + n - 1)
            .Println;

   // печатаем ответ:
   Console.WriteLine('Ж = ' + res.First);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

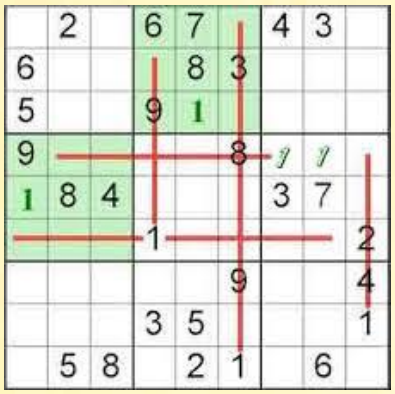
На рисунке вы видите, что Ж = 7, то есть полное решение такое:

$(7-1)^5 = 7776$



Девять в квадрате (Range.Where)

Задача 1 из книги *Удивительный мир чисел* [КА86], страница 70:



> **Найдите шестизначное число**, зашифрованное в ребусе:
>
> ДЕВЯТЬ^2=§§§§§§ДЕВЯТЬ

Мы легко найдём, что **наименьшее** 6-значное число, состоящее из разных цифр, равно 102345, а **наибольшее** – 987654. Таким образом, нужно перебрать все числа в этом диапазоне, возвести их в квадрат и сравнить последние 6 цифр квадрата с исходным числом. По условию задачи, они должны совпадать.

Метод **Range** легко выдаст нам нужную последовательность чисел:

```
Range(102345, 987654)
```

А вот с проверкой в методе **Where** возникают проблемы из-за больших чисел. Приходится вводить дополнительную **функцию** для проверки условия задачи:

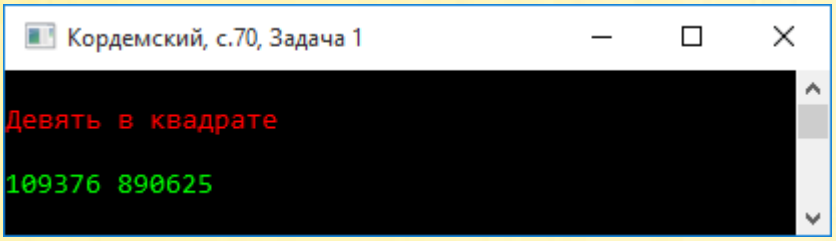
```
var f: int64 -> boolean := n -> n * n mod 1000000 = n;
```

Зато в методе **Where** проверка упрощается:

```
Where(n -> f(n) = true)
```

Метод **Println** печатает все варианты решения.

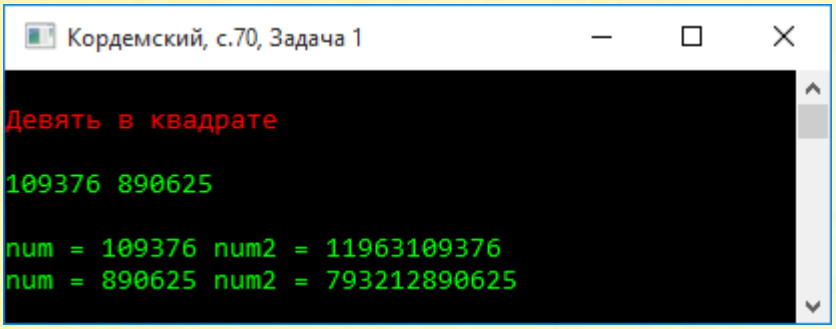
Задача имеет **2 решения**:



Напечатаем их *красиво*:

```
// печатаем ответ:
foreach var num: int64 in res do
begin
   var s := 'num = ' + num.ToString;
   s += ' num2 = ' + (num * num).ToString();
   Console.WriteLine(s);
end;
```

Теперь хорошо видно, что задача решена верно:



Вся программа целиком:

```
uses
   System;

/ /Кордемский, с.70, Задача 1


begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.70, Задача 1';
   Console.WriteLine('');
```

```
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Девять в квадрате');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var f: int64 -> boolean := n -> n * n mod 1000000 = n;
   var res := Range(102345, 987654)
              .Where(n -> f(n) = true)
              .Println();
   Println();

   // печатаем ответ:
   foreach var num: int64 in res do
   begin
      var s := 'num = ' + num.ToString;
      s += ' num2 = ' + (num * num).ToString();
      Console.WriteLine(s);
   end;
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Число 117 649 (Range.Where)

Задача 3.6 из книги *Удивительный мир чисел* [КА86], страница 44:

Число 117 649 существует одновременно в трёх качествах. Оно:

- *квадратное*
- *кубическое*
- *кратное семи*:

$117\,649 = 343^2 = 49^3 = 7k,\ k \in N.$

Более того, на отрезке от единицы до миллиона оно единственное с таким свойством.

Докажите!

Чтобы доказать это утверждение, нужно проверить все числа в заданном диапазоне (но мы расширим диапазон, чтобы найти и другие числа):

```
const MIN = 7;
      MAX = 1000000000;
```

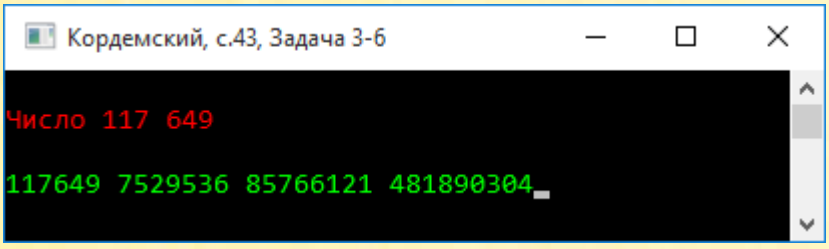
Чтобы не проверять делимость чисел на 7, можно генерировать только 1/7 часть чисел из диапазона MIN..MAX:

```
Range(MIN, MAX, 7)
```

Проверку числа на *квадратность* и *кубичность* мы доверим двум методам расширения из модуля OlympUnit:

```
uses
   System, OlympUnit;

Where(num -> num.IsQuadrat and num.IsCube)
```

Если очередное число num выдержало все проверки, то оно является решением задачи:

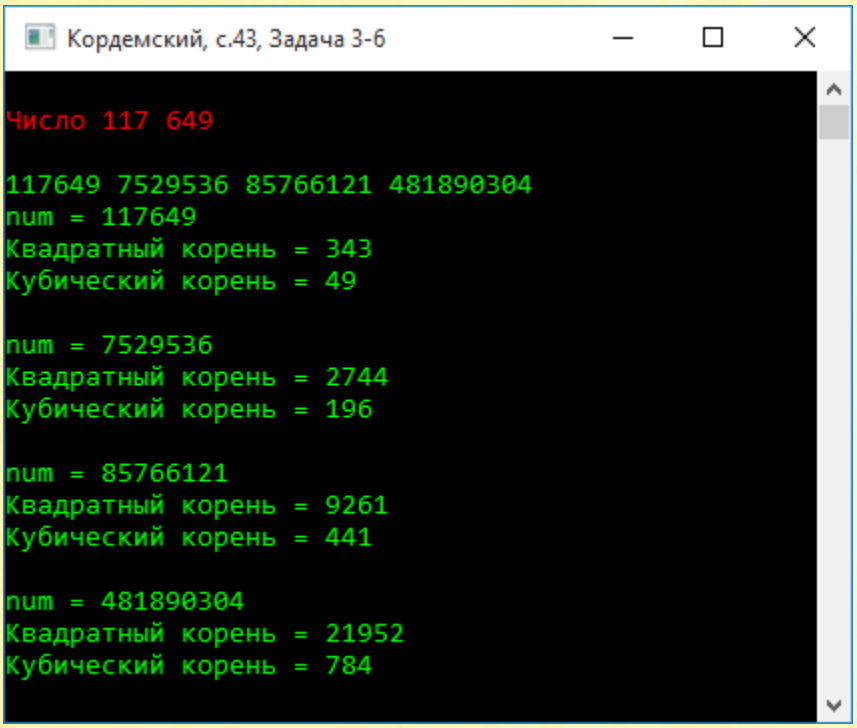


Красиво печатаем ответ:

```
   // печатаем ответ:
   Println;
   foreach var num: int64 in res do
   begin
      var s := 'num = ' + num.ToString();
```

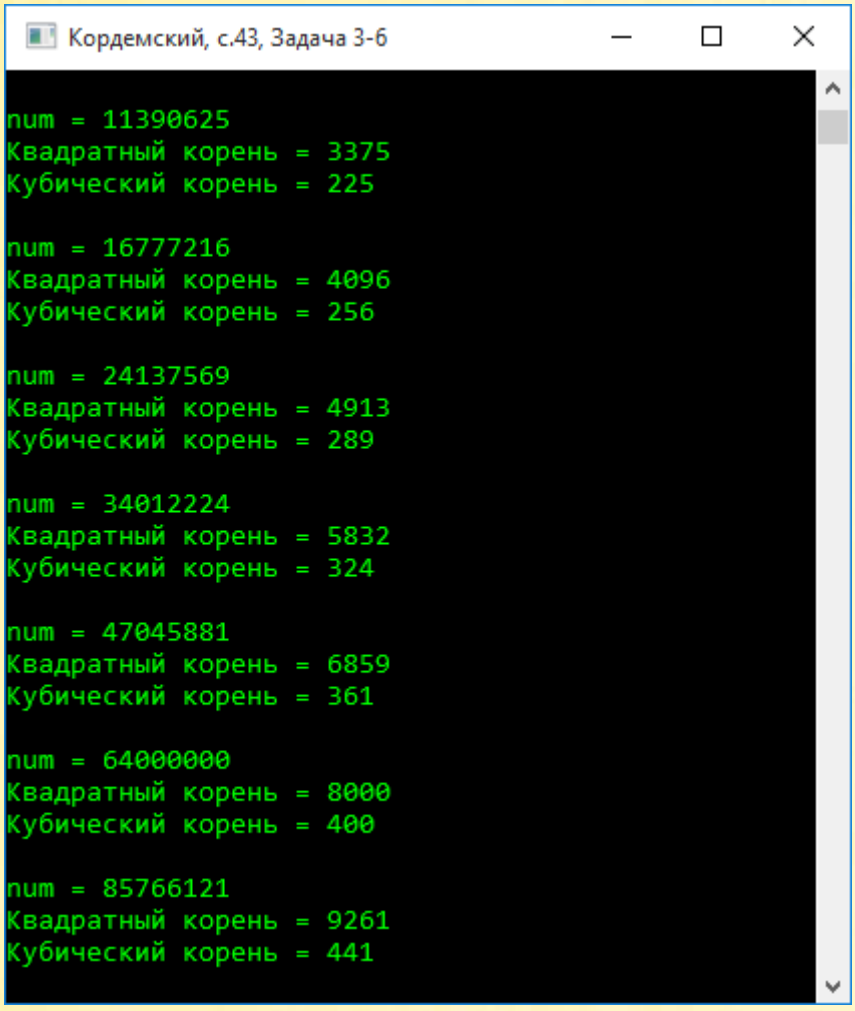
```
    Console.WriteLine(s);
    s := 'Квадратный корень = ' +
         Round(Power(num, 1.0 / 2.0));
    Console.WriteLine(s);
    s := 'Кубический корень = ' +
         Round(Power(num, 1.0 / 3.0));
    Console.WriteLine(s);
    Console.WriteLine();
end;
```

Рисунок ниже показывает, что первое число с указанными в задаче свойствами, - 117 649, а второе – 7 529 536 – значительно больше 1 миллиона. А всего среди первого миллиарда чисел только 4 кратны 7 и являются полными квадратами и кубами:



Если не требовать, чтобы числа были кратны семи, то можно найти ещё несколько любопытных чисел:

```
var res := Range(MIN, MAX, 1)
          .Where(num -> num.IsQuadrat and num.IsCube)
          .Println;
```



Или в более наглядной форме:

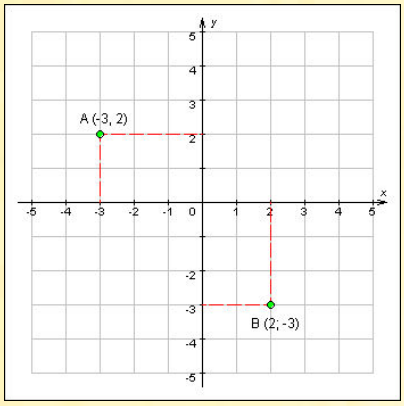
- $3375^2 = 225^3 = 11390625$
- $4913^2 = 289^3 = 24137560$
- И так далее.

Программа полностью:

```
uses
   System, OlympUnit;

// Кордемский, с.43, Задача 3-6

const MIN = 7;
```

```
    MAX = 100000000;

begin
  // заголовок окна:
  Console.Title := 'Кордемский, с.43, Задача 3-6';
  Console.WriteLine('');
  Console.ForegroundColor := ConsoleColor.Red;
  Console.WriteLine('Число 117 649');
  Console.ForegroundColor := ConsoleColor.Green;
  Console.WriteLine();

  // решаем задачу:
  {var f: int64 -> boolean:= num -> num.IsQuadrat and num.IsCube;
  var res:= Range(MIN, MAX)
          .Where(num -> f(num) = true)
          .Println;}
  {var res:= Range(MIN div 7, MAX div 7)
          .Where(num -> (num*7).IsQuadrat and (num*7).IsCube)
          .Select(num -> num*7)
          .Println;}

  var res:= Range(MIN, MAX, 1)
          .Where(num -> num.IsQuadrat and num.IsCube)
          .Println;

  // печатаем ответ:
  Println;
  foreach var num: int64 in res do
  begin
    var s := 'num = ' + num.ToString();
    Console.WriteLine(s);
    s := 'Квадратный корень = ' +
         Round(Power(num, 1.0 / 2.0));
    Console.WriteLine(s);
    s := 'Кубический корень = ' +
         Round(Power(num, 1.0 / 3.0));
    Console.WriteLine(s);
    Console.WriteLine();
  end;

  Console.WriteLine();
  Console.ForegroundColor := ConsoleColor.Red;
end.
```

Пара чисел: 3149 и 3151 (Range.Where)



Задача 3.2 из книги *Удивительный мир чисел* [КА86], страница 43:

> Десятичная запись куба каждого из чисел 3149 и 3151 начинается двумя его первыми цифрами, а оканчивается двумя его последними цифрами:
>
> 3149^3 = 31 226 116 949
> 3151^3 = 31 285 651 951
>
> Есть ещё два последовательных четырёхзначных числа, обладающих таким же свойством.
>
> Какие это числа?

Задача решается полным перебором всех четырёхзначных чисел:

```
Range(1000, 9999)
```

Для каждого четырёхзначного числа мы находим его куб.

> Числа большие, поэтому они должны иметь тип ***int64***!

Затем сравниваем в этих числах 2 первые и 2 последние цифры. Для этого мы привлекаем функции First2, First2Cube, Last2 и Last2Cube, разработанные нами в проекте *И «хвост», и «грива»*:

```
function First2(num: int64): int64;
begin
    while (num >= 100) do
        num := num div 10;
    Result := num;
```

```
end;

var First2Cube: int64-> int64 := num -> First2(num * num * num);

var Last2: integer-> integer := num -> num mod 100;

var Last2Cube: int64-> int64 := num -> num * num * num mod 100;
```

Благодаря этим функциям метод Where получился очень простым:

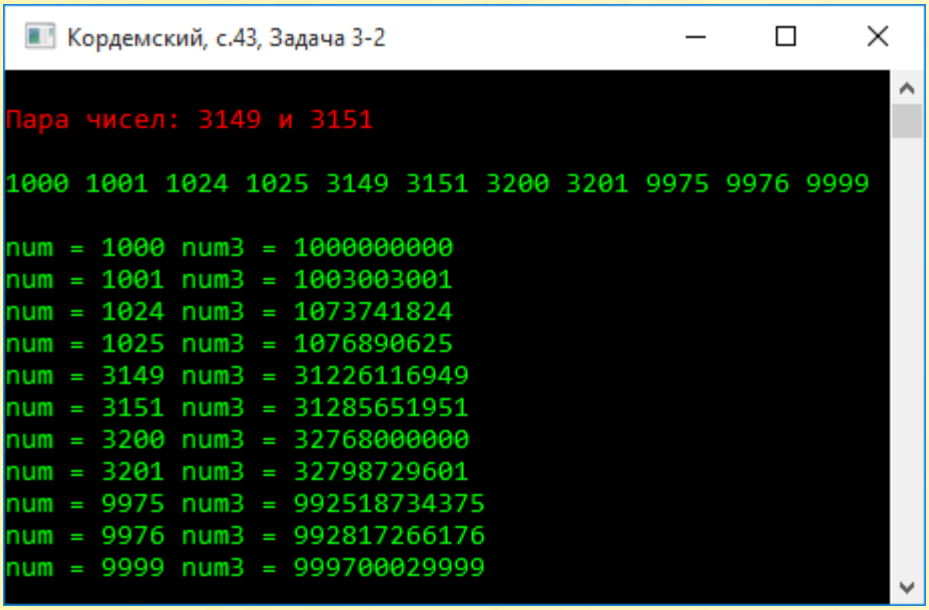
```
Where(n -> ((First2(n) = First2Cube(n)) and
            (Last2(n) = Last2Cube(n))))
```

Метод Println показывает, что задача имеет *несколько решений*. Печатаем их красиво:

```
uses
   System;

// Кордемский, с.43, Задача 3-2


function First2(num: int64): int64;
begin
    while (num >= 100) do
        num := num div 10;
    Result := num;
end;

var First2Cube: int64-> int64 := num -> First2(num * num * num);

var Last2: integer-> integer := num -> num mod 100;

var Last2Cube: int64-> int64 := num -> num * num * num mod 100;


begin
    // заголовок окна:
    Console.Title := 'Кордемский, с.43, Задача 3-2';
    Console.WriteLine();
```

```
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Пара чисел: 3149 и 3151');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    // решаем задачу:
    var nums := Range(1000, 9999)
                .Where(n -> ((First2(n) = First2Cube(n)) and
                             (Last2(n) = Last2Cube(n))))
                .Println;

    // печатаем ответ:
    Println;
    foreach var num: int64 in nums do
    begin
        var s := 'num = ' + num.ToString;
        s += ' num3 = ' + (num * num * num).ToString();
        Println(s);
    end;
    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

На рисунке приведён полный список чисел, выполняющих условие задачи:



В книге указана пара чисел 3200 и 3201, но вы видите, что есть и другие пары чисел:

1000 и 1001
1024 и 1025
9975 и 9976

Трёхзначное число (Range.Where)

Задача 3 из книги *Удивительный мир чисел* [КА86], страница 63:

> Найдите трёхзначное число, обладающее следующими свойствами:
>
> - число десятков на 4 меньше числа единиц, но на 4 больше числа сотен;
> - если цифры этого числа разместить в обратном порядке, то новое полученное число будет на 792 больше искомого.

Чтобы найти трёхзначное число с требуемыми свойствами, необходимо перебрать все трёхзначные числа и выбрать те из них, которые удовлетворяют условиям задачи.

Метод **Range** генерирует все трёхзначные числа:

```
Range(100, 999)
```

Нам нужно найти, сколько **единиц**, **десятков** и **сотен** содержит текущее число. Это нетрудно сделать с помощью двух операций – целочисленного деления и деления с остатком:

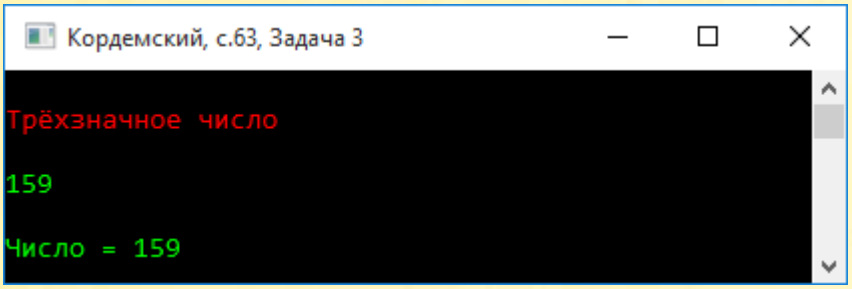
```
// число единиц:
var e: integer-> integer := n -> n mod 10;
// число десятков:
var d: integer-> integer := n -> n div 10 mod 10;
// число сотен:
```

```
var s: integer-> integer := n -> n div 100 mod 10;
```

Выполнить **проверки** и того проще:

```
 Where(n -> d(n) = e(n) - 4)
.Where(n -> d(n) = s(n) + 4)
```

И вот мы получили **ответ** на эту задачу:



Поскольку мы перебрали все трёхзначные числа и нашли только *одно*, которое удовлетворяет условиям задачи, то последняя проверка на разность с перевёрнутым числом оказалась бы лишней!

Вся программа целиком:

```
uses
   System;

// Кордемский, с.63, Задача 3


begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.63, Задача 3';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Трёхзначное число');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу -->
```

```
    // число единиц:
    var e: integer-> integer := n -> n mod 10;
    // число десятков:
    var d: integer-> integer := n -> n div 10 mod 10;
    // число сотен:
    var s: integer-> integer := n -> n div 100 mod 10;

    var res:= Range(100, 999)
              .Where(n -> d(n) = e(n) - 4)
              .Where(n -> d(n) = s(n) + 4)
              .Println;

    // печатаем ответ:
    Println;
    var str := 'Число = ' + res.First;
    Console.WriteLine(str);
    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Таких чисел только два (Range.Where)

Задача 1 из книги *Удивительный мир чисел* [КА86], страница 63:

Есть только два двузначных числа, каждое из которых равно неполному квадрату разности своих цифр.

Найдите эти числа.

Чтобы облегчить решение, подскажем, что одно число на 11 больше другого.

Так как двузначных чисел очень мало, то мы и без подсказки найдём оба искомых числа!

Получаем от метода Range все двузначные числа:

```
Range(10, 99)
```

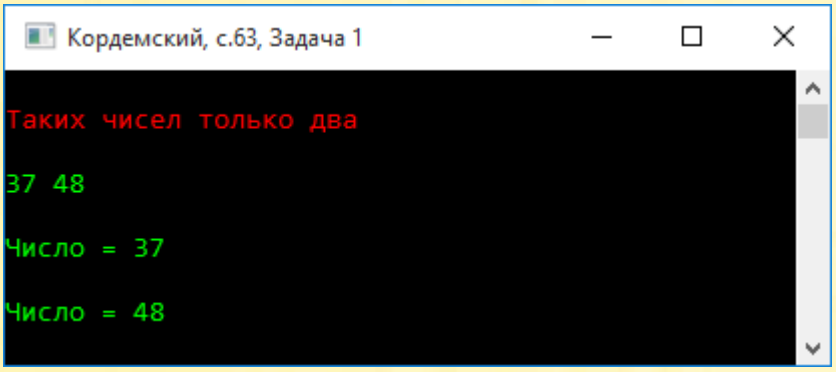
Выделяем из каждого числа **единицы** и **десятки**:

```
// число единиц:
var e: integer-> integer := n -> n mod 10;
// число десятков:
var d: integer-> integer := n -> n div 10 mod 10;
```

И сравниваем само число с неполным квадратом разности его цифр:

```
Where(n -> e(n) * e(n) + d(n) * d(n) - e(n) * d(n)  = n)
```

Задача решена:



Полный текст программы:

```
uses
   System;

// Кордемский, с.63, Задача 1


begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.63, Задача 1';
   Console.WriteLine('');
```

```
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Таких чисел только два');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    // решаем задачу -->
    // число единиц:
    var e: integer-> integer := n -> n mod 10;
    // число десятков:
    var d: integer-> integer := n -> n div 10 mod 10;

    var res:= Range(10, 99)
              .Where(n -> e(n) * e(n) + d(n) * d(n) - e(n) * d(n)  = n)
              .Println;

    // печатаем ответ:
    Println;
    foreach var n in res do
    begin
       var str := 'Число = ' + n;
       Console.WriteLine(str);
       Console.WriteLine();
    end;
    Console.ForegroundColor := ConsoleColor.Red;

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Ещё два числа (Range.Where)

Задача 2 из книги *Удивительный мир чисел* [КА86], страница 63:

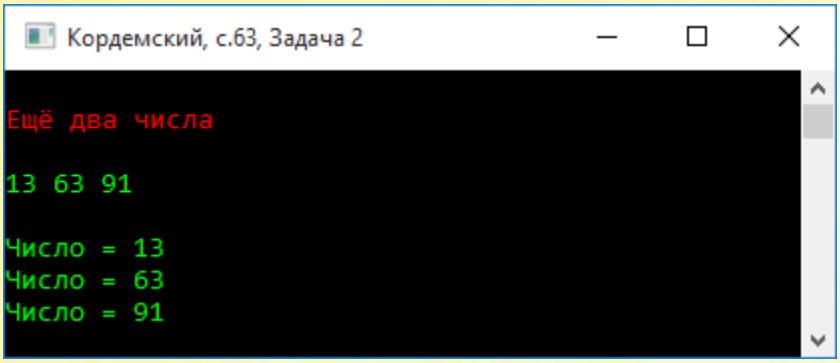
Сходным свойством обладают ещё два двузначных числа: каждое равно неполному квадрату суммы своих цифр.

Найдите эти числа, зная, что одно число на 50 больше другого.

Эта задача решается точно так же, как предыдущая. Только в методе Where нужно исправить один знак:

```
uses
   System;

// Кордемский, с.63, Задача 2


begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.63, Задача 2';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Ещё два числа');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу -->

   // число единиц:
   var e: integer-> integer := n -> n mod 10;
   // число десятков:
   var d: integer-> integer := n -> n div 10 mod 10;

   var res:= Range(10, 99)
            .Where(n -> e(n) * e(n) + d(n) * d(n) + e(n) * d(n)  = n)
            .Println;

   // печатаем ответ:
   Println;
   foreach var n in res do
   begin
      var str := 'Число = ' + n;
      Console.WriteLine(str);
   end;
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Правда, чисел оказалось не два, а три:



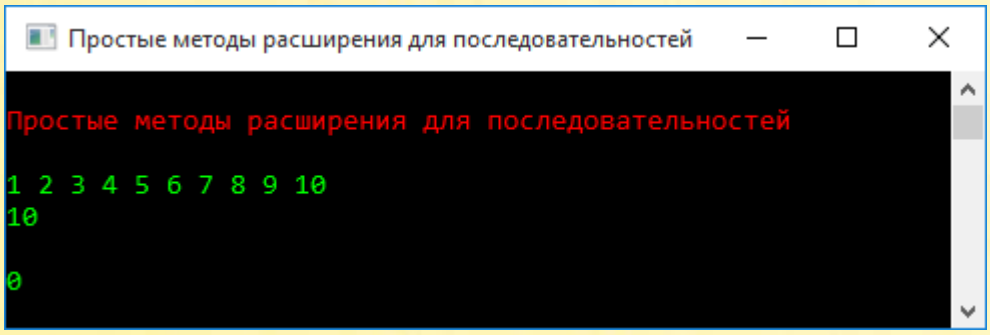
Методы *Count* и *LongCount*

Метод Count без параметров возвращает число элементов в последовательности.

```
function Count(): integer;
```

Если последовательность пустая, то метод вернёт 0:

```
// Count
var sq := Range(1, 10).Println;
Println(sq.Count);

sq := Range(10, 1).Println;
Println(sq.Count);
Println;
```

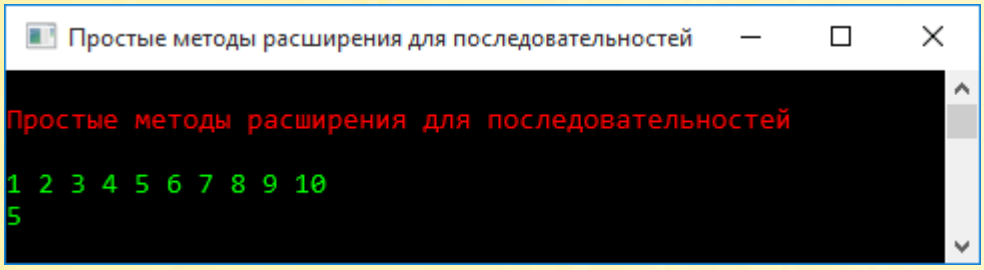


Во второй последовательности нет ни одного элемента!

Метод Count с параметром возвращает число элементов в последовательности, которые удовлетворяют заданному условию *predicate*:

```
function Count(predicate: T-> boolean): integer;
```

```
Println(sq.Count(n -> n < 6));
```



Методы LongCount действуют точно так же, но возвращают значение типа *int64*:

```
function LongCount(): int64;
function LongCount(predicate: T->boolean): int64;
```

Любопытное свойство чисел (Range.Count)

Задача 15 (16 – номер задачи в издании 1996 года) из книги *Удивительный мир чисел* [КА86], страница 102:

> Возьмите какое-либо б-значное число, делящееся на 7, например 325 836. Перенесите последнюю цифру в начало записи числа. Образуется новое число 632 583. Оно также делится на 7.

Докажите самостоятельно, что таким свойством обладает любое 6-значное число, делящееся на 7.

Рекомендация. Представьте заданное число так: 7k = = 10a+ 6 (1). Тогда новое число примет вид 100 0006 +a (2). Используя (1), докажите делимость (2) на число 7.

Доказать - значит, убедиться, что **все** 6-значные числа обладают указанным свойством. Поскольку 6-значных чисел совсем немного, то метод полного перебора здесь вполне уместен.

Поскольку диапазон значений проверяемых чисел точно очерчен, то мы генерируем их в методе **Range**:

```
Range(999999, 100000)
```

Все числа, кратные 7, мы пропускаем через первый метод **Where**:

```
Where(n -> n mod 7 = 0)
```

Этим числам назначаем **проверки**.

Чтобы перенести последнюю цифру числа в начало, нужно её **выделить**. Это легко сделать, применив к числу операцию деления по модулю:

```
// последняя цифра числа:
var endn: integer-> integer := n -> n mod 10;
```

Число из первых 5 цифр найти ещё проще – нужно просто разделить исходное число на 10:

```
// 5-значное число из первых 5 цифр числа num:
var start5: integer-> integer := n -> n div 10;
```

При формировании нового 6-значного числа мы переносим последнюю цифру в начало, что соответствует умножению на 100000. К этому произведению следует добавить 5-значное число из первых пяти цифр исходного числа:

```
// новое 6-значное число:
var newnum: integer-> integer := n -> endn(n) * 100000 + start5(n);
```

Эту операцию можно записать более наглядно:

```
123456 ←исходное 6-значное число
612345 ←новое 6-значное число
```

Проверяем условие задачи во втором методе Where:

```
Where(n -> newnum(n) mod 7 <> 0)
```

И последний метод – Count – возвращает число элементов в отфильтрованной последовательности.

Если оно равно 0, значит, не нашлось ни одного числа, опровергающего утверждение в условии задачи.

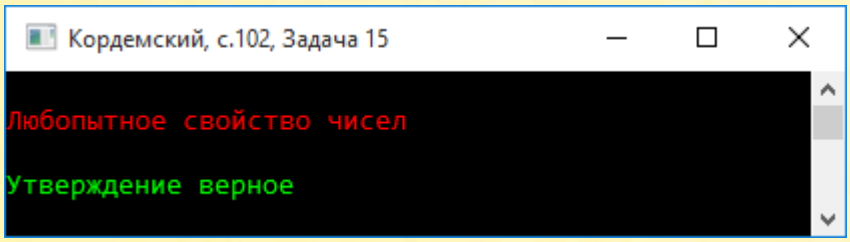
Печатаем на экране результаты проверки:

```
// все числа проверены:
if (res = 0) then
   Console.WriteLine('Утверждение верное')
else
   Console.WriteLine('Утверждение неверное');
```

Вся программа целиком:

```
uses
   System;
```

```
// Кордемский, с.102, Задача 15


begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.102, Задача 15';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Любопытное свойство чисел');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // последняя цифра числа:
   var endn: integer-> integer := n -> n mod 10;

   // 5-значное число из первых 5 цифр числа num:
   var start5: integer-> integer := n -> n div 10;

   // новое 6-значное число:
   var newnum: integer-> integer := n -> endn(n) * 100000 + start5(n);

   var res := Range(999999, 100000)
              .Where(n -> n mod 7 = 0)
              .Where(n -> newnum(n) mod 7 <> 0)
              .Count;

   // все числа проверены:
   if (res = 0) then
      Console.WriteLine('Утверждение верное')
   else
      Console.WriteLine('Утверждение неверное');

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```



Так как мы проверили все 6-значные числа, то можем уверенно утверждать, что любое 6-значное число удовлетворяет условиям задачи.

Как определил ошибку Чохбилмиш? (Range.Count)

Задача 23 (25) из книги *Удивительный мир чисел* [КА86], страница 57:

> Чохбилмиш предложил каждому из двух учеников задумать какое-либо шестизначное число и переставить первую цифру в конец записи числа. Одному сказал: «Найди сумму получившихся чисел». Другому сказал: «Найди разность».
>
> Ученики выполнили действия и написали:
>
> **913 485** и **167 860**.
>
> Чохбилмиш не знал, какие числа были задуманы учениками, но сразу определил: «Вы оба ошиблись».
>
> **Как рассуждал Чохбилмиш?**

В ответе на задачу утверждается, что сумма любого 6-значного числа с другим 6-значным числом, полученным из исходного переносом первой цифры в конец числа, кратна 11 и аналогично полученная разность – кратна 9.

Мы можем проверить эти утверждения с помощью **полного перебора**, тем более что самая трудная часть задачи – перенос цифры – нами уже решена в предыдущем проекте. Добавим ещё 2 функции для удобного вычисления суммы и разности двух чисел:

```
// сумма:
var sum: integer-> integer := n -> n + newnum(n);
// разность:
var razn: integer-> integer := n -> n - newnum(n);
```

Проверка в методе *Where* :

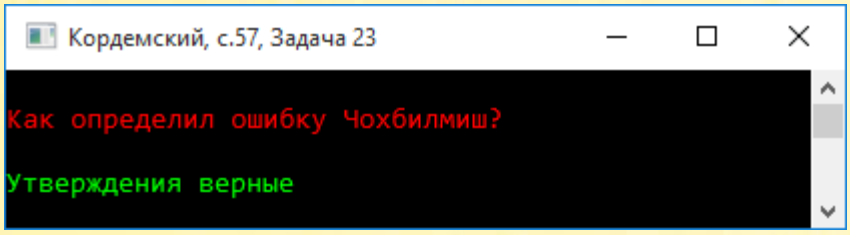
```
Where(n -> (sum(n) mod 11 <> 0) or (razn(n) mod 9 <> 0))
```

Вот и всё решение задачи:

```
uses
   System;

// Кордемский, с.57, Задача 23


begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.57, Задача 23';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Как определил ошибку Чохбилмиш?');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // последняя цифра числа:
   var endn: integer-> integer := n -> n mod 10;

   // 5-значное число из первых 5 цифр числа num:
   var start5: integer-> integer := n -> n div 10;

   // новое 6-значное число:
   var newnum: integer-> integer := n -> endn(n) * 100000 + start5(n);
   // сумма:
   var sum: integer-> integer := n -> n + newnum(n);
   // разность:
   var razn: integer-> integer := n -> n - newnum(n);

   var res := Range(100000, 999999)
              .Where(n -> (sum(n) mod 11 <> 0) or (razn(n) mod 9 <> 0))
              .Count;

   // все числа проверены:
   if (res = 0) then
      Console.WriteLine('Утверждения верные')
   else
      Console.WriteLine('Утверждения неверные');
```

```
    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Проверив все 6-значные числа, мы можем с уверенностью констатировать факт: оба утверждения верные:

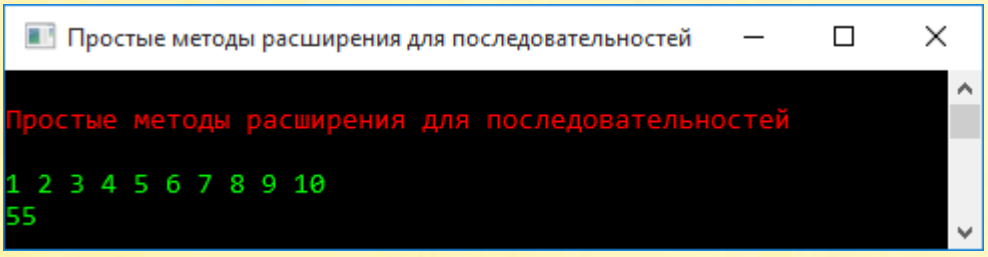


Метод *Sum*

Метод Sum без параметров возвращает *сумму* элементов последовательности:

```
function Sum(): decimal;
```

```
var sq := Range(1, 10).Println;
Println(sq.Sum());
```



Метод Sum с параметром возвращает сумму элементов последовательности, к каждому из которых применена функция *selector*:

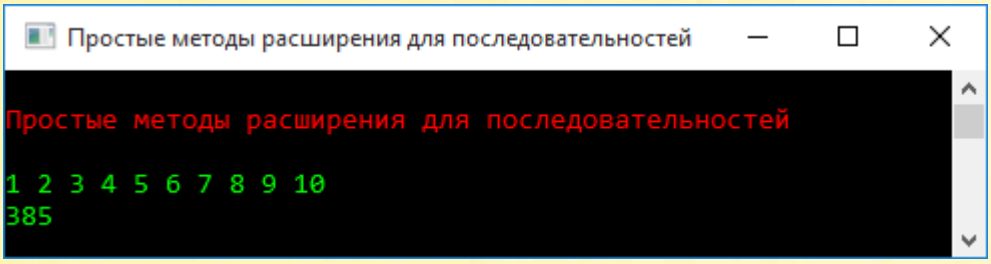
```
function Sum(selector: T -> decimal)): decimal;
```

Например, мы хотим найти *сумму квадратов* элементов. Тогда **функцию** для вычисления квадратов

```
n -> n*n
```

нужно передать методу *Sum*:

```
var sq := Range(1, 10).Println;
Println(sq.Sum(n -> n*n));
```



С помощью метода **Sum** мы легко решим историческую задачу.

Гаусс (Range.Sum)

Задача 5 из книги *Математическая шкатулка* **[Нагибин88]**, страница 15:

Рассказывают, что в начальной школе, где учился мальчик Карл Гаусс, ставший потом знаменитым математиком, учитель, чтобы занять класс на продолжительное время самостоятельной работой, дал детям такое задание - вычислить сумму всех натуральных чисел от 1 до 100. Но маленький Гаусс это задание моментально выполнил.

Попробуй и ты быстро выполнить это задание.

Нетрудно в этом ряде чисел увидеть **арифметическую прогрессию**, начинающуюся с единицы и насчитывающую 100 членов. Разность арифметической прогрессии равна 1. Зная все параметры этого ряда, мы быстро найдём его сумму:

> Сумма = (1 + 100)*100/2 = 5050

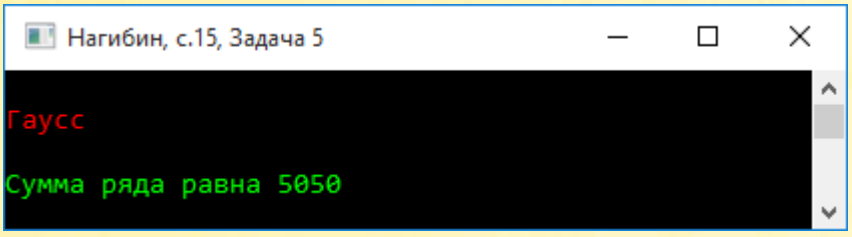
Учитель, конечно, предполагал, что школяры будут вычислять сумму ряда последовательным прибавлением очередного члена ряда к текущей сумме.

Давайте решим эту простую задачу на последовательности.

Метод **Range** генерирует заданную арифметическую прогрессию, а метод **Sum** находит сумму всех элементов последовательности:

```
uses
   System;

// Нагибин, с.15, Задача 5


begin
   // заголовок окна:
   Console.Title := 'Нагибин, с.15, Задача 5';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Гаусс');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var summa := Range(1,100).Sum;
   Console.WriteLine('Сумма ряда равна {0}', summa);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Всё решение уложилось в одну короткую строчку:



Подставляя нужные значения в метод **Range**, вы быстро найдёте сумму любой арифметической прогрессии. Например, давайте решим такую задачу.

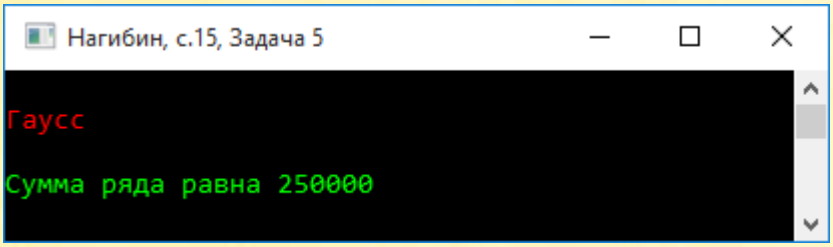
Задача 7.1 из книги *Математическая шкатулка* **[Нагибин88]**, страница 15:

> Как быстро вычислить сумму 1 + 3+ 5+7 + 9 + … + 997 +999?

Добавьте в **главный блок** новую строку:

```
// решаем задачу:
//var summa := Range(1,100).Sum;
var summa := Range(1, 999, 2).Sum;
```

И получите **ответ** на задачу:



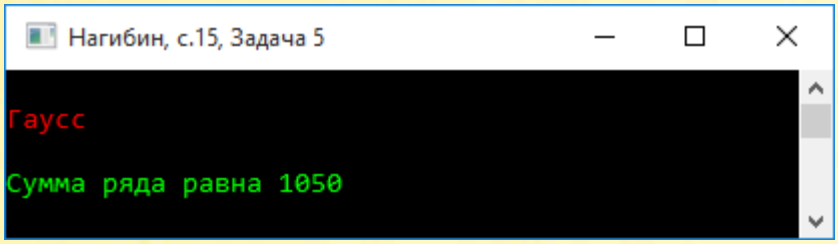
Задача 838 из книги *Математическая шкатулка* **[Нагибин88]**, страница 128:

> Вычислите: 5 + 10 + 15+20 + 25 + … + 100.

И опять одна строка программы решает задачу:

```
// решаем задачу:
```

```
//var summa := Range(1,100).Sum;
//var summa := Range(1, 999, 2).Sum;
var summa := Range(5, 100, 5).Sum;
```



«Избранные» числа (Range.Sum)

Задача 12 из книги *Удивительный мир чисел* [КА86], страница 65:

> Есть 80 четырёхзначных и 800 пятизначных чисел, не оканчивающихся нулем, и таких, что если от любого из них вычтем 999 в случае четырёхзначного числа и 9999 в случае пятизначного числа, то всякий раз получим **обращённое** число, т. е. записанное теми же цифрами, но в обратном порядке. Какие это числа?
>
> **Найдите способ быстрого вычисления суммы этих чисел** (без привлечения компьютера).

Мы проигнорируем замечание в скобках и решим сначала задачу для **4-значных** чисел. Так как условие задачи запрещает числам оканчиваться нулём, то минимальное 4-значное число равно **1001**, а максимальное – **9999**. Обозначим их **константами**:

```
const MIN = 1001;
      MAX = 9999;
      MINUS = 999;
```

Генерируем все 4-значные числа в методе **Range**:

```
// решаем задачу:
var res:= Range(MIN, MAX)
```

Отфильтровываем числа, заканчивающиеся нулём:

```
Where(num -> num mod 10 <> 0)
```

Вычитаем из каждого числа 999 и сравниваем его с обращённым исходным числом:

```
Where(num -> num - MINUS = num.ReverseNum)
```

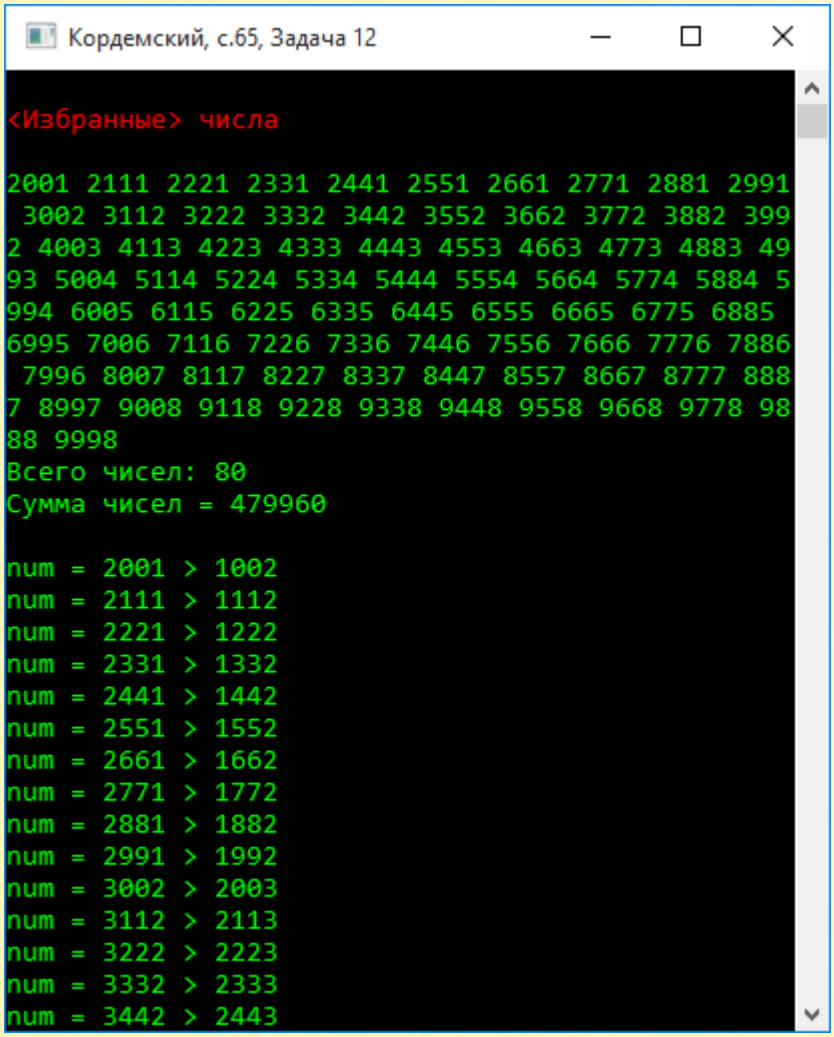
Здесь мы используем метод расширения ReverseNum из модуля OlympUnit.

Если они совпадают, то найдено ещё одно решение задачи. Печатаем его:

```
.Println;
```

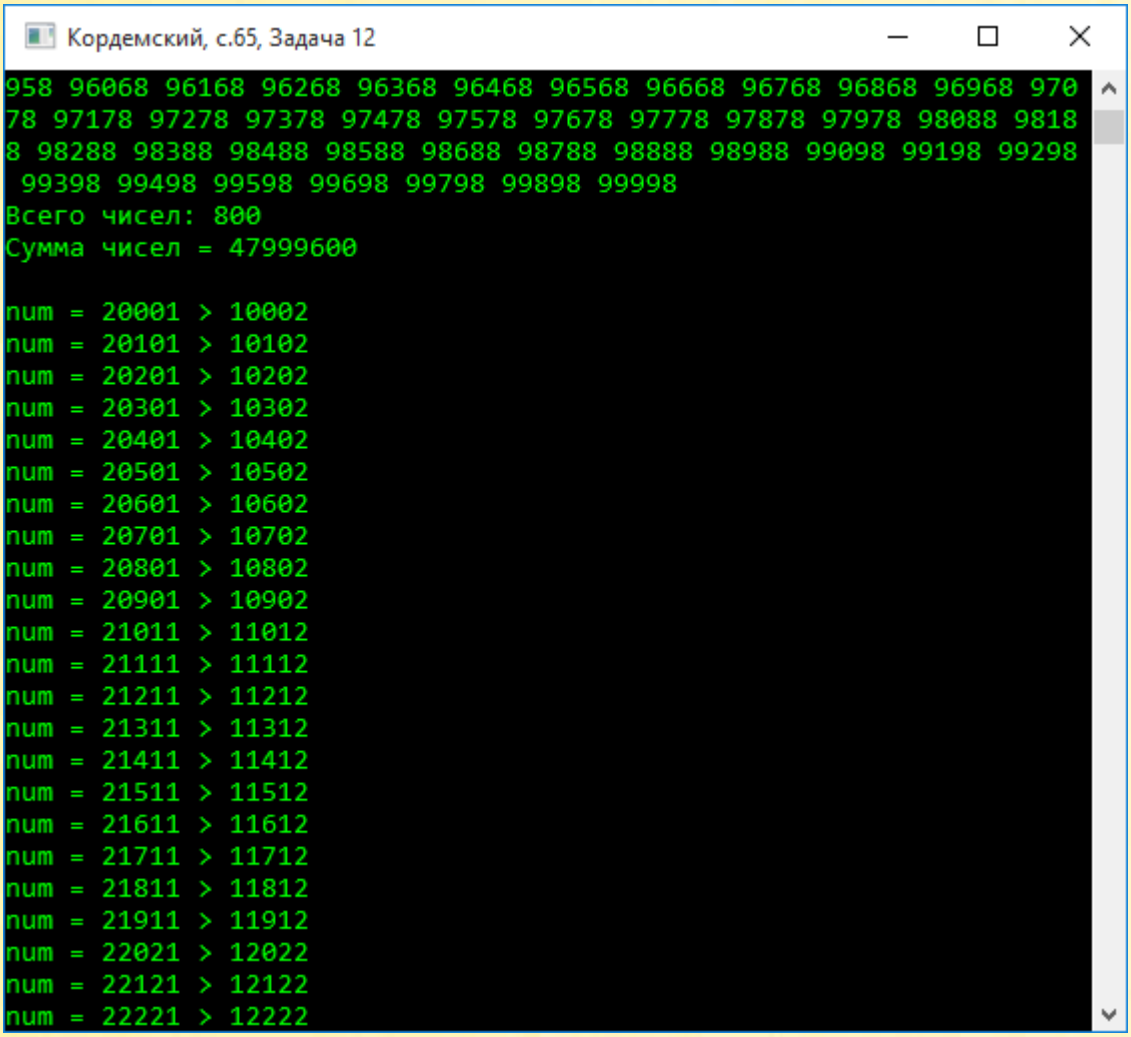
Как и указано в книге *Удивительный мир чисел*, всего существует 80 таких чисел.

Для поиска 5-значных чисел необходимо изменить значения констант:



```
const MIN = 10001;
      MAX = 99999;
      MINUS = 9999;
```

5-значных чисел ровно в 10 раз больше, чем 4-значных:



Поскольку мы решаем задачу на компьютере, то было бы странно, если бы мы принялись сумму найденных чисел вычислять вручную! Метод Sum быстро находит сумму:

```
Console.WriteLine('Сумма чисел = ' + res.Sum);
```

Вся программа целиком:

```
uses
   System, OlympUnit;

// Кордемский, с.65, Задача 12

{
const MIN = 1001;
      MAX = 9999;
      MINUS = 999;
}

const MIN = 10001;
      MAX = 99999;
      MINUS = 9999;

begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.65, Задача 12';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('«Избранные» числа');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

    // решаем задачу:
    var res:= Range(MIN, MAX)
              .Where(num -> num mod 10 <> 0)
              .Where(num -> num - MINUS = num.ReverseNum)
              .Println;

    Console.WriteLine('Всего чисел: ' + res.Count);
    Console.WriteLine('Сумма чисел = ' + res.Sum);

    // печатаем ответ:
    Console.WriteLine();
    foreach var num in res do
    begin
        var s := 'num = ' + num.ToString();
        s += ' > ' + num.ReverseNum.ToString();
        Console.WriteLine(s);
    end;
```

```
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

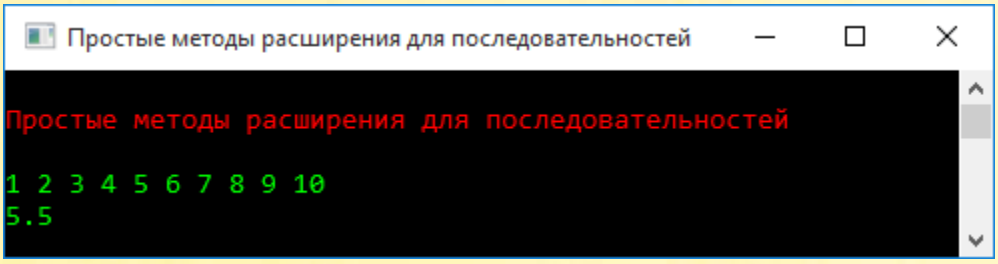
Метод *Average*

Метод Average без параметров возвращает *среднее арифметическое* элементов числовой последовательности:

```
function Average(): decimal;
```

Найдём среднее арифметическое последовательности первых десяти натуральных чисел:

```
var sq := Range(1, 10).Println;
Println(sq.Average());
```

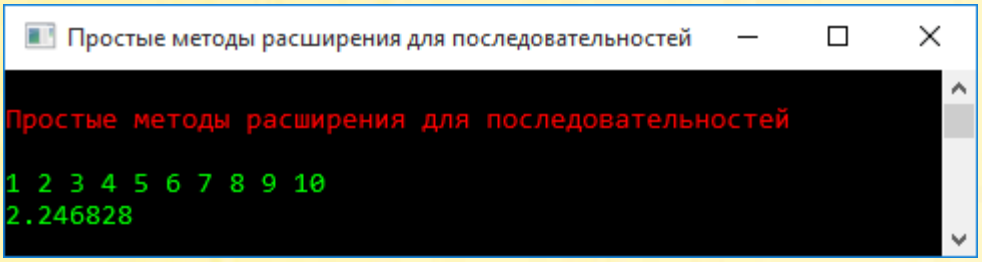


Метод Average с параметром возвращает среднее арифметическое элементов числовой последовательности, к каждому из которых применена функция *selector*:

```
function Average(selector: T -> decimal)): decimal;
```

Найдём среднее арифметическое квадратных корней из чисел, составляющих последовательность *sq*:

```
var sq := Range(1, 10).Println;
Println(sq.Average(n -> Sqrt(n)));
```



Методы *Min* и *Max*

Метод Min возвращает *минимальный* элемент последовательности чисел:

```
function Min(): число;
```

Второй метод Min возвращает минимальное значение функции преобразования *selector*, применённой к каждому элементу последовательности чисел:

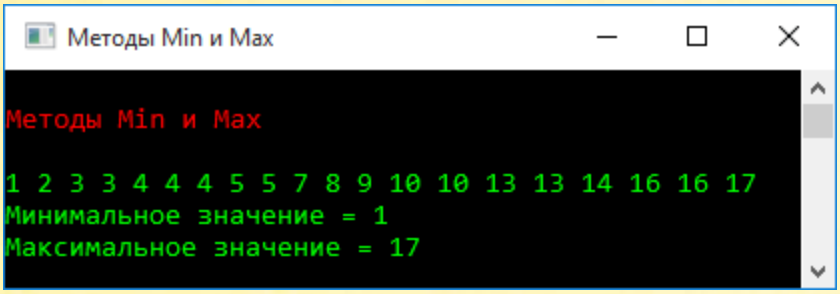
```
function Min(selector: T->число): число;
```

Методы Max возвращают *максимальный* элемент последовательности чисел:

```
function Max(): число;

function Max(selector: T->число): число;
```

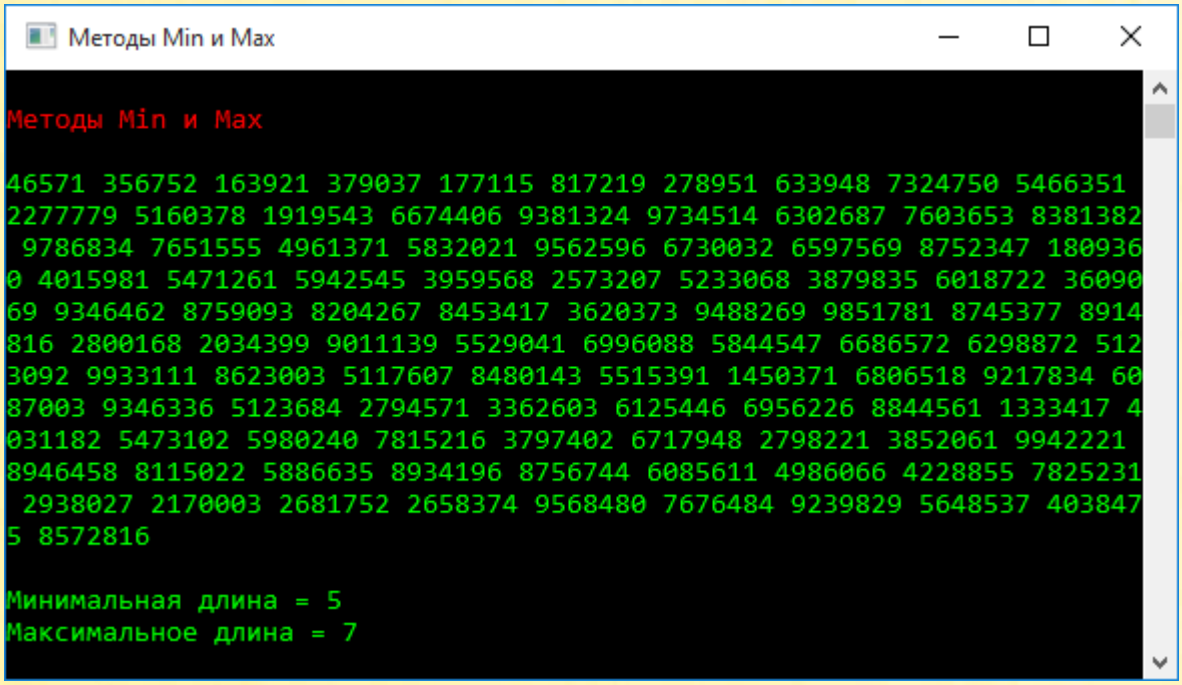
Найдём наибольший и наименьший элемент в последовательности случайных чисел:

```
var sq:= SeqRandom(20,1, 20).ToArray.OrderBy(n -> n).Println;
Println('Минимальное значение = ' + sq.Min(sq));
Println('Максимальное значение = ' + sq.Max(sq));
```



Значение функции преобразования selector определяет минимальный или максимальный «элемент» последовательности. Например, мы можем найти длину самого короткого и самого длинного числа в последовательности *sq*:

```
var sq:= SeqRandom(100,1, 10000000)
        .ToArray
        .OrderBy(n -> n.ToString.Length)
        .Println;
Println;
Println('Минимальная длина = ' + sq.Min(n -> n.ToString.Length));
Println('Максимальное длина = ' + sq.Max(n -> n.ToString.Length));
```

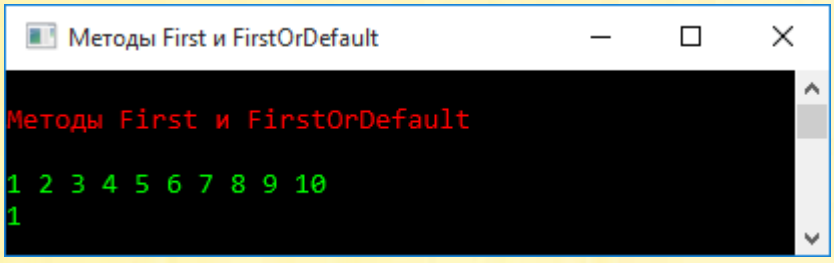


Методы *First* и *FirstOrDefault*

Метод First без параметров возвращает *первый* элемент последовательности:

```
function First(): T;
```

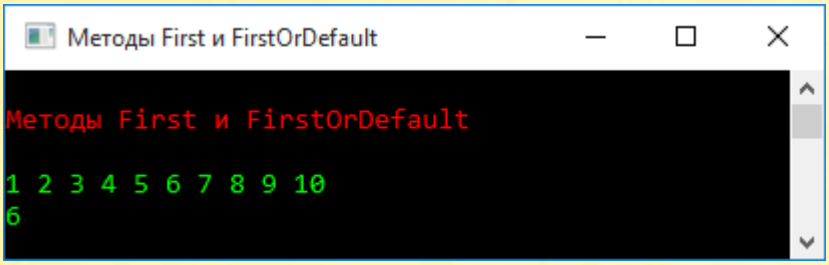
```
var sq := Range(1,10).Println;
Println(sq.First);
```



Метод First с параметром возвращает первый элемент последовательности, удовлетворяющий условию *predicate*:

```
function First(predicate: T->boolean): T;
```

```
var sq := Range(1,10).Println;
Println(sq.First(n -> n > 5));
```



В этом примере метод First возвращает первый элемент последовательности, который больше 5, то есть 6.

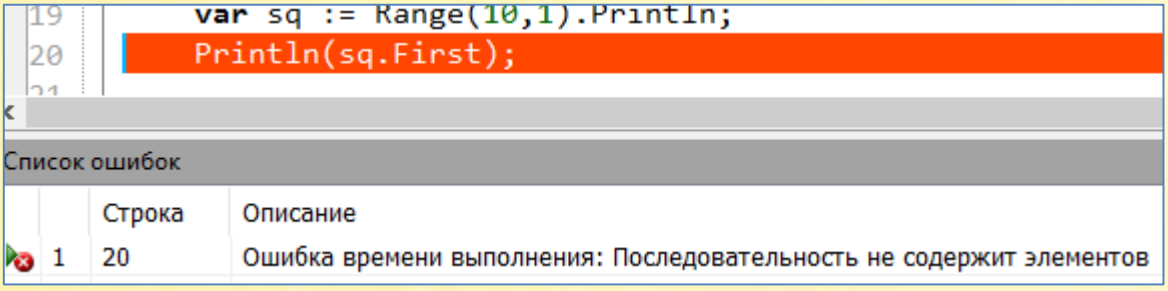
Методы FirstOrDefault действуют точно так же, но, если последовательность *пустая* (не содержит элементов), то они возвращают значение по умолчанию. Для числовых типов это 0:

```
function FirstOrDefault(): T;
function FirstOrDefault(predicate: T->boolean): T;
```

Создаём пустую последовательность, берём методом First её первый элемент и печатаем на экране:

```
var sq := Range(10,1).Println;
Println(sq.First);
```

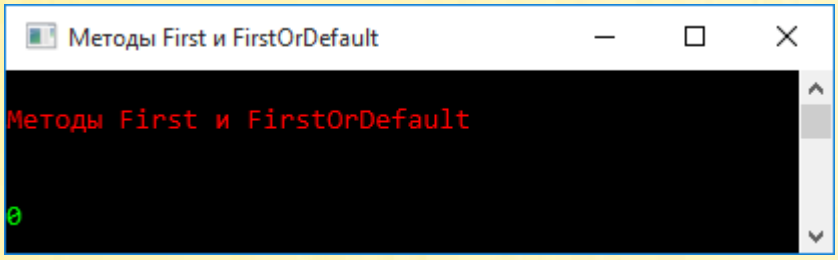
Так как у пустой последовательности нет первого элемента, то компилятор выдаёт сообщение об ошибке:



На этом выполнение программы завершается.

Естественно, такое поведение программы нельзя считать нормальным. Если у вас есть опасения, что последовательность может оказаться пустой, всегда применяйте методы FirstOrDefault. В этом случае ошибка в программе не возникнет, а метод FirstOrDefault вернёт нуль:

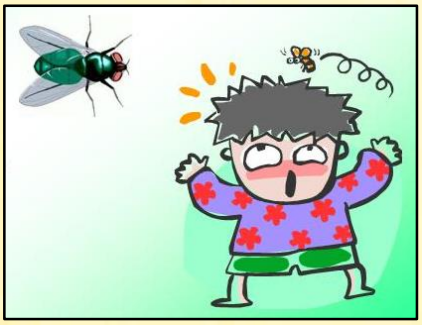
```
var sq := Range(10,1).Println;
Println(sq.FirstOrDefault);
```



Но если последовательность не пустая, и первый элемент равен нулю, то метод **FirstOrDefault** также вернёт нуль. Это может ввести программу в заблуждение. Используйте при необходимости метод **Count**, чтобы распознать пустую последовательность.

Многие задачи имеют **единственное** решение. Чтобы извлечь его из последовательности, применяют метод **First** или **ElementAt**(0), что то же самое.

Назойливый остаток (Range.First)



В книге Бориса Кордемского и Аскера Ахадова *Удивительный мир чисел* [КА86] вы найдёте немало интересных задач, в том числе и на делимость. На странице 86 авторы предлагают решить задачу **Назойливый остаток**:

> Некоторые числа, кратные числу 7, при делении на 2, на 3, на 4, на 5 и на 6 дают остаток 1.
>
> **Найдите наименьшее из таких чисел.**

Эта же задача напечатана в книге Фёдора Нагибина и Евгения Канина *Математическая шкатулка* [Нагибин88], задача 42, страницы 18-19, но в более занимательной форме:

> Колхозница привезла на рынок для продажи корзину яиц. Продавала она их по одной и той же цене. После продажи яиц колхозница пожелала проверить,

верно ли она получала деньги. Но вот беда: она забыла, сколько у неё было яиц. Вспомнила она только, что когда перекладывала яйца по 2, то оставалось одно яйцо; одно яйцо оставалось также при перекладывании яиц по 3, по 4, по 5, по 6. Когда же она перекладывала яйца по 7, то не оставалось ни одного.

Помоги колхознице сообразить, сколько у неё было яиц.

Поскольку речь идёт о **натуральных** числах, то мы можем начать наши поиски с *единицы*. Метод **Step** генерирует бесконечную последовательность натуральных чисел:

```
1.Step()
```

Метод **Where** пропускает только те из них, которые удовлетворяют условиям задачи:

```
Where(n -> (n mod 7 = 0) and
           (n mod 2 = 1) and
           (n mod 3 = 1) and
           (n mod 4 = 1) and
           (n mod 5 = 1) and
           (n mod 6 = 1))
```

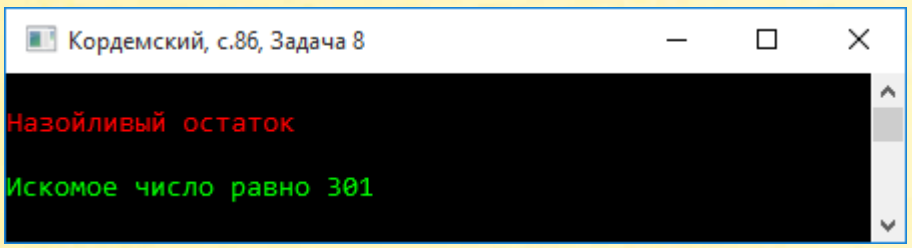
И наконец, метод **First** останавливает решение задачи, когда будет найдено первое (наименьшее) число, удовлетворяющее условиям задачи.

Весь код программы:

```
uses
   System;

// Кордемский, с.86, Задача 8


begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.86, Задача 8';
```

```
    Console.WriteLine('');
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Назойливый остаток');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    // решаем задачу:
    var res:= 1.Step()
              .Where(n -> (n mod 7 = 0) and
                          (n mod 2 = 1) and
                          (n mod 3 = 1) and
                          (n mod 4 = 1) and
                          (n mod 5 = 1) and
                          (n mod 6 = 1))
              .First;

    // печатаем ответ:
    Console.WriteLine('Искомое число равно ' + res);
    Println(NewLine);
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

На рисунке вы видите ответ на эту задачу:



А *есть ли ещё и другие числа, которые имеют назойливый остаток*?

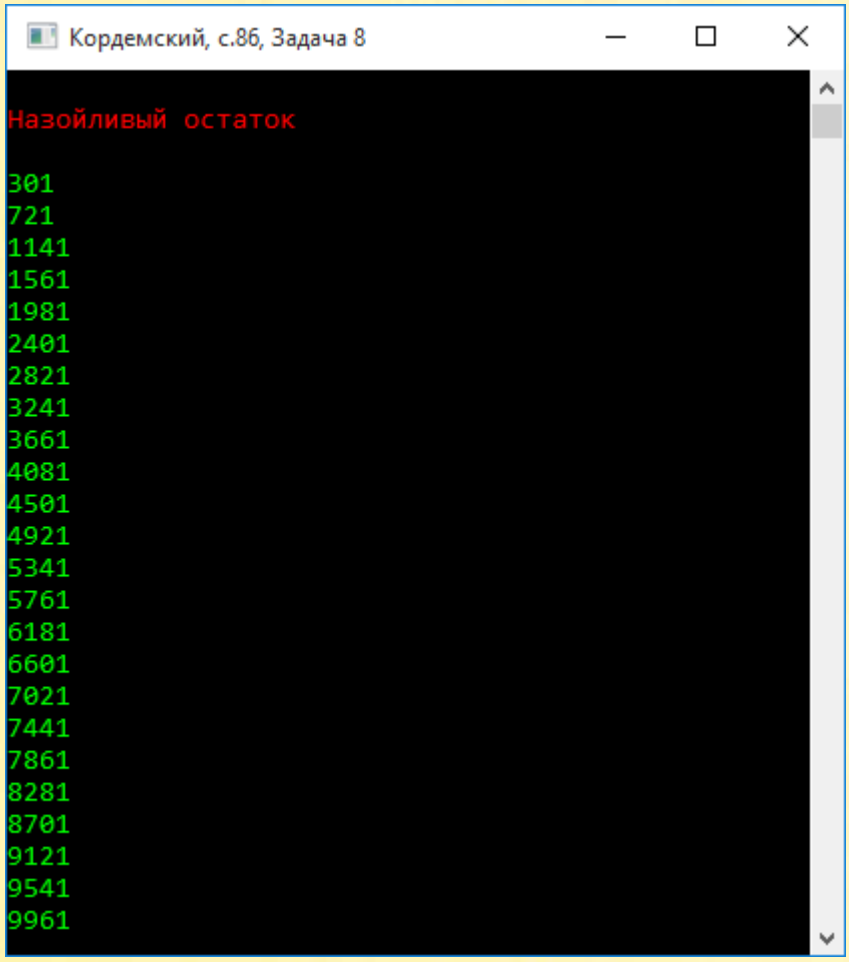
Тут, конечно, следует учесть, что бесконечный цикл уже не годится, потому что он действительно станет бесконечным, если искомых чисел очень много (а это нетрудно предвидеть). Поэтому мы объявляем константу с верхней границей поиска:

```
//макс. число:
const MAX = 10000;
```

Оператор для поиска чисел изменился несущественно:

```
Range(1, MAX).Where(n -> (n mod 7 = 0) and
                         (n mod 2 = 1) and
                         (n mod 3 = 1) and
                         (n mod 4 = 1) and
                         (n mod 5 = 1) and
                         (n mod 6 = 1))
             .Println(NewLine);
```

Запускаем приложение и видим, что среди первой десятки тысяч натуральных чисел довольно много подходящих под условие задачи:



Также нетрудно подметить такую **закономерность**. Если обозначить через **n** номер искомого числа, то все числа с назойливыми остатками можно легко найти по формуле:

$$num = 301 + 420(n-1)$$

Во Франции бытует подобная задач (*Математический фольклор*, *Задача Д17*):

Женщина несла на базар две корзины яиц. Прохожий случайно толкнул её, корзинки упали и яйца разбились. Виновник извинился, предложил возместить ущерб и спросил:

- Сколько яиц было в корзинах?
- Не помню точно, - ответила женщина, - но знаю, когда вынимала их по 2, по 3, по 4, по 5 или по 6 яиц, то в корзине оставалось по 1 яйцу, а когда вынимала по 7, то не оставалось ни одного.

Сколько яиц было в корзинах?

Популярна яичная тема и в Болгарии (*Математический фольклор, Задача 87*):

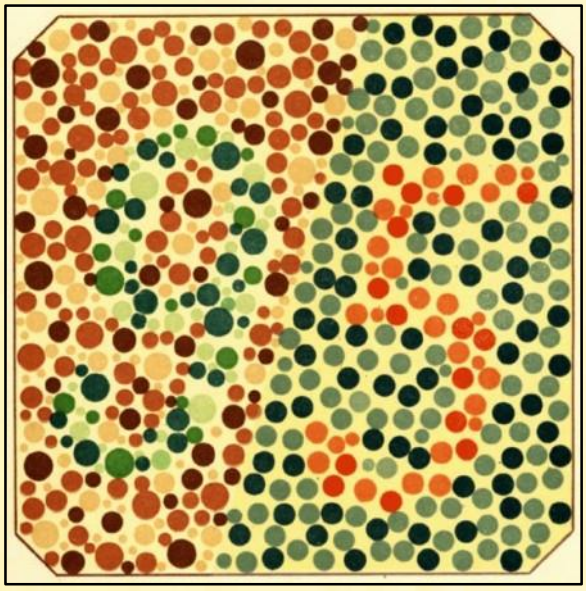
Женщина несла на базар две корзины яиц. Её нечаянно толкнул парень, корзины упали, а яйца разбились. Парень, чтобы заплатить, спросил, сколько было всего яиц.

- Я их не считала, но когда складывала в корзины по 2, по 3, по 4, по 5, по 6, то всякий раз оставалось по 1 яйцу, а когда складывала по 7 - не оставалось ни одного.

Сколько яиц было в корзинах?

И такие есть числа (Range.First)

Задача 4-1 из книги *Удивительный мир чисел* [КА86], страница 63:

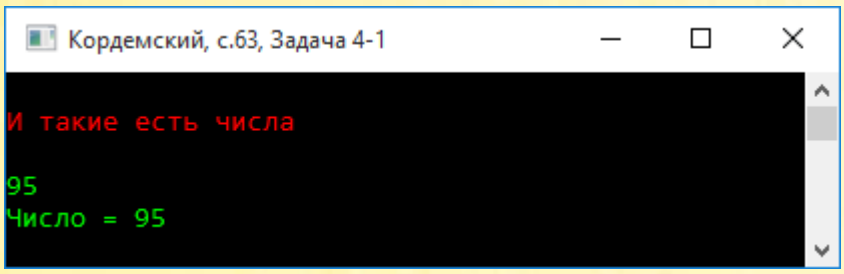


> Какое двузначное число в 19 раз больше числа его единиц?

Чтобы решить задачу, достаточно перебрать все двузначные числа и проверить условие задачи:

```
uses
   System;

// Кордемский, с.63, Задача 4-1


begin
   //заголовок окна:
   Console.Title := 'Кордемский, с.63, Задача 4-1';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('И такие есть числа');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   var res:= Range(10,99).Where(n -> n = n mod 10*19).Println.First;

   var s := 'Число = ' + res.ToString;
   Console.WriteLine(s);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Как вы видите на рисунке, задача имеет *единственное* решение: искомое число равно 95:



Шестизначное число (Range.First)

Задача 4 из книги *Удивительный мир чисел* [КА86], страница 44:

> Ученику понадобилось написать наибольшее из шестизначных чисел, кратных 11 и чтобы цифра 6 была первой слева. Как надо действовать ученику для быстрого выполнения задания, если признаков делимости на 11 он ещё не знает?
>
> Сообщим, что искомое число обладает забавной особенностью: если каждую его цифру повернуть на 180° в плоскости бумаги, оставляя её на прежнем месте, то образовавшееся число окажется дважды кратным 11 (делится на 11 и частное также делится на 11).
>
> Выявите ещё одну особенность чисел: найденного и с повёрнутыми цифрами.

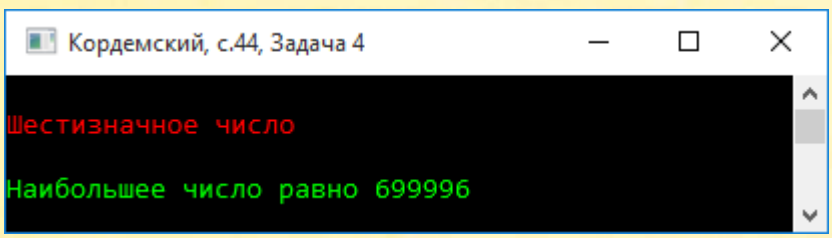
Легко догадаться, что нужно искать решение задачи среди шестизначных чисел, начинающихся с **шестёрки**. Причём перебирать числа следует задом наперёд, то есть от наибольшего к наименьшему, чтобы сразу же найти наибольшее из всех возможных чисел в заданном диапазоне. В методе Where мы отбираем числа, удовлетворяющие условиям задачи, а метод First присваивает первое прошедшее отбор число переменной *res*:

```
Where(n -> n mod 11 = 0)
```

Вся программа полностью:

```
uses
   System;

// Кордемский, с.44, Задача 4


begin
   // заголовок окна:
   Console.Title := 'Кордемский, с.44, Задача 4';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Шестизначное число');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var res:= Range(699999, 600000, -1)
            .Where(n -> n mod 11 = 0)
            .First;

   var s := 'Наибольшее число равно ' + res.ToString;
   Console.WriteLine(s);
   Console.WriteLine();


   if (966669 mod 121 = 0) then
      Console.WriteLine('966669 : 121 = ' + 966669 div 121)
   else
      Console.WriteLine('Число 966669 на 121 не делится!');

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

А вот и ответ:



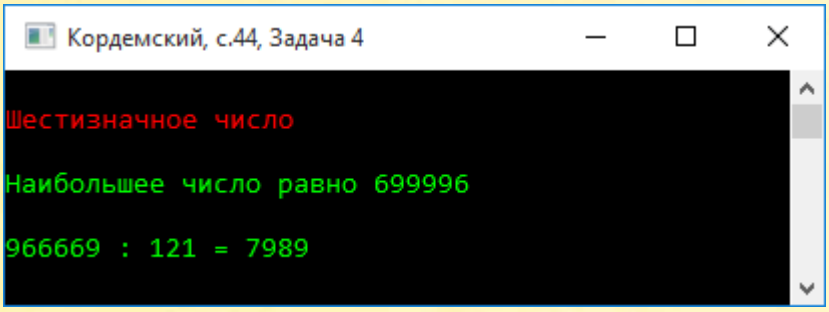
По условию задачи, **переворачиваем** все цифры найденного числа:

699996 → 966669

Проверяем, делится ли перевёрнутое число на 11 * 11 = 121:

```
if (966669 mod 121 = 0) then
    Console.WriteLine('966669 : 121 = ' + 966669 div 121)
else
    Console.WriteLine('Число 966669 на 121 не делится!');
```

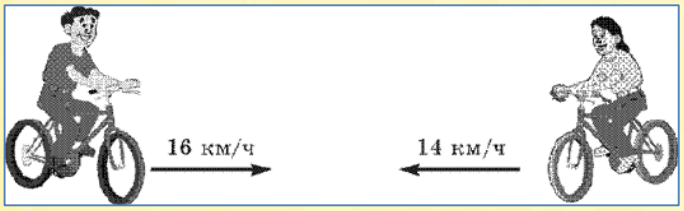
Рисунок подтверждает: **перевёрнутое число делится на 121**:



А ещё одной особенностью этих чисел является их **палиндромичность** – они одинаково читаются и в прямом, и в обратном направлении.

Три велосипедиста (Step.First)

Задача 16 из книги *Удивительный мир чисел* [КА86], страница 55:



Три велосипедиста начали с общего старта движение по круговой дорожке. Первый делает полный круг за 21 мин, второй - за 35 мин, а третий - за 15 мин.

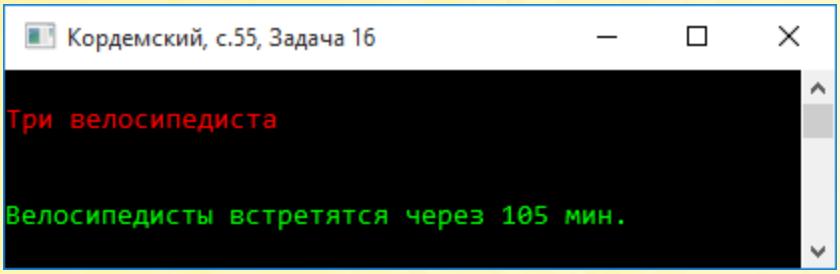
Через сколько минут они ещё раз окажутся вместе в начальном пункте?

Задача решается элементарно, если догадаться, что велосипедисты окажутся в точке старта через **целое** число кругов. А это случится, когда пройдёт время ***t*** (в минутах), которое нацело делится на 21, 35 и 15.

Запустим часы и будем ждать, пока не пройдёт столько времени (в минутах), чтобы оно делилось без остатка на означенные числа:

```
uses
   System;

// Кордемский, с.55, Задача 16


begin
    // заголовок окна:
    Console.Title := 'Кордемский, с.55, Задача 16';
    Console.WriteLine('');
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Три велосипедиста');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    // решаем задачу:
    var t := 1.Step()
              .Where(n -> (n mod 21 = 0) and
                          (n mod 35 = 0) and
                          (n mod 15 = 0))
              .First;
    // печатаем ответ:
    Println;
    Console.WriteLine('Велосипедисты встретятся через {0} мин.', t);


    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Встреча произойдёт через 105 минут:

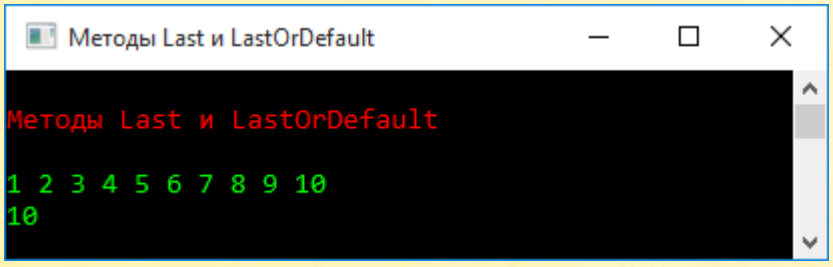


Методы *Last* и *LastOrDefault*

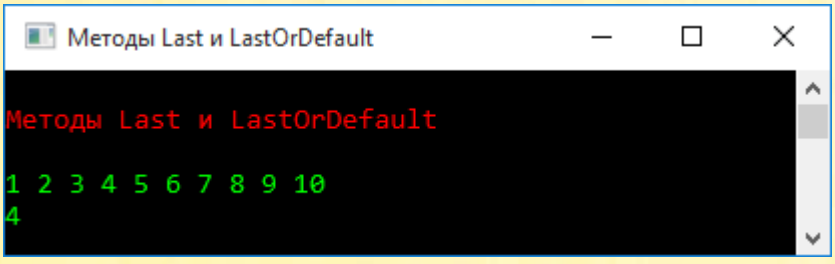
Методы Last и LastOrDefault похожи на предыдущие методы, но возвращают *последний* элемент последовательности:

```
function Last(): T;
function Last(predicate: T->boolean): T;
function LastOrDefault(): T;
function LastOrDefault(predicate: T->boolean): T;
```

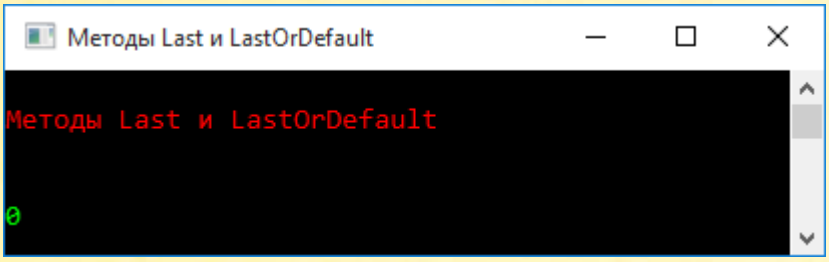
```
var sq := Range(1,10).Println;
Println(sq.Last);
```




```
var sq :=
Range(1,10).Println;
Println(sq.Last(n -> n
< 5));
```



```
var sq := Range(10,1).Println;
Println(sq.LastOrDefault(n -> n < 5));
```

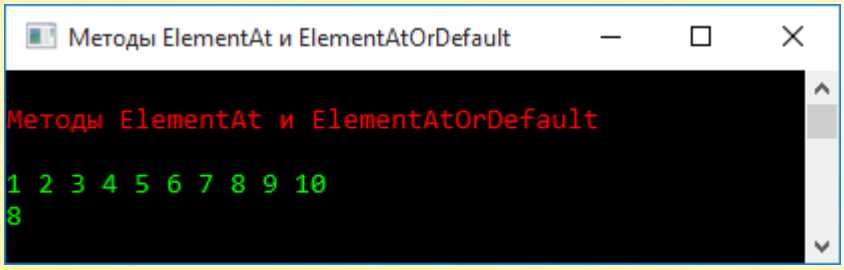


Методы *ElementAt* и *ElementAtOrDefault*

Методы ElementAt и ElementAtOrDefault возвращают элемент последовательности по его номеру (индексу). Нумерация начинается с нуля:

```
function ElementAt(index: integer): T;
```

```
var sq := Range(1,10).Println;
Println(sq.ElementAt(7));
```

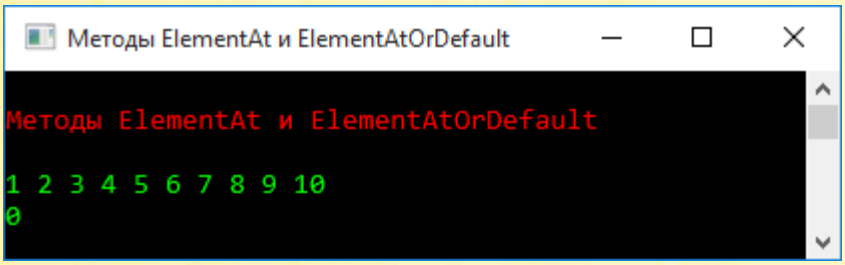


Будьте внимательны: элемент с индексом 7 занимает восьмое место в последовательности!

Метод ElementAtOrDefault действует так же, как *ElementAt*, но, если указанный номер лежит за границами последовательности, то он возвращает значение по *умолчанию*:

```
function ElementAtOrDefault(index: integer): T;
```

```
var sq := Range(1,10).Println;
Println(sq.ElementAtOrDefault(10));
```



Индексу 10 соответствует 11-й элемент последовательности. Но она содержит только 10 элементов, поэтому метод *ElementAtOrDefault* возвращает значение по умолчанию, то есть 0.

Пятизначный куб (Range.ElementAt)

Задача 556 из книги *Математическая шкатулка* [Нагибин88]:

> Задумано пятизначное число, являющееся кубом натурального числа.
>
> Восстановите задуманное число, если известно, что оно должно делиться на 3 и последняя цифра его 6.

Проще не извлекать кубические корни из чисел, а возводить их в куб. Пределы изменения основания степени легко найти:

```
// минимальное число:
var minn:= Trunc(Power(10000, 1/3));
// максимальное число:
var maxn:= Trunc(Power(99999, 1/3));
```

Метод Range генерирует последовательность пятизначных чисел:

```
Range(minn, maxn)
```

Из которой мы отбираем числа, удовлетворяющие условиям задачи:

```
Where(n -> (n*n*n mod 3 = 0) and (n*n*n mod 10 = 6))
```

Если вы распечатаете последовательность отфильтрованных чисел, то увидите, что в ней единственное число. Чтобы напечатать ответ на экране, мы извлекаем его:

```
ElementAt(0);
```

И присваиваем переменной res.

Задача решена:

```
uses
   System;

// Нагибин 556

begin
   // заголовок окна:
   Console.Title := 'Нагибин 556';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Пятизначный куб');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // минимальное число:
   var minn:= Trunc(Power(10000, 1/3));
   // максимальное число:
   var maxn:= Trunc(Power(99999, 1/3));

   // решаем задачу:
```

```
    var res:= Range(minn, maxn)
              .Where(n -> (n*n*n mod 3 = 0) and (n*n*n mod 10 = 6))
              .ElementAt(0);
    // печатаем ответ:
    Println( res, 'в кубе = ', (res*res*res));

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Рисунок показывает, что задумано было число **46 656** (в книге указан неверный ответ – 45 656):

Репетитор (Range.ElementAt)

В те далёкие времена, когда компьютеров ещё и в помине не было, школьники уже вовсю решали задачки по математике, что давалось им нелегко. Этот познавательный процесс хорошо описан Антоном Павловичем Чеховым в рассказе *Репетитор*.

Я надеюсь, что вы с удовольствием прочитаете этот рассказ Чехова от начала до конца, поэтому перейдём непосредственно к **задаче**:

> *Теперь по арифметике... Берите доску. Какая следующая задача?*
> *Петя плюёт на доску и стирает рукавом. Учитель берёт задачник и диктует:*

- «Купец купил 138 арш. чёрного и синего сукна за 540 руб. Спрашивается, сколько аршин купил он того и другого, если синее стоило 5 руб. за аршин, а чёрное 3 руб.?»

Тяготы репетиторского труда мы пропускаем и читаем финал этой арифметической истории:

- И без алгебры решить можно, - говорит Удодов, протягивая руку к счётам и вздыхая. - Вот, извольте видеть...

Он щёлкает на счётах, и у него получается 75 и 63, что и нужно было.

Итак, ответ на задачу известен, и нам только и остаётся, что заменить старые счёты современными, то есть компьютером.

Обозначаем **длину** (*length*) сукна и **стоимость** (*cost*) соответствующими **константами**:

```
const
    //общая длина сукна в аршинах:
    LENGTH: integer = 138;
    //общая стоимость сукна в рублях:
    COST: integer  = 540;
```

На *COST* рублей можно купить не более **maxBlue** аршин синего сукна:

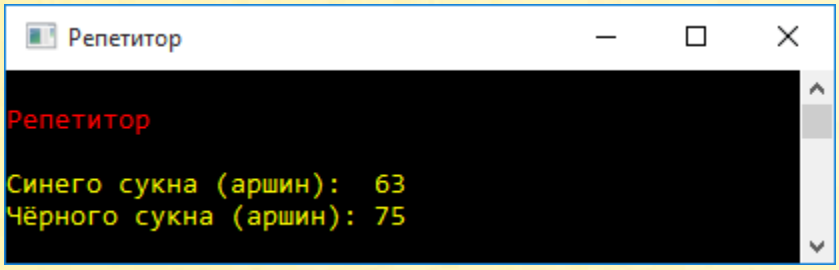
```
begin
    // заголовок окна:
    Console.Title := 'Репетитор';
    Console.WriteLine('');
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Репетитор');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    // макс. длина синего сукна:
    var maxBlue := COST div 5;
```

Из последовательности чисел 0..maxBlue нам нужно отобрать такие, которые удовлетворяют условиям задачи.

Условие задачи записываем в методе расширения Where. В итоге мы получаем последовательность, состоящую из *одного* элемента. Нужно взять его методом ElementAt, чтобы переменная blue имела тип *integer*:

```
    // решаем задачу:
    var blue := Range(0, maxBlue)
                .Where(n -> n * 5 + (LENGTH - n) * 3 = COST)
                .ElementAt(0);
    Writeln('Синего сукна (аршин):  ', blue);
    Writeln('Чёрного сукна (аршин): ' + (LENGTH - blue));

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Запускаем программу и получаем ответ:



Американское наследство (Range.ElementAt)

Задача про американские деньги:

Отец оставил сыновьям Чарлзу и Роберту 100 долларов.

Если одну треть части Чарлза вычесть из одной четверти части Роберта, то останется 11 долларов.

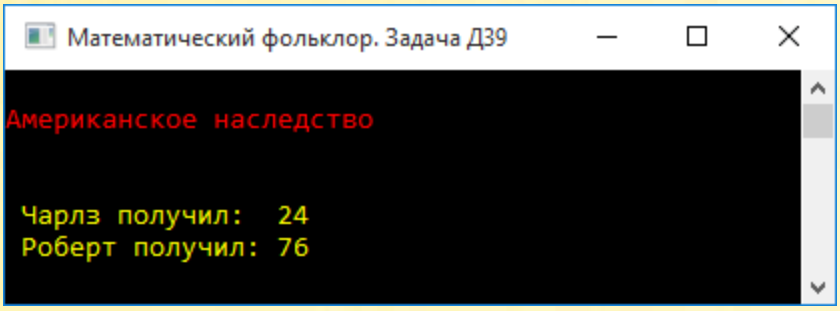
Сколько долларов получил каждый из братьев?

Задача про американское наследство решается так же, как про русское сукно:

```
uses
   System;

// Фольклор Д39


begin
   // заголовок окна:
   Console.Title := 'Математический фольклор. Задача Д39';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Американское наследство');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

    // решаем задачу:
    var Charles := Range(0, 100)
                   .Where(n -> (100 - n) div 4 - n div 3 = 11)
                   .ElementAt(0);
    Writeln(' Чарлз получил:  ', Charles);
    Writeln(' Роберт получил: ' + (100 - Charles));

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
```

```
end.
```

А вот и юридический отчёт о проделанной нами работе:



Четыре числа (Step.ElementAt)

Задача 341 из книги *Математическая шкатулка* [Нагибин88]:

> Произведение четырёх последовательных натуральных чисел равно 3024.
>
> Найдите эти числа.

Понятно, что задача имеет **единственное** решение. Если мы найдём 4 последовательных числа, произведение которых равно 3024, то следующая четвёрка даст большее произведение.

Условие задачи проверяем в методе **Where**. При первом же выполнении условия мы извлекаем из отфильтрованной последовательности **первое** число, и на этом генерирование чисел заканчивается:

```
uses
   System;

// Нагибин 341
```

```
begin
   // заголовок окна:
   Console.Title := 'Нагибин 341';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Четыре числа');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var m := 1;
   var n := m.Step()
            .Where(i -> i * (i + 1) * (i + 2) * (i + 3) = 3024)
            .ElementAt(0);
   // печатаем ответ:
   Println(n, (n + 1), (n + 2), (n + 3));

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

На рисунке вы видите, что в произведении участвуют числа 6, 7, 8 и 9:

Разность кубов (Step.ElementAt)

Задача 561 из книги *Математическая шкатулка* [Нагибин88]:

$$x^2 - y^2 = (x + y)(x - y)$$
$$(x \pm y)^2 = x^2 \pm 2xy + y^2$$
$$x^3 + y^3 = (x + y)(x^2 - xy + y^2)$$
$$x^3 - y^3 = (x - y)(x^2 + xy + y^2)$$

Разность кубов двух последовательных натуральных чисел равна 331.

Восстановите эти числа.

Генерируем бесконечную последовательность натуральных чисел – до тех пор, пока не найдём такую пару, которая удовлетворяет условию задачи:

```
uses
   System;

// Нагибин 561


begin
   // заголовок окна:
   Console.Title := 'Нагибин 561';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Разность кубов');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var m:= 1;
   var res:= m.Step()
            .Where(n -> (n+1)*(n+1)*(n+1) - n*n*n = 331)
            .ElementAt(0);
   // печатаем ответ:
   Println( res, 'в кубе = ', (res*res*res));
   res += 1;
   Println(res, 'в кубе = ', (res*res*res));

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Простой ответ на простую задачу:

Двузначное число (Range.ElementAt)

Задача 563 из книги *Математическая шкатулка* [Нагибин88]:

> Найдите двузначное положительное число, равное произведению суммы и разности его цифр.

Последовательность двузначных чисел генерирует метод Range:

```
Range(10,99)
```

Для каждого числа нужно найти разность и сумму его цифр. Чтобы не усложнять проверку в методе *Where*, мы напишем 2 функции:

```
var sub: integer -> integer := x -> Abs(x div 10 - x mod 10);
var sum: integer -> integer := x -> x div 10 + x mod 10;
```

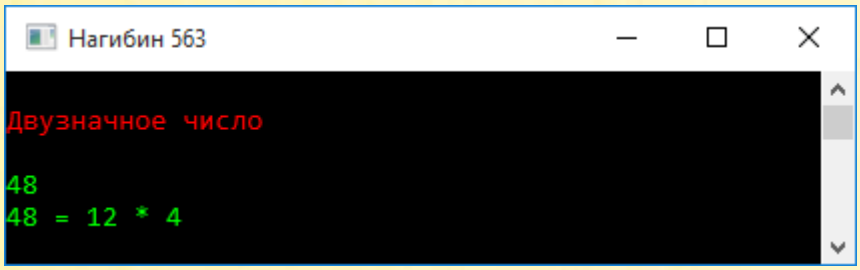
Здесь мы должны проследить, чтобы разность цифр была неотрицательной. Теперь задача решается без проблем:

```
uses
   System;

// Нагибин 563


begin
   // заголовок окна:
   Console.Title := 'Нагибин 563';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Двузначное число');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // функции для вычисления разности и суммы цифр
```

```
    // заданного числа:
    var sub: integer -> integer := x -> Abs(x div 10 - x mod 10);
    var sum: integer -> integer := x -> x div 10 + x mod 10;
    // находим число:
    var res:= Range(10,99)
              .Where(n -> sum(n)*sub(n) = n)
              .Println
              .ElementAt(0);
    // печатаем ответ:
    Console.WriteLine('{0} = {1} * {2}', res, sum(res), sub(res));

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

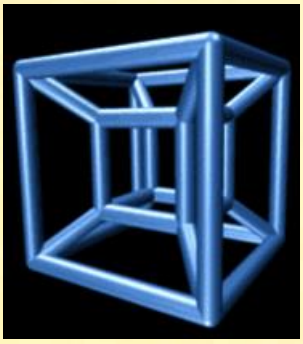
Метод **Println** распечатывает отфильтрованную последовательность, чтобы мы могли узнать, сколько решений имеет задача.

А решение *единственное*: двузначное число равно **48**. Сумма его цифр - 12, а разность – четыре:



Квадратный куб (Range.ElementAt)

Задача 584 из книги *Математическая шкатулка* [Нагибин88]:



> **Какое трёхзначное число** равно кубу цифр его единиц, а также квадрату числа, составленного из второй и первой цифр?

Генерируем все трёхзначные числа в методе Range:

```
Range(100,999)
```

Условия довольно сложные, поэтому пишем отдельные функции.

Последняя цифра числа равна остатку от его деления на 10:

```
var ed: integer -> integer := x -> x mod 10;
```

Её нужно возвести в куб:

```
var cube: integer -> integer := x -> x * x * x;
```

Немного труднее переставить местами первую и вторую цифры числа:

```
var n21: integer -> integer := x ->
                                (x div 10 mod 10) * 10 + x div 100;
```

А число мы возводим в квадрат:

```
var quad: integer -> integer := x -> x * x;
```

Теперь нам не составит труда записать условия задачи в методе Where:

```
Where(n -> (cube(ed(n)) = n) and (quad(n21(n)) = n))
```

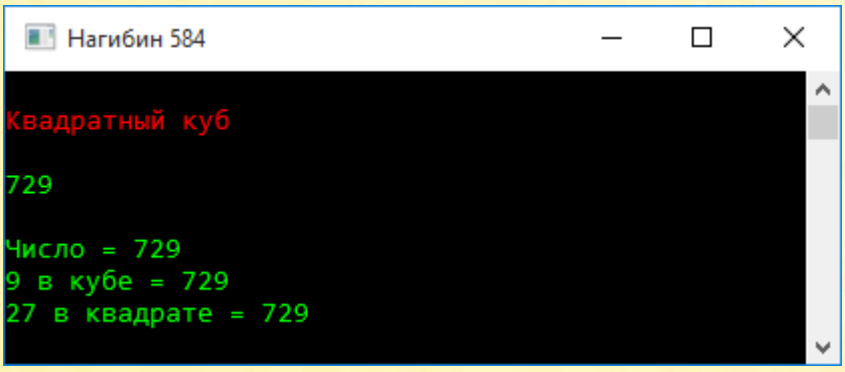
Если очередное трёхзначное число *i* выдержало проверки, то мы печатаем ответ на экране:

```
uses
   System;
```

```
/ /Нагибин 584


begin
   // заголовок окна:
   Console.Title := 'Нагибин 584';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Квадратный куб');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var ed: integer -> integer := x -> x mod 10;
   var cube: integer -> integer := x -> x * x * x;
   var n21: integer -> integer := x ->
                                  (x div 10 mod 10) * 10 + x div 100;
   var quad: integer -> integer := x -> x * x;
   var res:= Range(100,999)
            .Where(n -> (cube(ed(n)) = n) and (quad(n21(n)) = n))
            .Println
            .ElementAt(0);

   // печатаем ответ:
   Console.WriteLine();
   Console.WriteLine('Число = {0}', res);
   Console.WriteLine('{0} в кубе = {1}', ed(res), res);
   Console.WriteLine('{0} в квадрате = {1}', n21(res), res);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Задача имеет *единственное* решение:

$729 = 9^3 = 27^2$



Четырёхзначное число (Range.ElementAt)

Задача 620 из книги *Математическая шкатулка* [Нагибин88]:

> **Найдите нечётное четырёхзначное число**, две средние цифры которого образуют число в пять раз больше числа тысяч и втрое больше числа единиц этого числа.

Весь «фокус» этой задачи заключается в том, чтобы правильно **извлечь цифры** из числа:

```
// две средние цифры:
var n23: integer -> integer := x -> x div 10 mod 100;
// число тысяч:
var n1: integer -> integer := x -> x div 1000;
// число единиц:
var n4: integer -> integer := x -> x mod 10;
```

Так как мы не можем быть уверены, что задача имеет единственное решение, то печатаем всю отфильтрованную последовательность.

Генерируем последовательность нечётных четырёхзначных чисел методом **Range**:

```
Range(1001,9999,2)
```

Начинаем с первого нечётного четырёхзначного числа и заканчиваем последним – 9999. Шаг в методе *Range* равен **двум**.

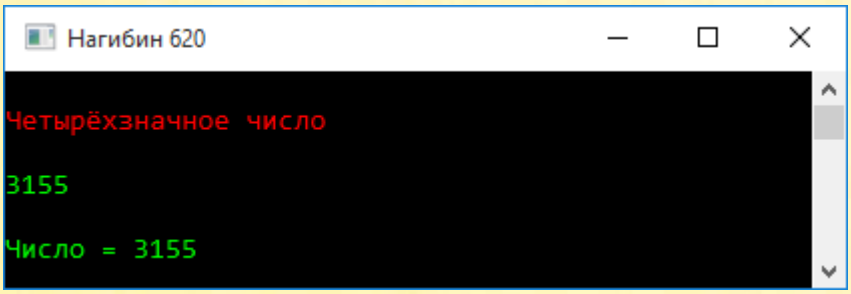
С нашими функциями условия в методе **Where** записываются вполне понятно:

```
Where(n -> (n23(n) = n1(n) * 5) and (n23(n) = n4(n) * 3))
```

Вся программа целиком:

```
uses
   System;

// Нагибин 620

begin
   // заголовок окна:
   Console.Title := 'Нагибин 620';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Четырёхзначное число');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу -->

   // две средние цифры:
   var n23: integer -> integer := x -> x div 10 mod 100;
   // число тысяч:
   var n1: integer -> integer := x -> x div 1000;
   // число единиц:
   var n4: integer -> integer := x -> x mod 10;

   var res:= Range(1001,9999,2)
             .Where(n -> (n23(n) = n1(n) * 5) and (n23(n) = n4(n) * 3))
             .Println
             .ElementAt(0);

   // печатаем ответ:
   Console.WriteLine();
   Console.WriteLine('Число = {0}', res);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Ответ на задачу показан на рисунке.



Зачёркнутая цифра (Range.ElementAt)

Задача 600 из книги *Математическая шкатулка* [Нагибин88]:

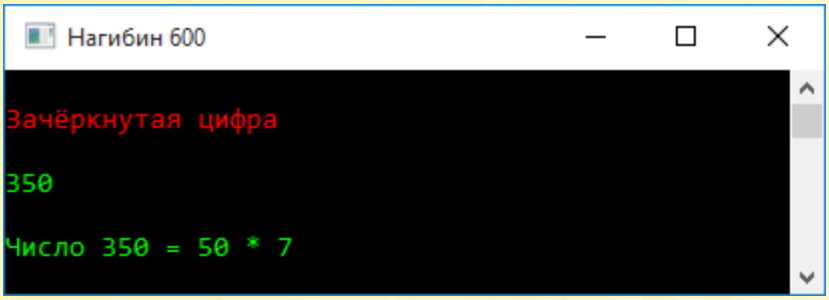
> В трёхзначном числе зачеркнули цифру сотен, затем полученное двузначное число умножили на 7 и получили вновь исходное трёхзначное число.
>
> Какое это число?

Очень простая задача с простым же условием в методе Where:

```
uses
   System;

// Нагибин 600


begin
   // заголовок окна:
   Console.Title := 'Нагибин 600';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Зачёркнутая цифра');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var res:= Range(100,999)
             .Where(n -> n mod 100 * 7 = n)
             .Println
             .ElementAt(0);

   // печатаем ответ:
   Console.WriteLine();
   Console.WriteLine('Число {0} = {1} * 7', res, res mod 100);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
```

```
end.
```

А вот и **ответ** на задачу:



Зачёркнутая цифра 2 (Range.ElementAt)

Задача 587 из книги *Математическая шкатулка* **[Нагибин88]**:

> В трёхзначном числе зачеркнули среднюю цифру. Полученное двузначное число оказалось в 6 раз меньше исходного трёхзначного.
>
> Какое такое трёхзначное число.

Самая «сложная» часть задачи заключается в зачёркивании **средней** цифры. В результате получается двузначное число, которое мы вычисляем в функции **n2**:

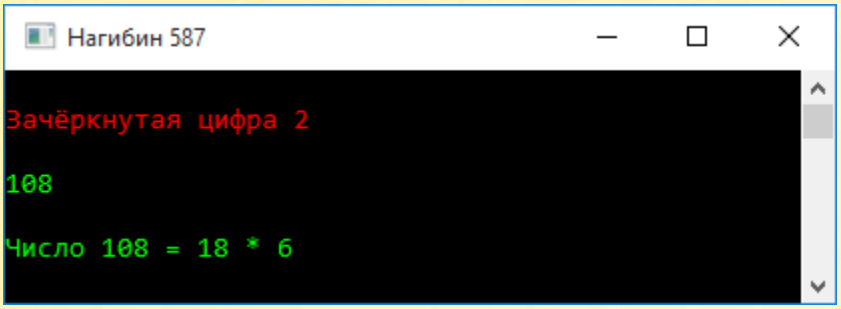
```
//двузначное число:
var n2: integer -> integer := x -> x div 100 * 10 + x mod 10;
```

Остальная часть программы не представляет ни малейшего труда:

```
uses
   System;
```

```
// Нагибин 587


begin
   //заголовок окна:
   Co nsole.Title := 'Нагибин 587';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Зачёркнутая цифра 2');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу -->

   // двузначное число:
   var n2: integer -> integer := x -> x div 100 * 10 + x mod 10;

   var res:= Range(100,999)
             .Where(n -> n = n2(n) * 6)
             .Println
             .ElementAt(0);

   // печатаем ответ:
   Console.WriteLine();
   Console.WriteLine('Число {0} = {1} * 6', res, n2(res));

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Вот и вся задача:

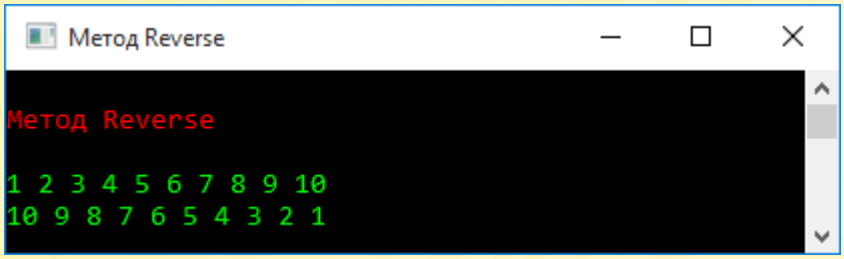


Метод *Reverse*

Метод Reverse возвращает последовательность, в которой элементы располагаются в *обратном порядке*:

```
function Reverse(): sequence of T;
```

```
var sq := Range(1, 10).Println;
sq.Reverse.Println;
```



Методы *OrderBy* и *OrderByDescending*

Метод OrderBy возвращает коллекцию типа *IOrderedEnumerable*, отсортированную в порядке *возрастания* значения ключа функцией *keySelector*:

```
function OrderBy<Key>(keySelector: T->Key):
                        System.Linq.IOrderedEnumerable<T>;
```

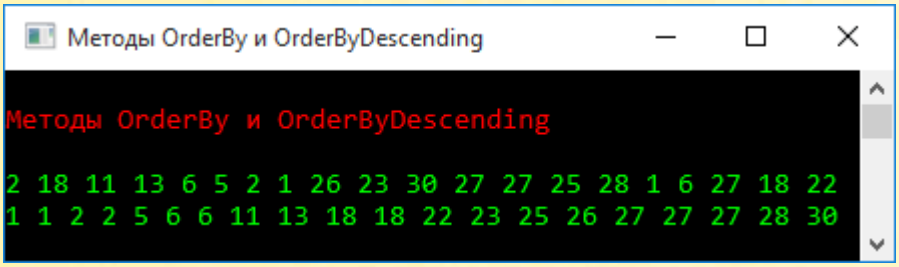
При равенстве значения ключей порядок элементов сохраняется (*строгая сортировка*). Функция keySelector определяет условия сортировки. Например, если мы хотим отсортировать коллекцию по значению элементов, то условие должно быть самым простым - *(x -> x)*. Числа сортируются по величине, а слова – в алфавитном порядке.

Создаём последовательность из 20 случайных чисел:

```
var sq := SeqRandom(20, 1, 30).ToArray.Println;
```

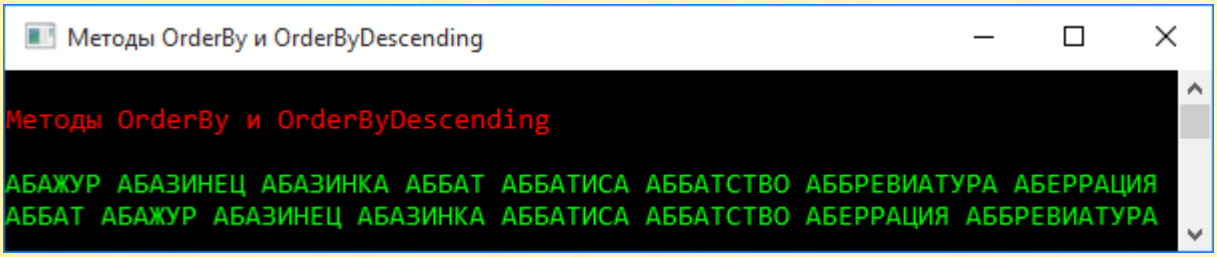
И сортируем их по величине:

```
sq.OrderBy(n -> n).Println;
```



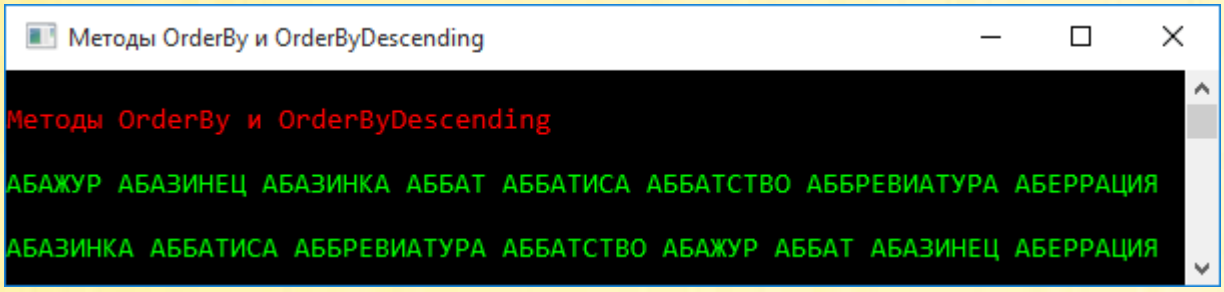
Задавая более сложные условия, можно, например, отсортировать слова по длине:

```
var lstStrAll := ReadAllLines('OSH-W97.txt');
var lstStr:= lstStrAll.Take(8).Println;
lstStr.OrderBy(s -> s.Length).Println;
```



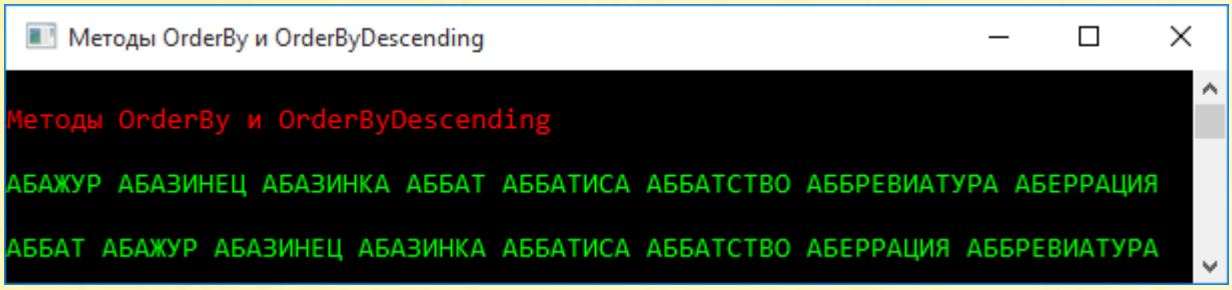
Или по последней букве:

```
lstStr.OrderBy(s -> s.Last()).Println;
```



Сортировка слов в «кроссвордном» порядке легко достижима последовательным применением методов OrderBy с соответствующими условиями:

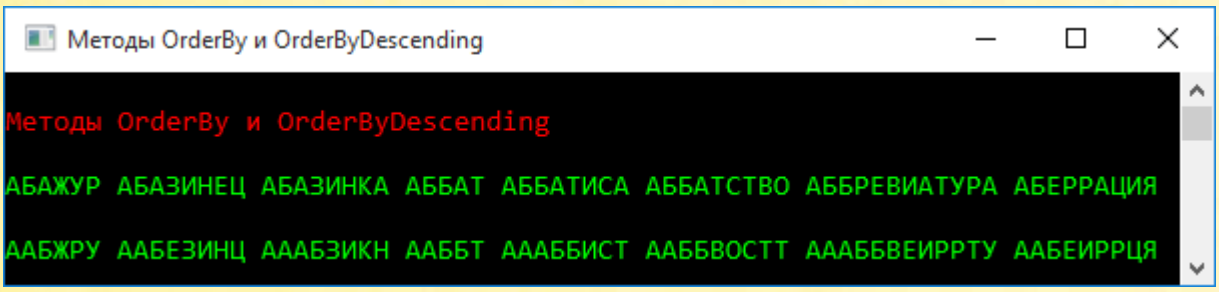
```
lstStr.OrderBy(s -> s)
      .OrderBy(s -> s.Length)
      .Println;
```



В отличие от простой сортировки по длине здесь слова равной длины дополнительно отсортированы *по алфавиту*.

А вот так просто можно отсортировать буквы в словах:

```
var chs := lstStr.Select(str ->
                  new string(str.ToCharArray.OrderBy(c -> c).ToArray))
                 .Println;
```



Второй метод OrderBy сортирует заданную последовательность, используя заданный компаратор *comparer:*

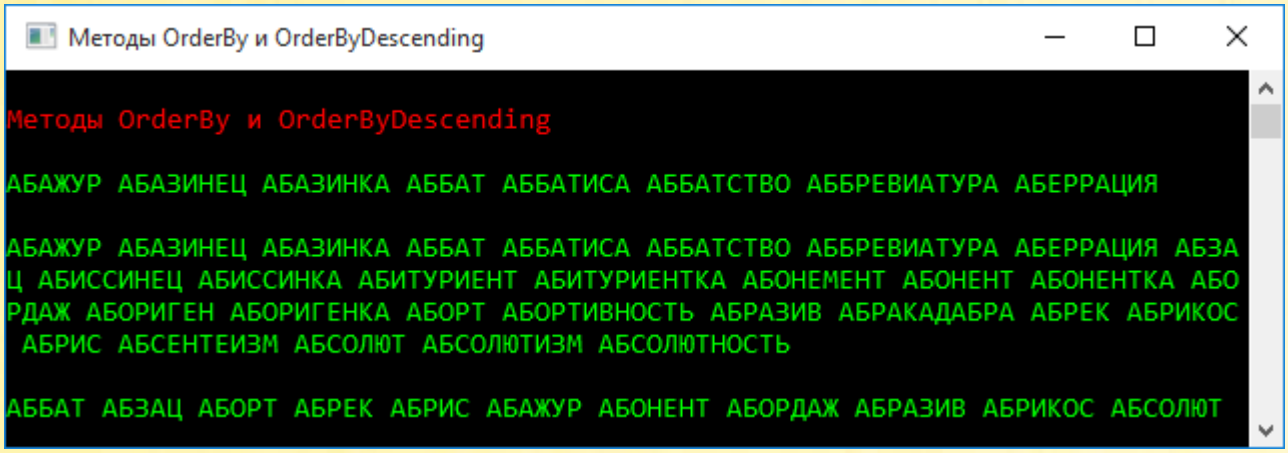
```
function OrderBy<Key>(keySelector: T->Key; comparer: IComparer<Key>):
                        System.Linq.IOrderedEnumerable<T>;
```

В методе Compare можно задать любые, самые сложные условия сравнения каждой пары слов. Например, для сравнения мы можем сравнивать слова по длине, а если они имеют одинаковую длину, то по алфавиту:

```
type
   StrComparer = class(IComparer<string>)
   public
      function Compare(s1, s2: string): integer;
      begin
         var len1 := s1.Length;
         var len2 := s2.Length;
         if (len1 <> len2) then
             Result:= s1.Length.CompareTo(s2.Length)
         else
         Result := s1.CompareTo(s2);
      end;
   end;
```

Чтобы сделать отсортированный список более наглядным, мы сначала выберем слова, в которых не более 7 букв, а затем сравним их с помощью нашего компаратора:

```
lstStr := lstStrAll.Take(30).Println;
Println;
lstStr.Where(s -> s.Length < 8)
       .OrderBy(s -> s, new StrComparer())
       .Println;
```



Методы OrderByDescending возвращают коллекцию типа *IOrderedEnumerable*, отсортированную в порядке убывания значения ключа функцией *keySelector*:

```
function OrderByDescending<Key>(keySelector: T->Key):
                                System.Linq.IOrderedEnumerable<T>;

function OrderByDescending<Key>(keySelector: T->Key;
                                comparer: IComparer<Key>):
                                System.Linq.IOrderedEnumerable<T>;
```

Методы *ThenBy* и *ThenByDescending*

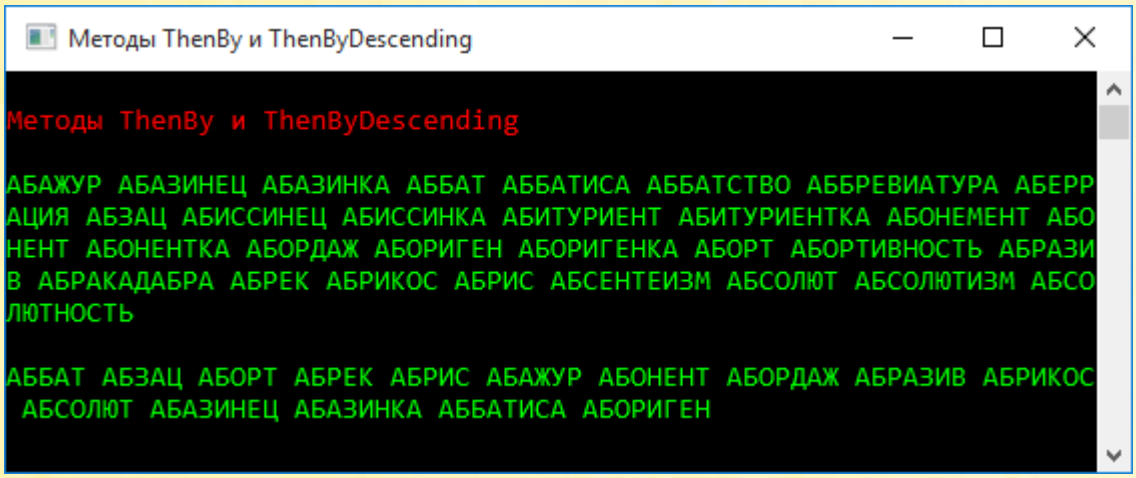
Метод ThenBy применяют для дополнительной сортировки последовательности типа *IOrderedEnumerable*, которую он получает после выполнения методов *OrderBy* или *OrderByDescending*, по возрастанию значения ключа:

```
function ThenBy<Key>(keySelector: T->Key): System.Linq.IOrderedEnumerable<T>;
```

При равенстве значения ключей порядок элементов сохраняется (*строгая сортировка*).

Опять выполним кроссвордную сортировку для слов из списка *lstStr*, которые не длиннее 8 букв (*Where*). Сначала мы сортируем слова по *длине* (OrderBy), а затем – в пределах слов равной длины – по *алфавиту* (ThenBy):

```
var lstStrAll := ReadAllLines('OSH-W97.txt');
var lstStr := lstStrAll.Take(30).Println;
Println;

lstStr.Where(s -> s.Length < 9)
      .OrderBy(s -> s.Length)
      .ThenBy(s -> s)
      .Println;
```



Второй метод ThenBy дополнительно сортирует заданную последовательность, используя заданный компаратор *comparer*:

```
function ThenBy<Key>(keySelector: T->Key; comparer: IComparer<Key>):
                    System.Linq.IOrderedEnumerable<T>;
```

Методы ThenByDescending дополнительно сортируют заданную последовательность в порядке убывания значения ключа:

```
function ThenByDescending<Key>(keySelector: T->Key):
                              System.Linq.IOrderedEnumerable<T>;

function ThenByDescending<Key>(keySelector: T->Key; comparer: IComparer<Key>):
                              System.Linq.IOrderedEnumerable<T>;
```

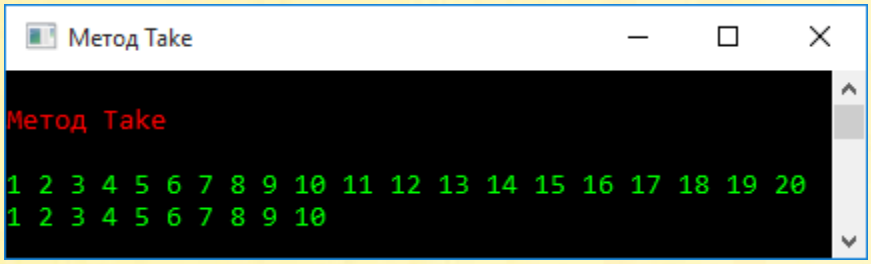
Методы *Take* и *TakeWhile*

Метод Take возвращает первые *count* элементов последовательности:

```
function Take(count: integer): sequence of T;
```

Создаём последовательность из 20 элементов и отбираем из неё первые 10 элементов методом Take:

```
var sq := Range(1,20).Println;
sq.Take(10).Println;
```



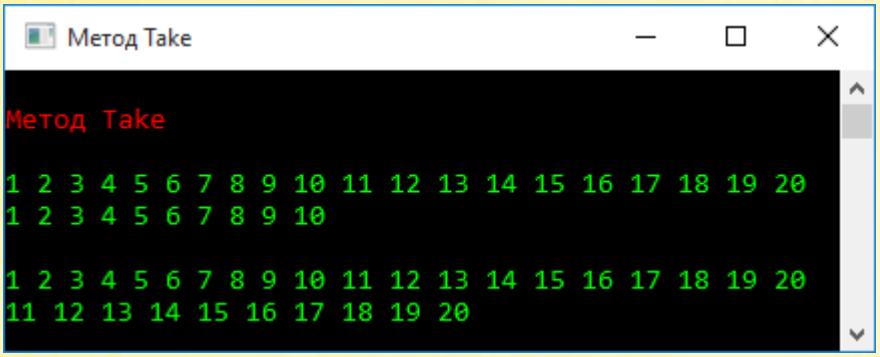
Если значение параметра *count* больше числа элементов в последовательности, то метод Take вернёт всю последовательность.

Метод TakeLast возвращает последние *count* элементов последовательности:

```
function TakeLast(count: integer): sequence of T;
```

Создаём такую же последовательность, что и в первом примере, но отбираем 10 *последних* элементов:

```
Println;
sq := Range(1,20).Println;
sq.TakeLast(10).Println;
```

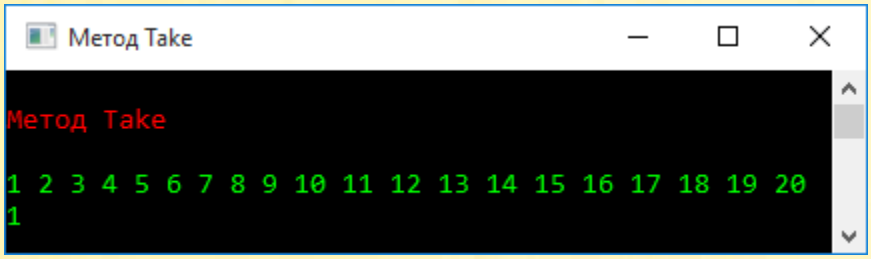


Метод TakeWhile возвращает элементы последовательности, удовлетворяющие заданному условию, пока не встретится первый элемент, не удовлетворяющий этому условию:

```
function TakeWhile(predicate: T->boolean): sequence of T;
```

Отберём из последовательности первые *нечётные* числа:

```
var sq := Range(1,20).Println;
sq.TakeWhile(n -> n mod 2 <> 0).Println;
```



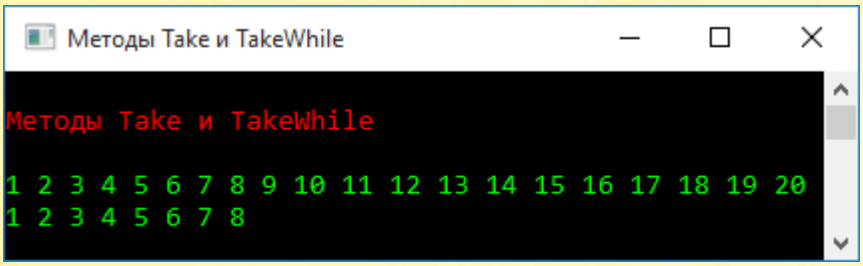
1 – нечётное число. Напечатано.
2 – чётное число – метод *TakeWhile* заканчивает работу.

Второй метод TakeWhile возвращает элементы последовательности, удовлетворяющие заданному условию с учётом *индексов*, пока не встретится первый элемент, не удовлетворяющий этому условию:

```
function TakeWhile(predicate: (T,integer)->boolean): sequence of T;
```

Из той же последовательности натуральных чисел выберем такие, которые имеют значение меньше 9 и индекс меньше 12:

```
var sq := Range(1,20).Println;
sq.TakeWhile((n, i) -> (i < 12) and (n < 9)).Println;
```



Индийские обезьяны 2 (Step.Take)

Квадрат восьмой части стада обезьян играют в роще, а остальные 12 – на горке.

Сколько обезьян в стаде?

Эта задача решается с помощью **квадратного уравнения**, у которого 2 действительных корня. Если задача может иметь несколько решений, то лучше заменить метод **SkipWhile** методом **Where**, который из бесконечной последовательности отберёт только те числа, которые удовлетворяют заданному условию. При этом метод **Step** будет выдавать бесконечную последовательность. У задачи ровно **2 решения**, поэтому после метода **Where** нужно поставить метод **Take**(2). Так мы получим оба решения и остановим генерирование элементов.

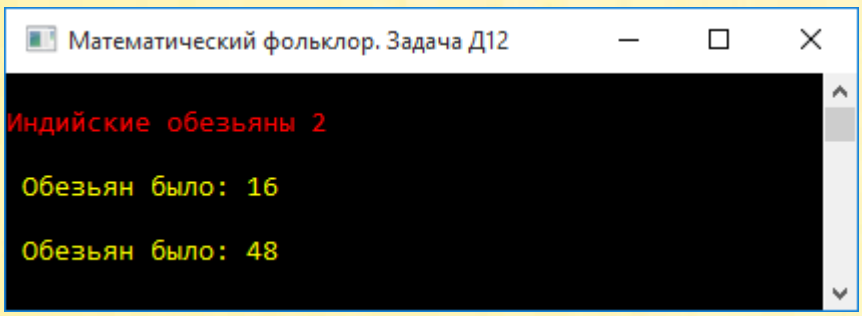
Если условие длинное, то его можно записать как **функцию**:

```
var f: integer -> boolean :=  n -> n - (n div 8) * (n div 8) = 12;
```

Число обезьян заведомо кратно 8, поэтому генерируем последовательность от 0 с шагом 8:

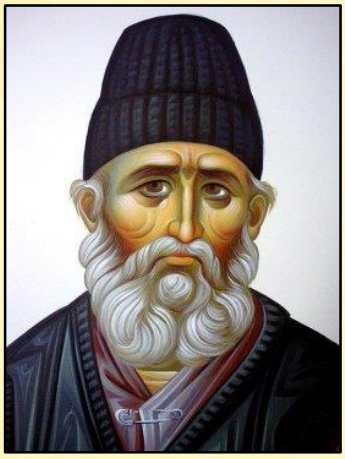
```
uses
   System;

// Фольклор Д12


begin
   // заголовок окна:
   Console.Title := 'Математический фольклор. Задача Д12';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Индийские обезьяны 2');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу
   var m := 0;
   var f: integer -> boolean :=  n -> n - (n div 8) * (n div 8) = 12;
   Print(' Обезьян было: ');
   m.Step(8)
    .Where(f)
    .Take(2)
    .Println;

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Два ответа на одну задачу:



Жизнь Демохара (Step.Take)

Множество занимательных задач связано с вычислением **возраста**. Одна из первых появилась в Греции в конце IV века:



> *Демохар четверть жизни прожил мальчиком, пятую часть – юношей, одну треть – зрелым мужчиной и 13 лет пожилым.*
>
> *Сколько лет прожил Демохар?*

Простейшая тренировочная задача. Очевидно, что возраст Демохара нацело делится на 3, 4 и 5. Но мы предположим, что это не так и генерируем последовательность чисел типа *double*. Условие задачи удобно записать в виде **функции**:

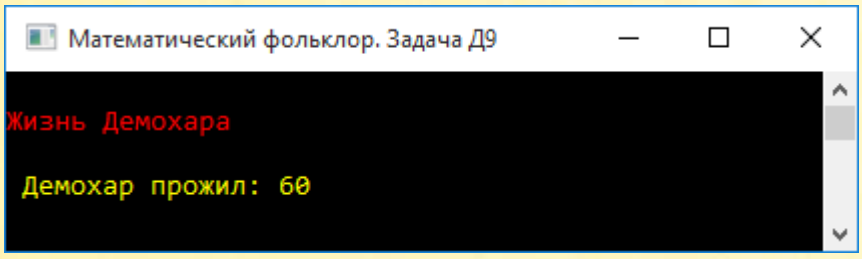
```
var f: double -> boolean :=  n -> n / 4 + n / 5 + n / 3 + 13 = n;
```

Вся программа целиком:

```
uses
   System;

// Математический фольклор. Задача Д9


begin
   // заголовок окна:
   Console.Title := 'Математический фольклор. Задача Д9';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Жизнь Демохара');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу
   var v := 0.0;
   var f: double -> boolean :=  n -> n / 4 + n / 5 + n / 3 + 13 = n;
```

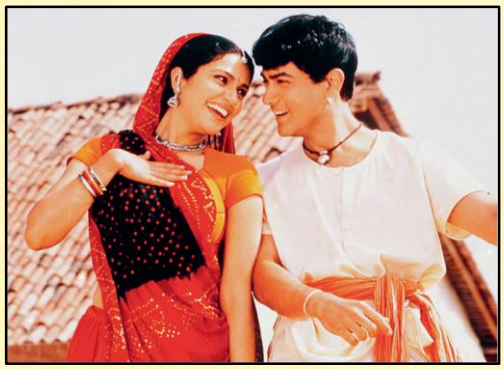
```
    Print(' Демохар прожил: ');
    v.Step(1)
     .Where(f)
     .Take(1)
     .Println;

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Демохар прожил 60 лет:



Индийское число (Step.Take)

Условие задачи:



Если некоторое число умножить на 5, от произведения отнять его треть, остаток разделить на 10 и к полученному числу прибавить последовательно одну треть, одну вторую и одну четвёртую первоначального числа, то получится 68.

Какое это число?

Мы можем только посочувствовать древним индийцам, у которых не было компьютеров, чтобы решать такие задачи.

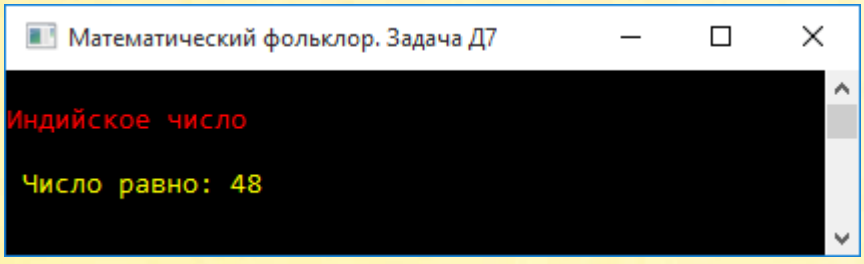
Задача очень простая. Главное - правильно записать её условие:

```
var f: integer -> boolean :=  n -> (n * 5 - n * 5 div 3)
                          div 10 + n div 3 + n div 2 + n div 4 = 68;
```

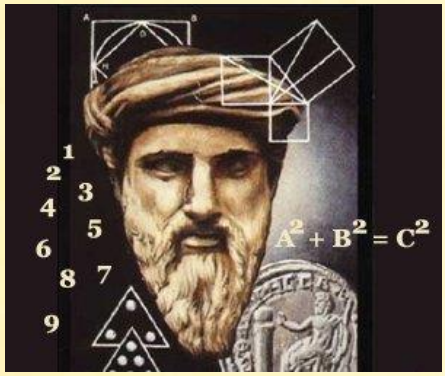
Вот и вся задача:

```
uses
   System;

// Фольклор Д7

begin
   // заголовок окна:
   Console.Title := 'Математический фольклор. Задача Д7';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Индийское число');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу
   var m := 1;
   var f: integer -> boolean :=  n -> (n * 5 - n * 5 div 3)
          div 10 + n div 3 + n div 2 + n div 4 = 68;
   Print(' Число равно: ');
   m.Step(1)
    .Where(f)
    .Take(1)
    .Println;

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Искомое число должно делиться нацело на 2, 3 и 4, поэтому мы могли бы значительно сократить перебор, но в данном случае этого не требуется.



Пифагорейское число (Step.Take)



Знаменитого Пифагора спросили:

- Сколько учеников посещают твою школу и слушают твои беседы?

Пифагор ответил:

- Половина моих учеников изучает математику, четверть – музыку, седьмая часть проводит время в молчаливом размышлении, остальную часть составляют 3 ученика.

Сколько учеников было у Пифагора?

Эта задача напечатана и в книге *Увлекательная математика. Античные этюды. Задача 5. Пифагор Самосский (ок. 508-501 гг. до н.э.)*:

Поликрат (известный из баллады Шиллера тиран с острова Самос) однажды спросил на пиру у Пифагора, сколько у того учеников.

- Охотно скажу тебе, о Поликрат, - отвечал Пифагор. - Половина моих учеников изучает прекрасную математику, четверть исследует тайны вечной природы, седьмая часть молча упражняет силу духа, храня в сердце учение. Добавь ещё к ним трёх юношей, из которых Теон превосходит прочих своими способностями.

Сколько учеников было у Пифагора?

Очевидно, что число учеников должно быть кратно 4 и 7, но тогда нечего решать. Поэтому мы проигнорируем это наблюдение и выберем для переменной *m* тип *double*.

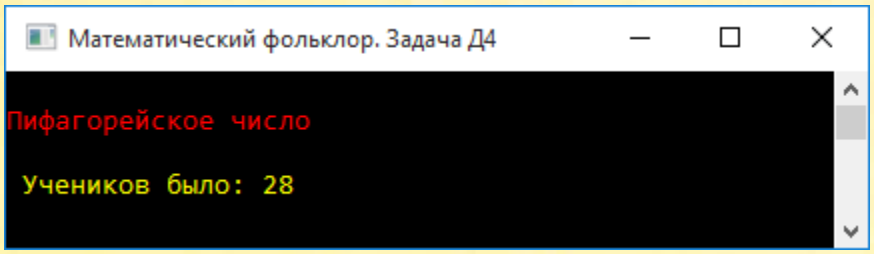
Условие легко выводится из текста задачи:

```
var f: double -> boolean :=  n -> n / 2 + n / 4 + n / 7 + 3 = n;
```

И всё равно решение задачи очень простое:

```
uses
   System;

// Фольклор Д4


begin
   // заголовок окна:
   Console.Title := 'Математический фольклор. Задача Д4';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Пифагорейское число');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу
   var m := 1.0;
   var f: double -> boolean :=  n -> n / 2 + n / 4 + n / 7 + 3 = n;

   Print(' Учеников было: ');
   m.Step(1)
    .Where(f)
    .Take(1)
    .Println;

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

У Пифагора было 28 учеников:



Диофантово число (Step.Take)

Задача древнегреческого математика Диофанта Александрийского, жившего в третьем веке:

> *По двум данным числам 200 и 5 найти третье число, которое, если его умножить на одно из них, даёт полный квадрат, а если его умножить на другое число, даёт квадратный корень из этого квадрата.*

Генерируем бесконечную последовательность, начиная с 1. Условие задачи переводим в лямбда-выражение:

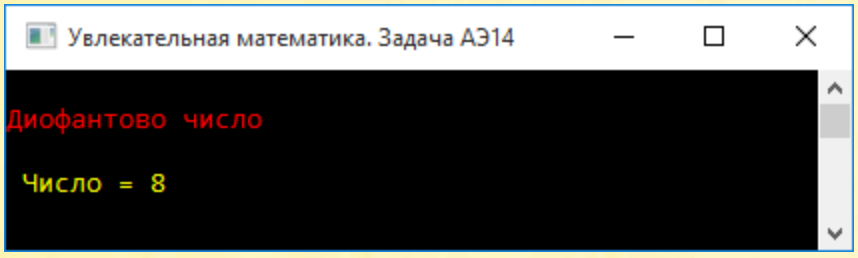
```
var f: integer -> boolean :=  n -> (n * 200).IsQuadrat and
                                   ((n * 200).Sqrt = 5 * n);
```

Здесь мы использовали метод расширения IsQuadrat для типа *integer*. Других сложностей в задаче нет:

```
uses
   System, OlympUnit;

// Увлекательная математика АЭ14


begin
   // заголовок окна:
   Console.Title := 'Увлекательная математика. Задача АЭ14';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Диофантово число');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу
   var m := 1;
```

```
    var f: integer -> boolean :=  n -> (n * 200).IsQuadrat and
                                       ((n * 200).Sqrt = 5 * n);
    Print(' Число = ');
    m.Step(1)
     .Where(f)
     .Take(1)
     .Println;

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Искомое число равно 8:



Годится также число 0, но оно явно не древнегреческое.

Если не копировать полностью требование задачи, то условие можно записать проще:

```
var f: integer -> boolean :=  n -> 5 * n = (n * 200).Sqrt;
```

Индийские квадраты (Step.Take)

Индийская задача VIII века из *«Бахшалийской» рукописной арифметики*:

Найти число, которое от прибавления 5 или отнятия 11 обращается в полный квадрат.

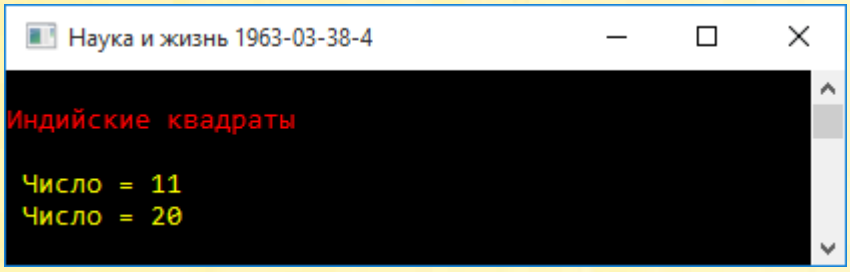
Искомое число не меньше 11. Ищем его в бесконечной последовательности. Условие задачи записывается очень просто:

```
var f: integer -> boolean :=  n -> (n + 5).IsQuadrat and
                                   (n - 11).IsQuadrat;
```

Задача имеет 2 решения, поэтому ставим двойку в вызове метода Take:

```
uses
   System, OlympUnit;

// Наука и жизнь 1963-03-38-4


begin
   // заголовок окна:
   Console.Title := 'Наука и жизнь 1963-03-38-4';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Индийские квадраты');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу
   var m := 11;
   var f: integer -> boolean :=  n -> (n + 5).IsQuadrat and
                                      (n - 11).IsQuadrat;
   Print(' Число = ');
   m.Step(1)
    .Where(f)
    .Take(2)
    .Println;

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Вот 2 числа, удовлетворяющие условиям задачи:



При n = 11 мы получаем квадраты 16 и 0, а при n = 20 – 25 и 9.

Половина и треть (Step.Take)

Задача 489 из книги *Математическая шкатулка* [Нагибин88]:

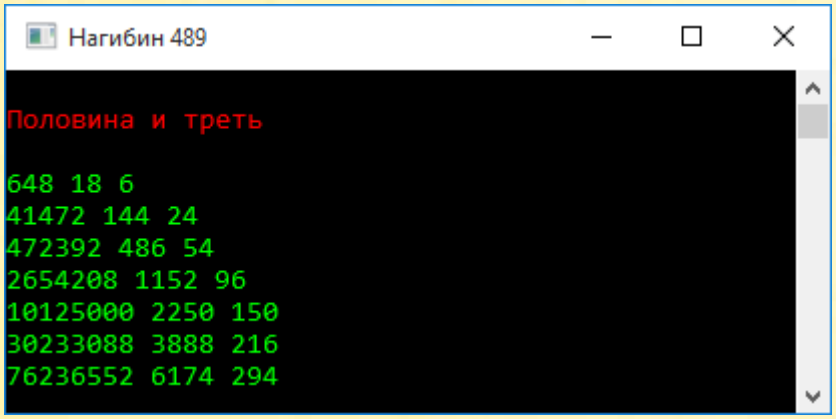
> Запишите наименьшее натуральное число, половина и треть которого были бы соответственно квадратом и кубом некоторых натуральных чисел.

Так как искомое число одновременно делится на *2* и на *3*, то его следует искать среди чисел, кратных 6. Мы не ограничимся поиском единственного числа с означенными свойствами, а найдёт 7 вариантов решения задачи среди чисел до 100 миллионов:

```
uses
   System, OlympUnit;

// Нагибин 489


begin
   // заголовок окна:
   Console.Title := 'Нагибин 489';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
```

```
   Console.WriteLine('Половина и треть');
   Console.ForegroundColor := ConsoleColor.Green;

   var m:= 6;
   var res:= m.Step(6)
             .Where(n-> (n div 2).IsQuadrat and
                        (n div 3).IsCube)
             .Take(7);

   // печатаем ответ:
   Println;
   foreach var n in res do
   begin
      Println(n, Sqrt(n div 2), Power((n div 3), 1.0 / 3.0));
   end;

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Задача – совершенно компьютерная, поэтому он справляется с ней в считанные мгновения. Наименьшее число – 648:



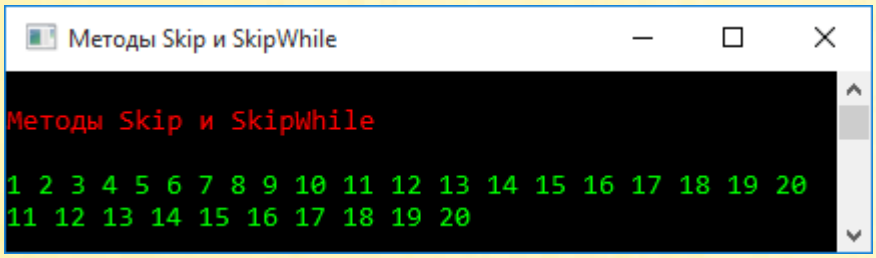
Методы *Skip* и *SkipWhile*

Метод Skip пропускает первые *count* элементов последовательности и возвращает остальные:

```
function Skip(count: integer): sequence of T;
```

Создаём последовательность из 20 элементов, пропускаем первые 10 методом Skip и печатаем оставшиеся:

```
var sq := Range(1,20).Println;
sq.Skip(10).Println;
```



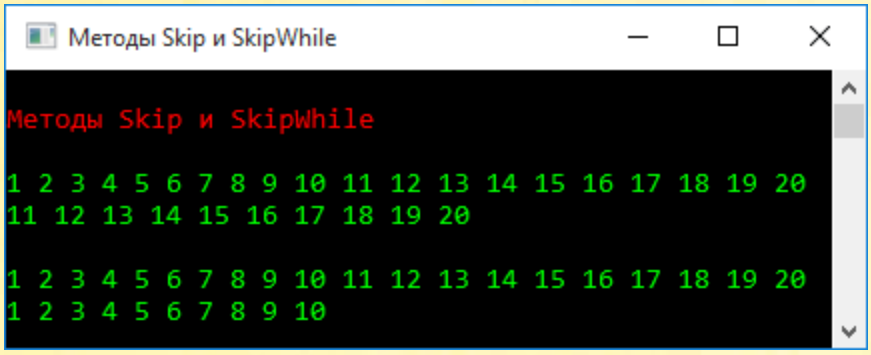
Если значение параметра *count* больше числа элементов в последовательности, то метод Skip вернёт *пустую* последовательность.

Метод SkipLast возвращает последовательность без *последних count* элементов:

```
function SkipLast(count: integer): sequence of T;
```

Создаём такую же последовательность, что и в первом примере, но печатаем последовательность без 10 последних элементов:

```
Println;
sq := Range(1,20).Println;
sq.SkipLast(10).Println;
```

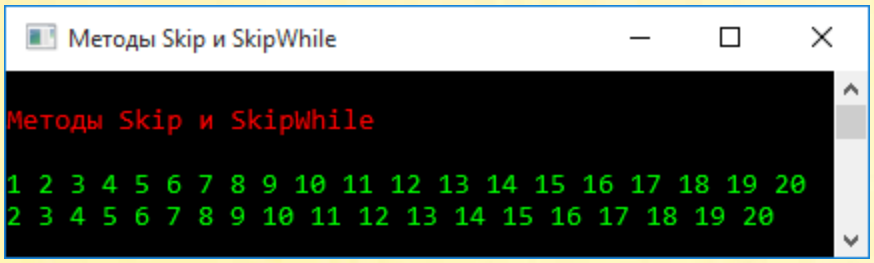


Метод SkipWhile пропускает начальные элементы последовательности, удовлетворяющие заданному условию, а потом возвращает оставшиеся элементы:

```
function SkipWhile(predicate: T->boolean): sequence of T;
```

Пропускаем первые нечётные числа и печатаем все остальные:

```
var sq := Range(1,20).Println;
sq.SkipWhile(n -> n mod 2 <> 0).Println;
```

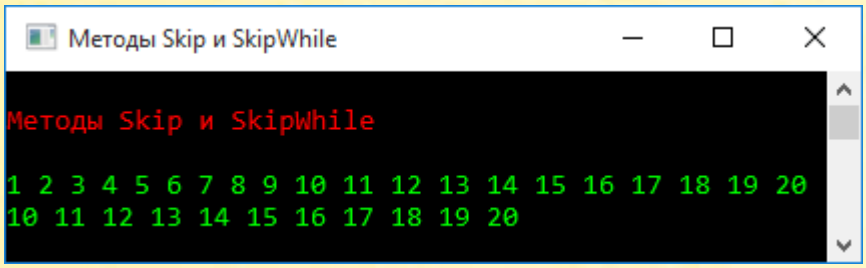


Второй метод SkipWhile пропускает начальные элементы последовательности, удовлетворяющие заданному условию с учётом индексов, а потом возвращает оставшиеся элементы:

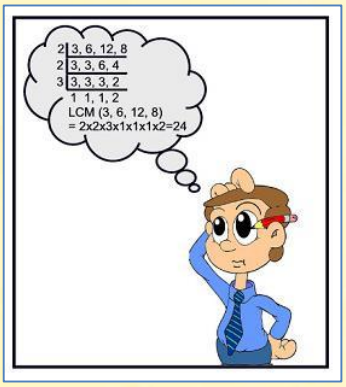
```
function SkipWhile(predicate: (T,integer)->boolean): sequence of T;
```

В той же последовательности натуральных чисел пропускаем такие, которые имеют значение меньше 9 и индекс меньше 12:

```
var sq := Range(1, 20).Println;
sq.SkipWhile((n, i) -> (i < 9) and (n < 12)).Println;
```



Наименьшее число (Step.SkipWhile)



В журнале *Наука и жизнь*, №2 за 1968 год, на странице 57 напечатана такая задача:

НАЙТИ НАИМЕНЬШЕЕ ЧИСЛО

Найдите наименьшее число, которое при делении на 2, 3, 4, 5 и 6 дает остатки соответственно 1, 2, 3, 4 и 5.

Очень простая задача на сообразительность:

НАЙТИ НАИМЕНЬШЕЕ ЧИСЛО

Прибавим к искомому числу 1. Полученная сумма должна делиться одновременно на 2, 3, 4, 5 и 6. Наименьшее такое число 60. Следовательно, искомое число:

60 — 1 = 59.

Мы не знаем диапазона для поиска чисел, но понятно, что искомое число – натуральное, поэтому мы будем искать его в **бесконечной** последовательности.

Бесконечную последовательность натуральных чисел можно получить с помощью метода расширения **Step** для чисел типа *integer*. Начало последовательности задаём в переменной **n**:

```
var n := 1;
```

По умолчанию шаг изменения элементов последовательности равен 1, что нам и нужно.

Распечатать бесконечную последовательность нельзя, но метод *Step* работает верно.

Так как нас интересует число, которое удовлетворяет условиям задачи, то все прочие мы должны пропустить. Для этого предназначен метод расширения **SkipWhile**, которому нужно передать условие задачи тем же способом, что и методу *Where*:

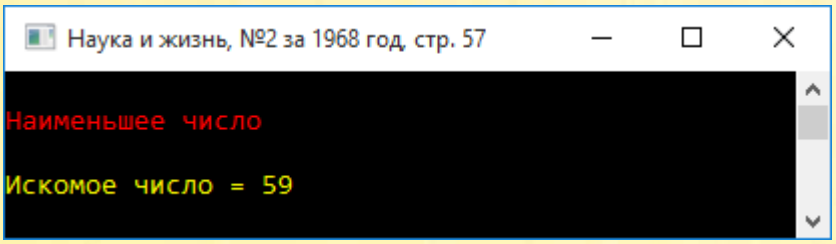
```
.SkipWhile(n -> (n mod 2 <> 1) or
                (n mod 3 <> 2) or
                (n mod 4 <> 3) or
                (n mod 5 <> 4) or
                (n mod 6 <> 5))
```

Метод **SkipWhile** пропускает элементы последовательности, которые не удовлетворяют условию задачи, до тех пор, пока не встретится элемент, который ему удовлетворяет. После этого он будет оставлять все следующие элементы, которых бесконечно много. Но нам нужен только 1 элемент – то число, которое первым подошло под условие задачи. А оно и есть первое в той последовательности, которую мы получили от метода **SkipWhile**. Начало последовательности можно получить от метода расширения **Take**, которому нужно передать число элементов, которые мы желаем получить. Нам нужен только 1 элемент, поэтому в скобках указываем единицу. Метод **Take** останавливает дальнейшее генерирование элементов последовательности.

Задача решена:

```
uses
   System;

// Наука и жизнь, №2 за 1968 год, стр. 57

begin
   // заголовок окна:
   Console.Title := 'Наука и жизнь, №2 за 1968 год, стр. 57';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Наименьшее число');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу
   var n := 1;
   var sq := n.Step
             .SkipWhile(n -> (n mod 2 <> 1) or
                             (n mod 3 <> 2) or
                             (n mod 4 <> 3) or
                             (n mod 5 <> 4) or
                             (n mod 6 <> 5))
             .Take(1)
             .Println;

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Искомое число оказалось очень небольшим:



Легко заметить, что метод расширения Step действует аналогично бесконечному циклу *while*.

Индийские пчёлы (Step.SkipWhile)

А эту задачу придумали в Древней Индии:

Пчёлы числом, равным квадратному корню из половины их числа во всём рое, сели на куст жасмина, 8/9 пчёл полетели назад к рою. И только одна пчела из того же роя кружилась над цветком лотоса, привлечённая жужжанием подруги, неосторожно угодившей в ловушку сладко благоухающего цветка.

Сколько всего пчёл было в рое?

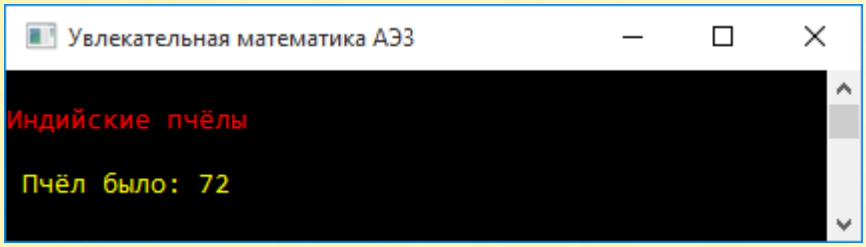
Так число пчёл – **целое**, то в рое должно быть столько пчёл, чтобы их число нацело делилось и на 2 и на 9, то есть на 18.

Как и в предыдущей задаче, создаём бесконечную последовательность и пропускаем все элементы, которые не удовлетворяют условию задачи. Первое подходящее число мы извлекаем методом **Take** и печатаем его на экране:

```
uses
   System;

// Увлекательная математика АЭЗ


begin
   // заголовок окна:
   Console.Title := 'Увлекательная математика АЭЗ';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Индийские пчёлы');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   var b := 0;
   b.Step(18)
```

```
    .SkipWhile(d -> (d div 2).Sqrt + d * 8 div 9 + 2 <> d)
    .Take(1)
    .Println;

  Console.WriteLine();
  Console.ForegroundColor := ConsoleColor.Red;
end.
```

Пчёл в рое было 72:



Индийские обезьяны (Step.SkipWhile)

Очень старая индийская задача про обезьян. Подобные задачи были известны в Индии ещё в XII веке.

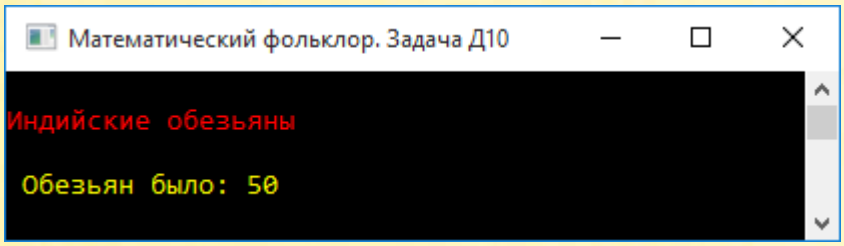
Сколько обезьян в стаде, если квадрат одной пятой их без 3 скрылось в пещере, а одна залезла на дерево?

Задача решается аналогично предыдущим. В методе Step нужно учесть, что число обезьян кратно 5. Начальное значение переменной *m* равно 15, чтобы избежать отрицательных чисел:

```
uses
  System;

//Фольклор Д10
```

```
begin
   //заголовок окна:
   Console.Title := 'Математический фольклор. Задача Д10';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Индийские обезьяны');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу
   var m := 15;
   Print(' Обезьян было: ');
   m.Step(5)
    .SkipWhile(n -> (n div 5 - 3).Sqr + 1 <> n)
    .Take(1)
    .Println;

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

В стаде было 50 обезьян:



Дедушка и внучка (Step.SkipWhile)

Задача 10 (10.1) из книги *Удивительный мир чисел* [КА86], страница 52:

> Сколько дедушке лет, столько месяцев внучке. Дедушке с внучкой вместе 91 год.
>
> Сколько лет дедушке и сколько внучке?

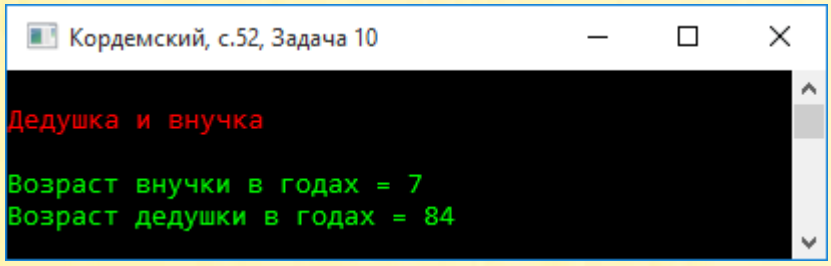
Метод Step генерирует последовательность натуральных чисел, а метод SkipWhile пропускает все числа, не удовлетворяющие условию задачи.

Так как нас интересует первое же подходящее число, то мы вызываем метод First и получаем его в переменной res:

```
uses
   System;

//Кордемский, с.52, Задача 10


begin
   //заголовок окна:
   Console.Title := 'Кордемский, с.52, Задача 10';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Дедушка и внучка');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

    var res:= 0.Step()
              .SkipWhile(x -> x + 12 * x <> 91 * 12)
              .First;
   Console.WriteLine('Возраст внучки в годах = {0}', res div 12);
   Console.WriteLine('Возраст дедушки в годах = {0}',
                                                 91 - res div 12);
```

```
    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Задача решается очень просто:



На одно делится, на другое нет (Step.SkipWhile)

Задача 2 из книги *Удивительный мир чисел* [КА86], страница 84:

> Найдите наименьшее число, которое делится на 77, а при делении на 74 даёт в остатке 48.

Простая переборная задача – как раз для решения на компьютере. Поскольку искомое число кратно 77, то его можно представить в виде:

```
    num = 77*n
```

Значение переменной n нужно изменять от 1 до … - пока задача не будет решена. Здесь вполне уместно использовать метод Step:

```
1.Step()
```

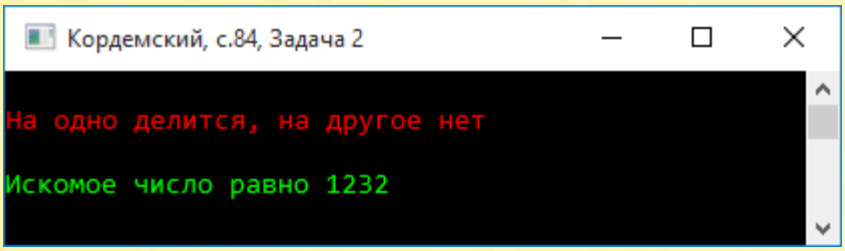
Метод SkipWhile пропускает все числа, не удовлетворяющие условиям задачи:

```
SkipWhile(n -> 77 * n mod 74 <> 48)
```

Метод First прерывает бесконечную последовательность, как только искомое число будет найдено:

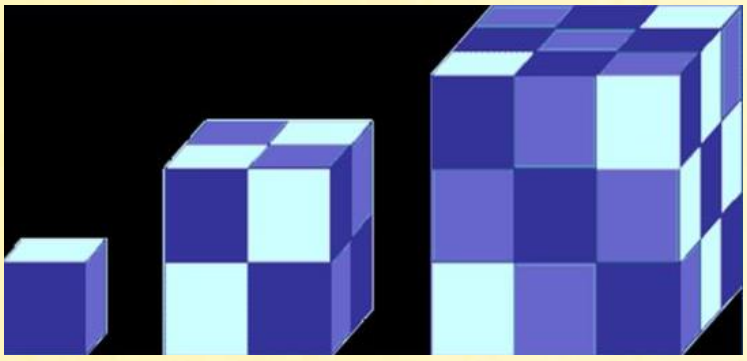
```
uses
   System;

// Кордемский, с.84, Задача 2


begin
    // заголовок окна:
    Console.Title := 'Кордемский, с.84, Задача 2';
    Console.WriteLine('');
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('На одно делится, на другое нет');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    // решаем задачу:
    var num := 1.Step()
                .SkipWhile(n -> 77 * n mod 74 <> 48)
                .First;

    //печатаем искомое число:
    Console.WriteLine('Искомое число равно {0}', num * 77);

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Ответ на задачу показан на рисунке:



Кубическое число (Step.SkipWhile)

Задача 1 из книги *Удивительный мир чисел* [КА86], страница 89:



> Найдите число, куб которого - данное число:
>
> 1) M = 996 703 628 669;
>
> 2) N = 1 011 443 374 872.

Самый простой способ решения – извлечь из заданных чисел кубический корень. Но поскольку кубические корни даже из очень больших чисел невелики, то можно найти их простым перебором, даже начиная с нуля!

Однако методы расширения для последовательностей не умеют работать с числами типа *int64*, поэтому пишем простую функцию cube:

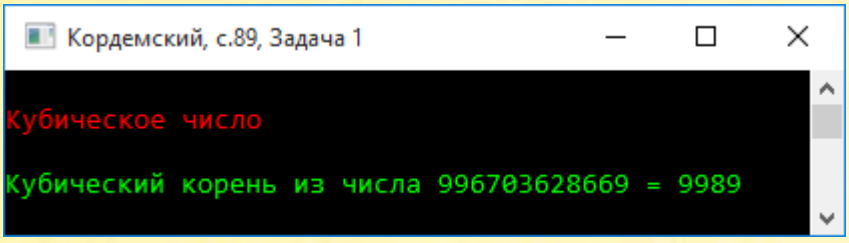
```
// решаем задачу:
var num : int64 := 996703628669;
//var num : int64 := 1011443374872;
var cube: int64 -> boolean:= i -> i * i * i < num;
var res := 0.Step().SkipWhile(i -> cube(i) = true).First;
```

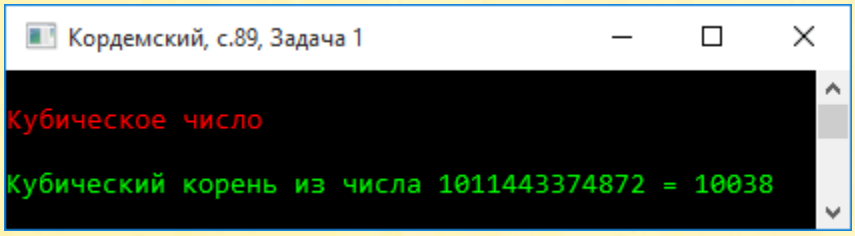
Вот и вся программа:

```
uses
   System;

//Кордемский, с.89, Задача 1


begin
    //заголовок окна:
    Console.Title := 'Кордемский, с.89, Задача 1';
    Console.WriteLine('');
    Console.ForegroundColor := ConsoleColor.Red;
```

```
    Console.WriteLine('Кубическое число');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    // решаем задачу:
    var num : int64 := 996703628669;
    //var num : int64 := 1011443374872;
    var cube: int64 -> boolean:= i -> i * i * i < num;
    var res := 0.Step().SkipWhile(i -> cube(i) = true).First;

    // печатаем ответ:
    Console.WriteLine('Кубический корень из числа {0} = {1}',
                      num, res);

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Точные значения искомых корней:





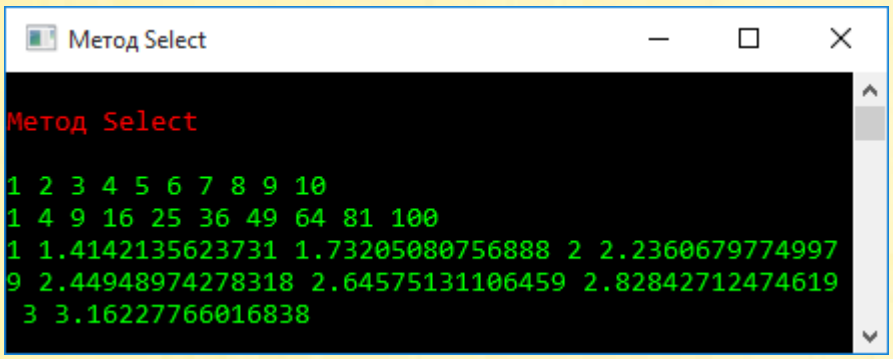
Метод *Select*

Метод Select возвращает последовательность элементов исходной последовательности, к которым была применена функция *selector*:

```
function Select(selector: T -> TRes): sequence of TRes;
```

С помощью этого метода мы легко найдём квадраты и квадратные корни элементов заданной последовательности:

```
var sq := Range(1,10).Println;
sq.Select(n -> n * n).Println;
sq.Select(n -> Sqrt(n)).Println;
```

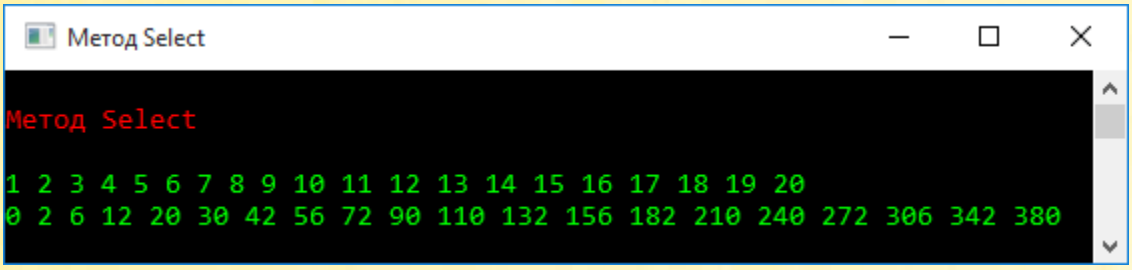


Второй метод Select возвращает последовательность элементов исходной последовательности с учётом *индекса*, к которым была применена функция *selector*:

```
function Select(selector: (T,integer)->TRes): sequence of TRes;
```

Создаём последовательность натуральных чисел и находим произведение каждого элемента с его индексом:

```
var sq := Range(1,20).Println;
sq.Select((n, i) -> n * i).Println;
```



Кузнечики (Range.Select)

В седьмом номере журнала *Квантик* за 2016 год, на странице 32 напечатана вот такая конкурсная задача:

31. Двадцать пять ребят пошли в лес и стали ловить кузнечиков. Несколько ребят поймали по одному кузнечику, половина оставшихся ребят поймали по два кузнечика, а остальные не смогли поймать ни одного. Сколько всего кузнечиков поймали ребята?

Можно решить задачу алгебраически, обозначив буквой *n* число ребят, которые поймали по одному кузнечику. Но на *паскале* задачу лучше решать простым перебором.

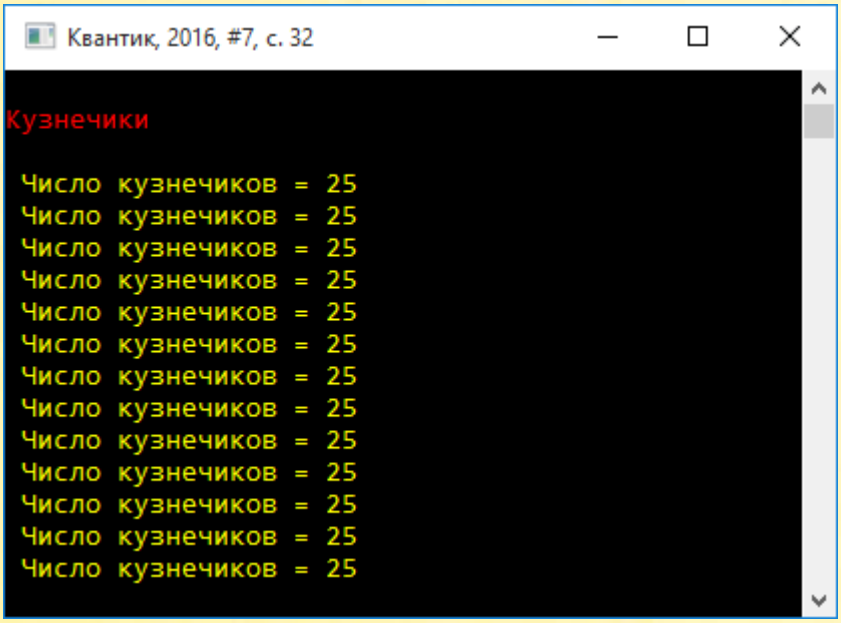
Мы знаем, что *x* не меньше 1 и не больше 25. Причём это число нечётное, иначе число оставшихся ребят также будет нечётным, а от такого числа нельзя взять ровно половину.

В методе Range мы изменяем значение переменной *n* на 2, чтобы оно всегда было нечётным. Число кузнечиков вычисляем в методе Select по формуле:

```
n + (25 – n)  div 2 * 2
```

```
uses
   System;

// Кузнечики


begin
   // заголовок окна:
   Console.Title := 'Квантик, 2016, #7, с. 32';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Кузнечики');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   Range(1,25,2)
   .Select(n -> n + (25 - n) div 2 * 2)
   .Println(NewLine);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Запускаем программу и смотрим на список:



Оказывается, каким бы ни было число ребят, поймавших по 1 кузнечику, **общее число кузнечиков всегда равняется 25**.

А теперь легко догадаться, почему так происходит. Выражение

```
x + (25 – x) / 2 * 2
```

при любом значении ***x*** равно 25:

```
x + (25 – x) = 25
```

Задача пустяковая, а ведь не сразу это поймёшь!

Дробный ряд (Range.Select)

Задача 9-1 из книги *Математическая шкатулка* **[Нагибин88]**, страница 16:

> Найдите простой приём вычислений и воспользуйтесь им для вычисления суммы:
>
> $$\frac{1}{1\cdot2}+\frac{1}{2\cdot3}+\frac{1}{3\cdot4}+\frac{1}{4\cdot5}+\frac{1}{5\cdot6}+\frac{1}{6\cdot7}+\frac{1}{7\cdot8}+\frac{1}{8\cdot9}+\frac{1}{9\cdot10}$$

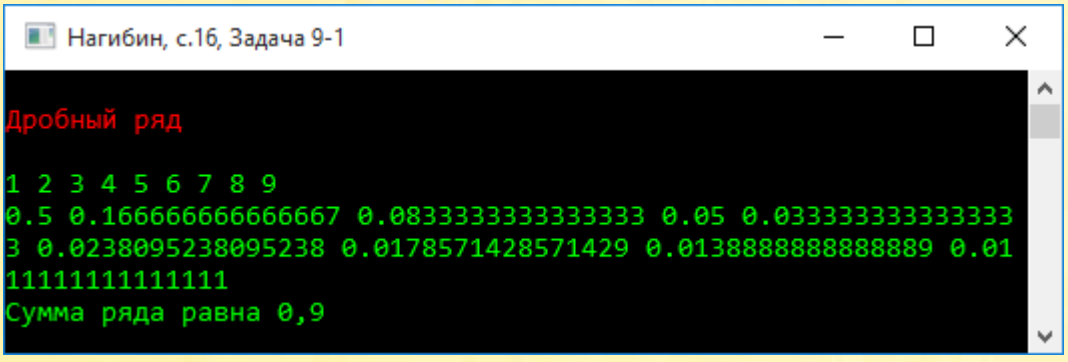
Этот ряд – почти арифметическая прогрессия: в знаменателе первый сомножитель изменяется от 1 до 9 с приращением 1. Второй сомножитель равен первому, плюс 1. Произведение этих сомножителей, а значит и дробей, мы легко найдём в методе **Select**:

```
Range(1,9).Select(n -> 1.0/(n*(n+1)))
```

Осталось вычислить **сумму** дробей:

```
uses
   System;

// Нагибин, с.16, Задача 9-1


begin
   // заголовок окна:
   Console.Title := 'Нагибин, с.16, Задача 9-1';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Дробный ряд');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var summa:= Range(1,9).Println
              .Select(n -> 1.0/(n*(n+1)))
              .Println
              .Sum;

   // печатаем ответ:
   Console.WriteLine('Сумма ряда равна {0}', summa);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Рисунок показывает, что сумма равна 0,9, и подсказывает нам, что у этой задачи имеется и красивое некомпьютерное решение:



Ещё один дробный ряд (Range.Select)

Задача 9-2 из книги *Математическая шкатулка* [Нагибин88], страница 16:

> Найдите простой приём вычислений и воспользуйтесь им для вычисления суммы:
>
> $$\frac{1}{10\cdot 11}+\frac{1}{11\cdot 12}+\frac{1}{12\cdot 13}+\frac{1}{13\cdot 14}+\frac{1}{14\cdot 15}+\ldots+\frac{1}{98\cdot 99}+\frac{1}{99\cdot 100}$$

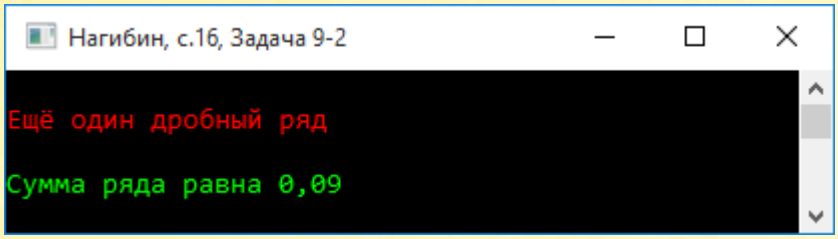
Задача очень похожа на предыдущую, поэтому нужно исправить только аргументы в методе Range:

```
Range(10,99)
```

Вся программа полностью:

```
uses
   System;

// Нагибин, с.16, Задача 9-2


begin
   // заголовок окна:
   Console.Title := 'Нагибин, с.16, Задача 9-2';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Ещё один дробный ряд');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var summa:= Range(10,99).Println
               .Select(n -> 1.0/(n*(n+1)))
               .Println
               .Sum;
```

```
    // печатаем ответ:
    Console.WriteLine('Сумма ряда равна {0}', summa);

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```



Трёхзначное число 2 (Step.Select)

Задача 597 из книги *Математическая шкатулка* [Нагибин88], страницы 97-98:

> Сколько слагаемых суммы 1 + 2 + 3 + 4 + 5 + ... надо взять, чтобы получить трёхзначное число, состоящее из одинаковых цифр?

Числовой ряд представляет собой арифметическую прогрессию, сумму которой мы умеем находить:

```
// сумма арифметической прогрессии:
var summa: integer -> integer := n -> (1 + n) * n div 2;
```

Нужно только не пропустить момент, когда эта сумма превратится в трёхзначное число с одинаковыми цифрами. Искать заданное число можно в бесконечной последовательности, генерируемой методом Step, начиная с единицы:

```
1.Step()
```

В методе Select мы находим сумму *n* первых элементов последовательности:

```
Select(n -> (n,summa(n)))
```

Но нам нужно запомнить и число элементов *n*, поэтому мы создаём кортеж: число элементов – их сумма. В этом кортеже число элементов имеет индекс 0, а сумма – индекс 1.

Метод SkipWhile пропускает все суммы, которые меньше 100:

```
SkipWhile(n -> n[1] < 100)
```

Нам также нужны функции для выделения цифр из трёхзначного числа:

```
// число сотен:
var s: integer -> integer := x -> x div 100;
// число десятков:
var d: integer -> integer := x -> x div 10 mod 10;
// число единиц:
var e: integer -> integer := x -> x mod 10;
```

Метод Where пропускает дальше только суммы с одинаковыми цифрами:

```
Where(n -> (e(n[1]) = d(n[1])) and (e(n[1]) = s(n[1])))
```

Так как интересует первое число, удовлетворяющее условиям задачи, то мы выбираем первый элемент отфильтрованной последовательности:

```
ElementAt(0)
```

Как вы помните, это кортеж из двух элементов. Элемент с индексом 0 – это число слагаемых, а элемент с индексом 1 – сумма:

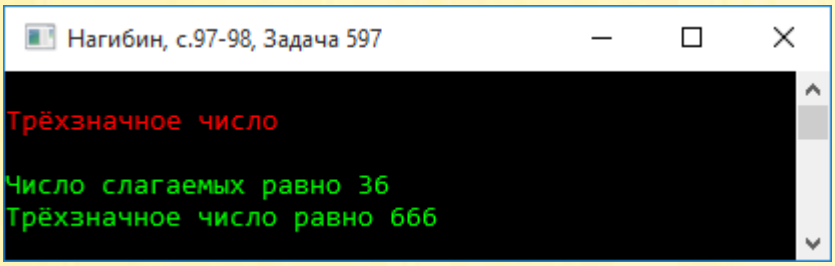
```
// печатаем ответ:
Console.WriteLine('Число слагаемых равно {0}', res[0]);
Console.WriteLine('Трёхзначное число равно {0}', res[1]);
```

Вся программа целиком:

```
uses
   System;

// Нагибин, с.97-98, Задача 597


begin
   // заголовок окна:
   Console.Title := 'Нагибин, с.97-98, Задача 597';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Трёхзначное число');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // число сотен:
   var s: integer -> integer := x -> x div 100;
   // число десятков:
   var d: integer -> integer := x -> x div 10 mod 10;
   // число единиц:
   var e: integer -> integer := x -> x mod 10;
   // сумма арифметической прогрессии:
   var summa: integer -> integer := n -> (1 + n) * n div 2;

  // решаем задачу:
   var res := 1.Step()
              .Select(n -> (n,summa(n)))
              .SkipWhile(n -> n[1] < 100)
              .Where(n -> (e(n[1]) = d(n[1])) and
                          (e(n[1]) = s(n[1])))
              .ElementAt(0);

   // печатаем ответ:
```

```
    Console.WriteLine('Число слагаемых равно {0}', res[0]);
    Console.WriteLine('Трёхзначное число равно {0}', res[1]);

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Задача имеет *единственное* решение:



Наименьшее число (Step.Select)

Найдите наименьшее число, которое начинается с 1993 и делится на все числа от 1 до 9.

Понятно, что задача, скорее, на сообразительность, чем переборная, но именно с помощью перебора её очень просто решить, что мы и сделаем в этом проекте.

Будем считать, что переменная num равна началу числа:

```
var num:= 1993;
```

Легко заметить, что число 19930 на 9 не делится, поэтому сзади будем приставлять числа, начиная с 1.

Если начало числа незыблемо и непоколебимо, то **окончание** числа изменяется от нуля и … пока мы не найдём решения задачи. Эта неопределённость диктует нам применение **бесконечной последовательности** для поиска заданного числа:

```
var res:= 1.Step()
```

Элементы последовательности мы не можем просто приставить в хвост к числу 1993 – сначала его следует умножить на 10, если элемент имеет значение меньше 10, на 100, если 10..99, и так далее.

Показатель степени равен *Trunc(Log10(n)) + 1*, поэтому в методе **Select** мы легко найдём очередное число, которое начинается с 1993:

```
Select(n -> Trunc(num * Power(10, Trunc(Log10(n)) + 1) + n))
```

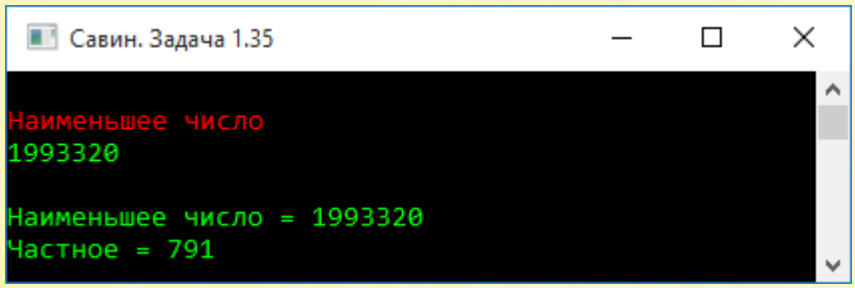
Чтобы проверить его делимость на ряд чисел 1..9, достаточно привлечь к испытаниям только числа 5, 7, 8 и 9, остальные удовлетворятся автоматически:

```
.Where(n -> n mod 5 = 0)
.Where(n -> n mod 7 = 0)
.Where(n -> n mod 8 = 0)
.Where(n -> n mod 9 = 0)
```

Как только мы обнаружим первое число, удовлетворяющее условию задачи, сразу печатаем **ответ** на экране и завершаем бесконечную последовательность:

```
.Take(1)
.Println;
```

Наименьшее число найдено:

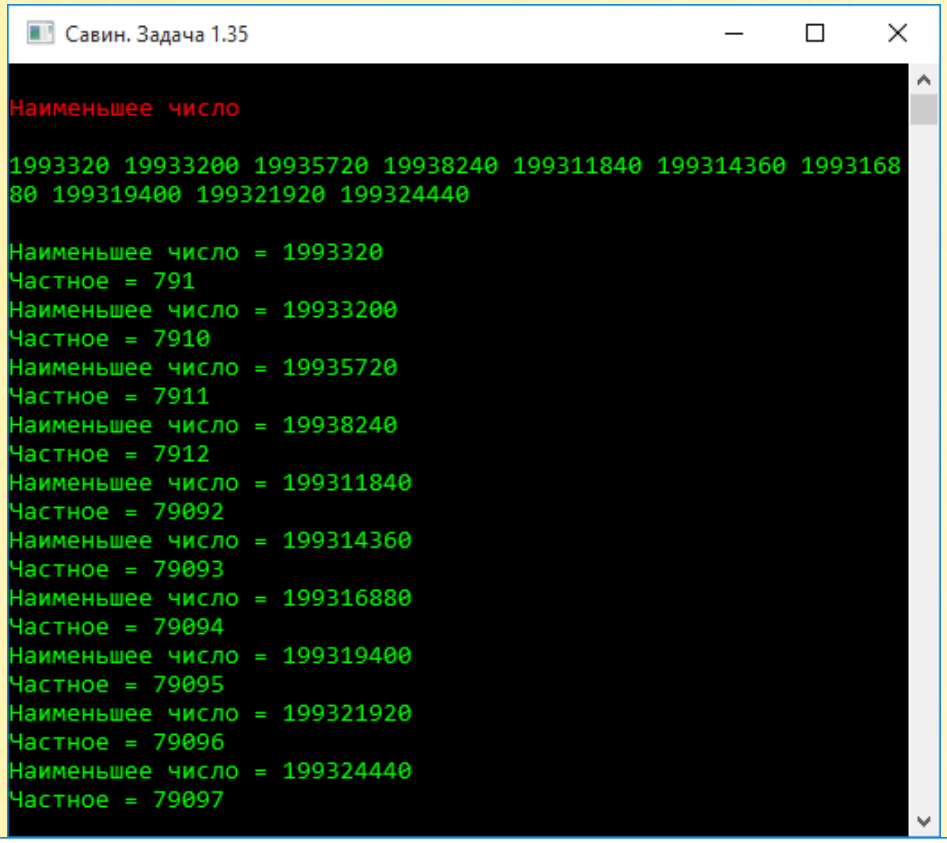


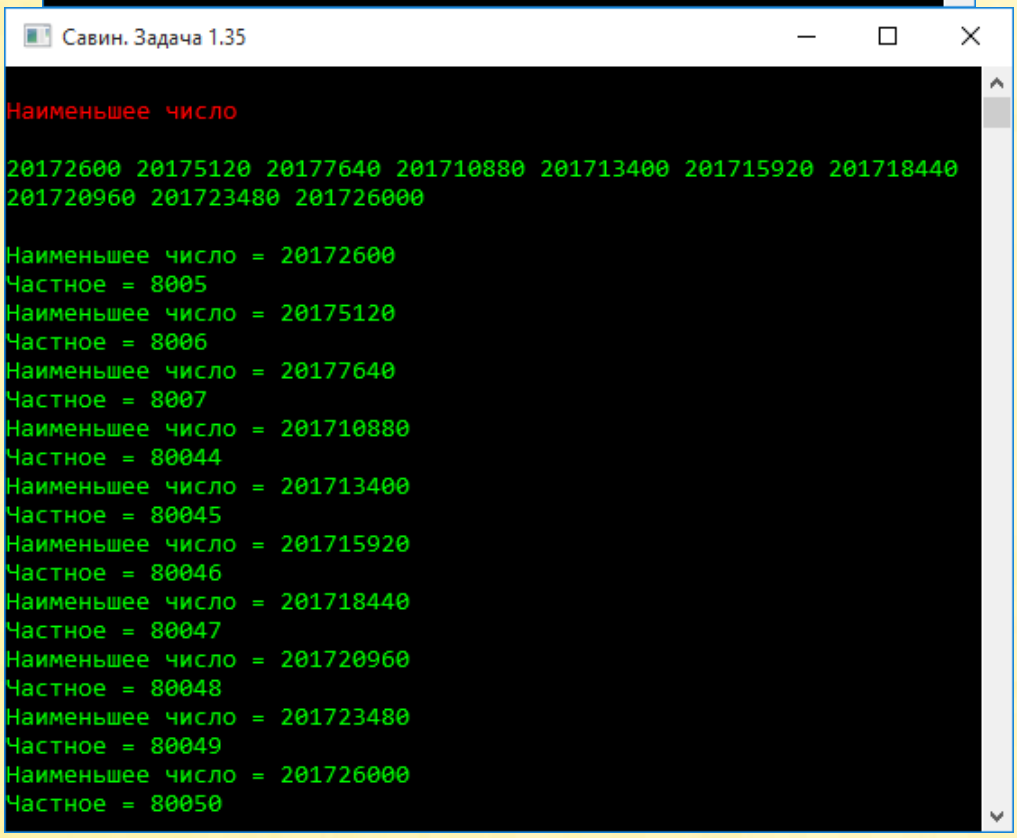
Вот вся программа целиком:

```
uses
   System;

// Савин 1-35


begin
   // заголовок окна:
   Console.Title := 'Савин. Задача 1.35';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Наименьшее число');
   Console.ForegroundColor := ConsoleColor.Green;

   var num:= 1993;
   var res:= 1.Step()
             .Select(n -> Trunc(num * Power(10,
                         Trunc(Log10(n)) + 1) + n))
             .Where(n -> n mod 5 = 0)
             .Where(n -> n mod 7 = 0)
             .Where(n -> n mod 8 = 0)
             .Where(n -> n mod 9 = 0)
             .Take(1)
             .Println;

    Println;
    Console.WriteLine('Наименьшее число = ' + res.First);
    Console.WriteLine('Частное = ' + res.First div 5 div 7
                                                 div 8 div 9);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

При известном любопытстве можно найти ещё несколько решений этой задачи, если не требовать, чтобы числа были наименьшими из возможных (рис. сверху).

Чем хороши компьютерные программы, так это тем, что с их помощью можно запросто перерешать множество подобных задач. Например, исправив значение единственной переменной num, мы тут же получим ответ на задачу для *2017* года (рис. снизу).


Савин. Задача 1.35
Наименьшее число
1993320 19933200 19935720 19938240 199311840 199314360 1993168
80 199319400 199321920 199324440
Наименьшее число = 1993320
Частное = 791
Наименьшее число = 19933200
Частное = 7910
Наименьшее число = 19935720
Частное = 7911
Наименьшее число = 19938240
Частное = 7912
Наименьшее число = 199311840
Частное = 79092
Наименьшее число = 199314360
Частное = 79093
Наименьшее число = 199316880
Частное = 79094
Наименьшее число = 199319400
Частное = 79095
Наименьшее число = 199321920
Частное = 79096
Наименьшее число = 199324440
Частное = 79097


Савин. Задача 1.35
Наименьшее число
20172600 20175120 20177640 201710880 201713400 201715920 201718440
201720960 201723480 201726000
Наименьшее число = 20172600
Частное = 8005
Наименьшее число = 20175120
Частное = 8006
Наименьшее число = 20177640
Частное = 8007
Наименьшее число = 201710880
Частное = 80044
Наименьшее число = 201713400
Частное = 80045
Наименьшее число = 201715920
Частное = 80046
Наименьшее число = 201718440
Частное = 80047
Наименьшее число = 201720960
Частное = 80048
Наименьшее число = 201723480
Частное = 80049
Наименьшее число = 201726000
Частное = 80050

Enigma 1436: One more step Кубы (Select)

В журнале *New Scientist #2597* от 31-го марта 2007 года была напечатана задача Enigma 1436:

I invite you to consider the following series of numbers: 1, 3, 7, 13, 21 …

Then find the following:

(a) The six-hundredth member of the series.
(b) A member of the series above the first with less than five digits which is a perfect cube.
(c) A member which is a five-digit palindrome which can also be read as a binary number.
(d) The smaller of the two consecutive members which are 1000 apart.

Рассмотрим следующий ряд чисел: 1, 3, 7, 13, 21 …

Найдите:

(a) Шестисотый член ряда.
(b) Член ряда, который:
больше первого,
состоит менее чем из пяти цифр и
является кубом целого числа.
(c) Член ряда, который:
является палиндромом с пятью цифрами,
может быть прочитан как двоичное число.
(d) Два наименьших последовательных члена ряда, разность между которыми равняется 1000.

Прежде чем решать задачу, мы должны научиться вычислять члены ряда.

Обозначим ряд буквой a, тогда первый член равняется:

$$a_1 = 1$$

Обозначим разность между членами ряда буквой d, тогда:

$d_1 = 3-1 = 2$
$d_2 = 7-3 = 4$
$d_3 = 13-7 = 6$
$d_4 = 21-13 = 8$
. . .
$d_n = 2 * n,$

где n – номер члена ряда.

Теперь мы легко найдём любой член ряда:

$$a_n = a_{n-1} + d_n$$

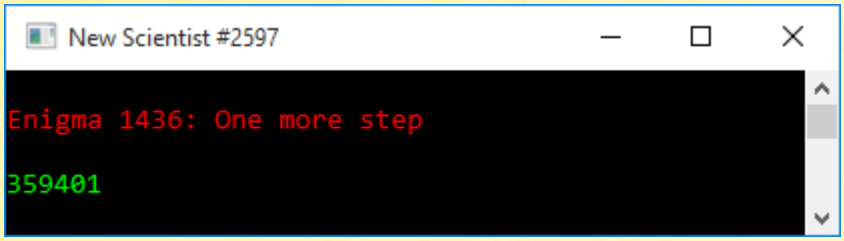
Для вывода **нерекуррентной** формулы воспользуемся *методом исчисления конечных разностей*, который описан в книге Мартина Гарднера **[ГМ72]**, на страницах 81-95, а также в книге *Как решать комбинаторные задачи на компьютере*:

```
1 3 7 13 21
 2 4 6  8
  2 2 2

Откуда:

an = n^2 - n + 1
(1)
```

Переходим к решению задачи-

Подзадачу (a) мы решаем, просто генерируя 600 элементов заданной последовательности:

```
var sq := SeqGen(601, i -> i*i - i + 1);//.Println;
var a := sq.Last;
Println(a);
```



Для решения подзадачи (b) нам потребуется функция IsCube из модуля *OlympUnit*, которая умеет определять, является ли заданное целое число точным кубом. Все условия подзадачи (b) легко записать в методе Where:

```
var b := sq.Skip(2)
          .Where(i -> (i.NumDigit < 5) and (i.IsCube))
          .Println;
```

Здесь:

- Номер n искомого члена ряда больше единицы:

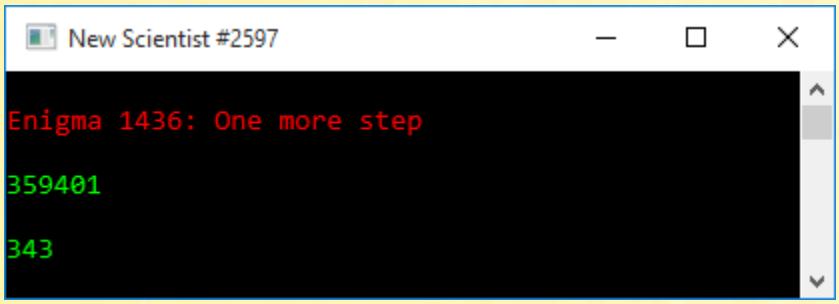
```
Skip(2)
```

- Искомый член a ряда состоит менее чем из пяти цифр:

```
i.NumDigit < 5
```

- Искомый член ряда a – точный куб:

```
i.IsCube
```



Подзадачу (c) мы решаем аналогично, но условие прекращения цикла должно включать проверки, что очередной член ряда:

- пятизначное число
- палиндром
- состоит только из цифр 0 и 1:

```
var c := sq.Where(i -> (i.NumDigit = 5) and (i.IsPalindrome));
var c2 := c.Select(i -> new string(i.ToString.ToCharArray))
           .Where(s -> s.IndexOfAny(new char[]
                 ('2', '3', '4', '5', '6', '7', '8', '9' )) = -1)
           .Println;
```

Здесь:

- Искомый член a ряда состоит из пяти цифр:

```
i.NumDigit = 5
```

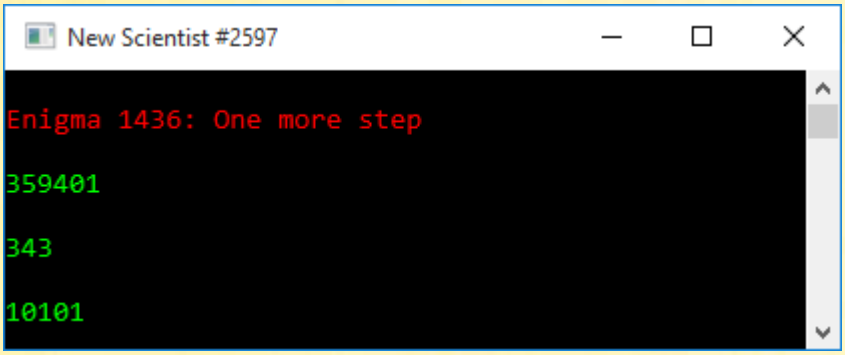
- Искомый член ряда a – палиндром:
- Искомый член ряда состоит из цифр 0 и 1:

```
i.IsPalindrome

var c2 := c.Select(i -> new string(i.ToString.ToCharArray))
          .Where(s -> s.IndexOfAny(new char[]
                 ('2', '3', '4', '5', '6', '7', '8', '9' )) = -1)
```

С помощью метода IndexOfAny мы легко определим, имеются ли в заданном числе (точнее – в строке, полученной из этого числа) «запрещённые» цифры.

На самом деле многие проверки в данном случае излишни; тот же самый ответ мы получили бы и с гораздо меньшими усилиями.

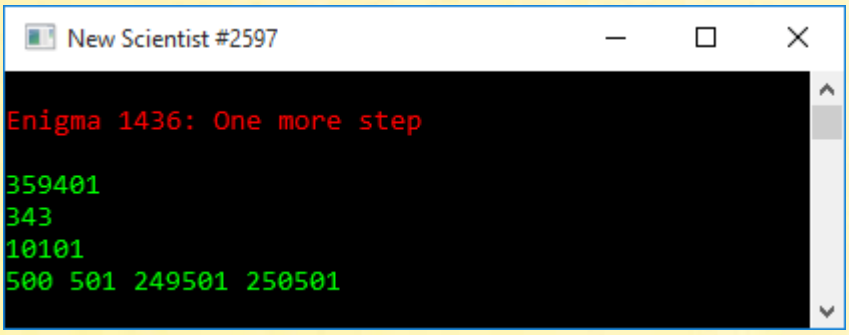
Все условия соблюдены, запускаем программу – и находим решение подзадачи (c):



И последнюю подзадачу мы решаем без труда:

```
for var i := 1 to sq.Count-2 do
begin
   if (sq.ElementAt(i+1) - sq.ElementAt(i) = 1000) then
   begin
      Println(i, i + 1, sq.ElementAt(i), sq.ElementAt(i + 1));
   end;
end;
```

Если мы начнём с первого члена ряда (*n=1*), то нам нужно найти разность между ним и *следующим* членом ряда, номер которого на единицу больше. Если эта разность не равняется 1000, то переходим ко второму члену ряда и находим разность между ним и третьим членом ряда. Продолжаем так до тех пор, пока не получим заданную разность:



Метод *SelectMany*

Метод SelectMany возвращает последовательность, составленную из отдельных подпоследовательностей, возвращаемых функцией *selector*:

```
function SelectMany<Res>(selector: T->sequence of Res): sequence of Res;
```

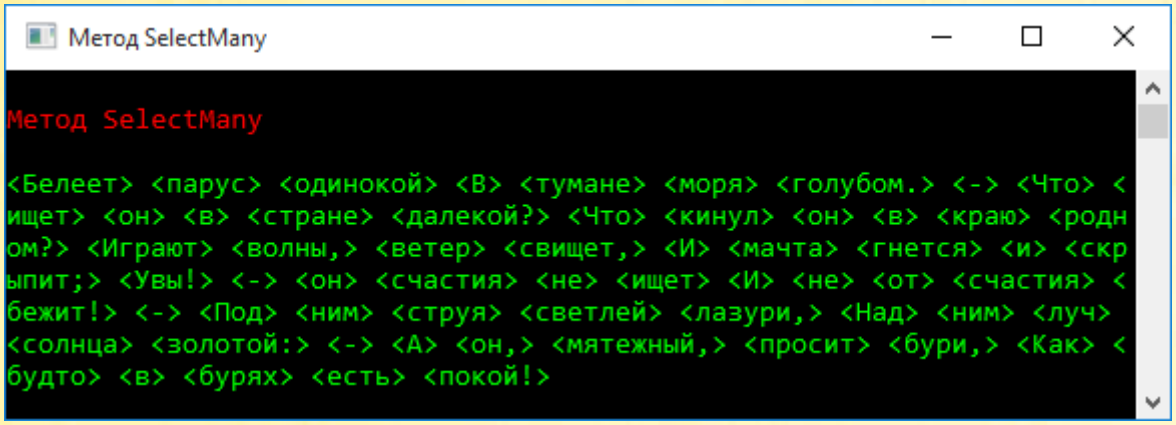
Пусть у нас имеется стихотворение Лермонтова *Парус*, представленное в виде массива строк text:

```
var text : array of string :=
                ('Белеет парус одинокой',
                 'В тумане моря голубом.',
                 '– Что ищет он в стране далекой?',
                 'Что кинул он в краю родном?',
                 'Играют волны, ветер свищет,',
                 'И мачта гнется и скрыпит;',
                 'Увы! – он счастия не ищет',
                 'И не от счастия бежит! –',
                 'Под ним струя светлей лазури,',
                 'Над ним луч солнца золотой: –',
                 'А он, мятежный, просит бури,',
                 'Как будто в бурях есть покой!');
```

Мы хотим получить последовательность всех слов в этом стихотворении.

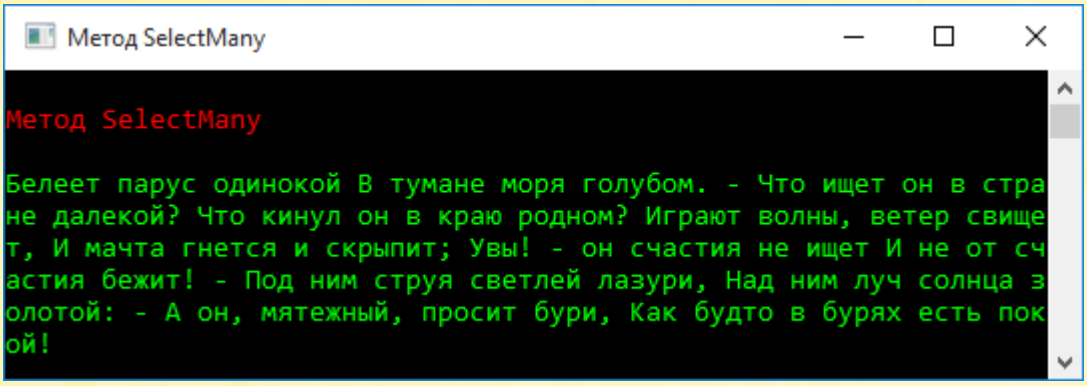
Метод SelectMany последовательно получает строки массива *text*- Функция *selector* разбивает их на отдельные слова обычным способом – методом *Split* класса *String* - и возвращает последовательность отдельных слов строки. Все эти последовательности затем объединяются в единую последовательность слов, которую и возвращает метод *SelectMany*:

```
var words := text
            .SelectMany(s -> s.Split().Select(s2 -> '<' + s2 + '>'));
```



Метод Select нужен только для выделения отдельных слов. Можно обойтись и без него:

```
var words := text
             .SelectMany(s -> s.Split());
```



Вся программа целиком:

```
uses System;

begin
    //заголовок окна:
    Console.Title := 'Метод SelectMany';
    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Метод SelectMany');
    Console.ForegroundColor := ConsoleColor.Green;
    Println;
```

```
    var text : array of string :=
                ('Белеет парус одинокой',
                 'В тумане моря голубом.',
                 '– Что ищет он в стране далекой?',
                 'Что кинул он в краю родном?',
                 'Играют волны, ветер свищет,',
                 'И мачта гнется и скрыпит;',
                 'Увы! – он счастия не ищет',
                 'И не от счастия бежит! –',
                 'Под ним струя светлей лазури,',
                 'Над ним луч солнца золотой: –',
                 'А он, мятежный, просит бури,',
                 'Как будто в бурях есть покой!');

    var words := text
                .SelectMany(s -> s.Split()
                                  .Select(s2 -> '<' + s2 + '>'));
    words.Println;

    Println;
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Во **втором методе SelectMany** функция *selector* дополнительно получает *индекс* каждого элемента исходной последовательности:

```
function SelectMany<Res>(selector: (T,integer)->sequence of Res):
                         sequence of Res;
```

В **третьем методе SelectMany** дополнительно вызывается функция *resultSelector* для каждого элемента выходной последовательности:

```
function SelectMany<Coll,Res>(collSelector:
                              (T,integer)->sequence of Coll;
                               resultSelector: (T,Coll)->Res):
                               sequence of Res;
```

В **четвёртом** методе SelectMany функция *collectionSelector* дополнительно получает *индекс* каждого элемента исходной последовательности:

```
function SelectMany<Coll,Res>(collSelector: T->sequence of Coll;
                              resultSelector: (T,Coll)->Res):
                              sequence of Res;
```

Методы последовательностей хорошо заменяют обычные циклы, но **вложенные циклы** даются им с трудом. В следующем проекте мы привлечём к этому метод SelectMany.

Трёхзначное число 2 (Range.SelectMany)

Задача 585 из книги *Математическая шкатулка* [Нагибин88]:

> Если первую цифру трёхзначного числа увеличить на *n*, а вторую и третью цифру уменьшить на *n*, то получится число в *n* раз больше исходного.
>
> Найдите *n* и исходное число.

В первую очередь нам нужны **функции** для извлечения цифр из трёхзначного числа:

```
// первая цифра:
var n1: integer -> integer := x -> x div 100;
// вторая цифра:
var n2: integer -> integer := x -> x div 10 mod 10;
// третья цифра:
var n3: integer -> integer := x -> x mod 10;
```

Тут всё просто.

Ещё проще получить **последовательность трёхзначных чисел**:

```
Range(100,999)
```

Трёхзначное – это **первое** из искомых чисел. Но есть ещё и **второе**. В задаче оно обозначено буквой ***n***. Оно однозначное.

Вторую последовательность получить сложнее, чем первую. Здесь нам необходим метод **SelectMany**. Он позволяет к каждому числу первой последовательности добавить число из второй последовательности:

```
Range(1,9)
```

В итоге мы получаем все **пары** чисел: трёхзначное число – однозначное число. Но они нужны нам не сами по себе, а как аргументы в функциях *n1, n2, n3*. Мы могли бы все вычисления провести и в методе *SelectMany*, но тогда нужные нам числа не удалось бы распечатать. Поэтому метод *SelectMany* будет создавать последовательность из пар чисел. Проще всего сохранить пары чисел в **кортеже**:

```
Range(100,999).SelectMany(y -> Range(1,9), (x,y) -> (x,y)).Println;
```

Мы распечатали последовательность кортежей, и на рисунке хорошо видны все пары чисел от (100,1) до (999,9) (рис. на следующей странице).

Полдела сделано!

В каждой паре первое (трёхзначное) число имеет индекс **0**, а второе (однозначное) – индекс **1**. Теперь в методе **Where** мы можем выбрать такую пару чисел (или несколько пар), которые удовлетворяют условиям задачи:

```
Where(n -> (n1(n[0]) + n[1]) * 100 +
           (n2(n[0]) - n[1]) * 10 +
           (n3(n[0]) - n[1]) = n[0] * n[1])
```

(975,4) (975,5) (975,6) (975,7) (975,8) (975,9) (976,1) (976,2) (976,3) (976,4
) (976,5) (976,6) (976,7) (976,8) (976,9) (977,1) (977,2) (977,3) (977,4) (977
,5) (977,6) (977,7) (977,8) (977,9) (978,1) (978,2) (978,3) (978,4) (978,5) (9
78,6) (978,7) (978,8) (978,9) (979,1) (979,2) (979,3) (979,4) (979,5) (979,6)
(979,7) (979,8) (979,9) (980,1) (980,2) (980,3) (980,4) (980,5) (980,6) (980,7
) (980,8) (980,9) (981,1) (981,2) (981,3) (981,4) (981,5) (981,6) (981,7) (981
,8) (981,9) (982,1) (982,2) (982,3) (982,4) (982,5) (982,6) (982,7) (982,8) (9
82,9) (983,1) (983,2) (983,3) (983,4) (983,5) (983,6) (983,7) (983,8) (983,9)
(984,1) (984,2) (984,3) (984,4) (984,5) (984,6) (984,7) (984,8) (984,9) (985,1
) (985,2) (985,3) (985,4) (985,5) (985,6) (985,7) (985,8) (985,9) (986,1) (986
,2) (986,3) (986,4) (986,5) (986,6) (986,7) (986,8) (986,9) (987,1) (987,2) (9
87,3) (987,4) (987,5) (987,6) (987,7) (987,8) (987,9) (988,1) (988,2) (988,3)
(988,4) (988,5) (988,6) (988,7) (988,8) (988,9) (989,1) (989,2) (989,3) (989,4
) (989,5) (989,6) (989,7) (989,8) (989,9) (990,1) (990,2) (990,3) (990,4) (990
,5) (990,6) (990,7) (990,8) (990,9) (991,1) (991,2) (991,3) (991,4) (991,5) (9
91,6) (991,7) (991,8) (991,9) (992,1) (992,2) (992,3) (992,4) (992,5) (992,6)
(992,7) (992,8) (992,9) (993,1) (993,2) (993,3) (993,4) (993,5) (993,6) (993,7
) (993,8) (993,9) (994,1) (994,2) (994,3) (994,4) (994,5) (994,6) (994,7) (994
,8) (994,9) (995,1) (995,2) (995,3) (995,4) (995,5) (995,6) (995,7) (995,8) (9
95,9) (996,1) (996,2) (996,3) (996,4) (996,5) (996,6) (996,7) (996,8) (996,9)
(997,1) (997,2) (997,3) (997,4) (997,5) (997,6) (997,7) (997,8) (997,9) (998,1
) (998,2) (998,3) (998,4) (998,5) (998,6) (998,7) (998,8) (998,9) (999,1) (999
,2) (999,3) (999,4) (999,5) (999,6) (999,7) (999,8) (999,9)

Распечатав отфильтрованную последовательность, мы убеждаемся, что задача имеет *единственное* решение. Поскольку нам нужно только оно, то мы берём первый и единственный элемент последовательности:

```
ElementAt(0)
```

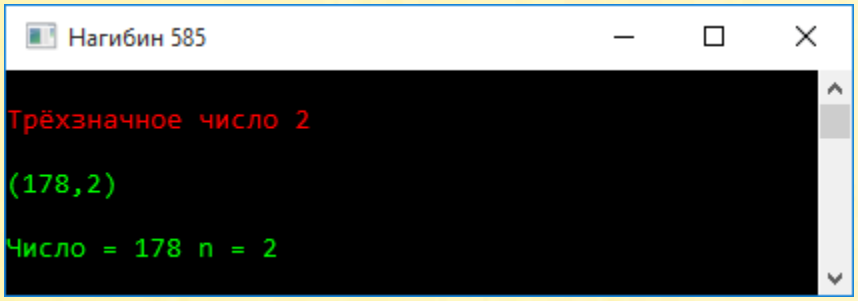
Так как это кортеж, то при печати ответа мы извлекаем из него оба числа по их индексам.

Задача решена:

```
uses
   System;

// Нагибин 585


begin
   // заголовок окна:
   Console.Title := 'Нагибин 585';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Трёхзначное число 2');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу -->

   // первая цифра:
   var n1: integer -> integer := x -> x div 100;
   // вторая цифра:
   var n2: integer -> integer := x -> x div 10 mod 10;
   // третья цифра:
   var n3: integer -> integer := x -> x mod 10;

   var res:= Range(100,999)
            .SelectMany(y -> Range(1,9), (x,y) -> (x,y))//.Println
            .Where(n -> (n1(n[0]) + n[1]) * 100 +
                        (n2(n[0]) - n[1]) * 10 +
                        (n3(n[0]) - n[1]) = n[0] * n[1])
            .Println
```

```
            .ElementAt(0);

    // печатаем ответ:
    Console.WriteLine();
    Console.WriteLine('Число = {0} n = {1}', res[0], res[1]);

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

А вот и **ответ** на эту замысловатую задачу:



Метод *Zip*

Метод Zip возвращает последовательность, объединяющую 2 заданные последовательности. К каждой паре элементов обеих последовательностей с одинаковыми индексами применяется функция *resultSelector*:

```
function Zip<TSecond,Res>(second: sequence of TSecond;
                          resultSelector: (T,TSecond)->Res):
                          sequence of Res;
```

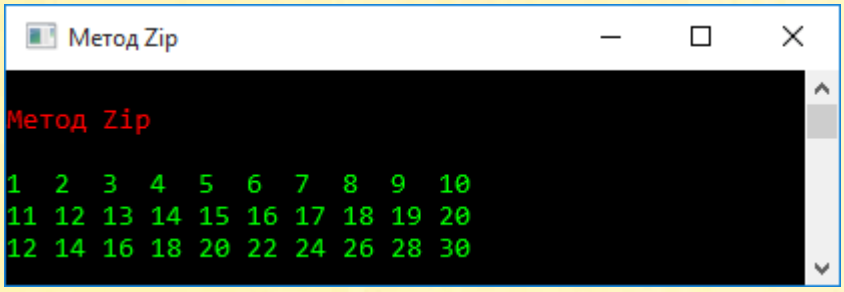
Если одна последовательность короче другой, то в результирующей последовательности будет столько же элементов, сколько их имеется в более короткой последовательности, так как для остальных элементов более длинной последовательности не найдётся пары в короткой последовательности.

Создаём 2 последовательности по 10 натуральных чисел:

```
var sq1 := Range(1,10).Println('  ');
var sq2 := Range(11,20).Println;
```

И в методе Zip попарно складываем их:

```
sq1.Zip(sq2, (x,y) -> x + y).Println;
```



Плюс-минус (Range.Zip)

Задача 7-2 из книги *Математическая шкатулка* [Нагибин88], страница 15:

> Как быстро вычислить:
>
> 99 - 97 + 95 - 93 + 91 - 89 + ... + 7 - 5 + 3 – 1?

Конечно, быстро вычислить можно на компьютере!

По сравнению с обычной арифметической прогрессией (здесь она убывающая) знак членов ряда чередуется: *плюс-минус-плюс-минус...*

Последовательность легко получить от метода Range:

```
Range(99, 1, -2)
```

Здесь мы учитываем, что абсолютная величина элементов последовательности уменьшается на 2.

Более сложно решить проблему с чередующимися знаками.

Заметим, что все полученные числа последовательности нужно умножать попеременно на числа 1 и -1. Метод Range возвращает числа от 1 до 50. Нам нужно, чтобы нечётные числа равнялись 1, а чётные - -1. В методе Select мы без труда справляемся с этой задачей.

```
Range(1,50).Select(x -> x.IsOdd ? 1 : -1)
```

Мы получили 2 последовательности:

```
var seq1 := Range(99, 1, -2);
var seq2 := Range(1,50).Select(x -> x.IsOdd ? 1 : -1);
```

В первой хранятся числа, а во второй – знаки. Чтобы получить заданную последовательность, нужно эти последовательности поэлементно перемножить. А это умеет делать метод Zip:

```
seq1.Zip(seq2, (x,y)->x*y)
```

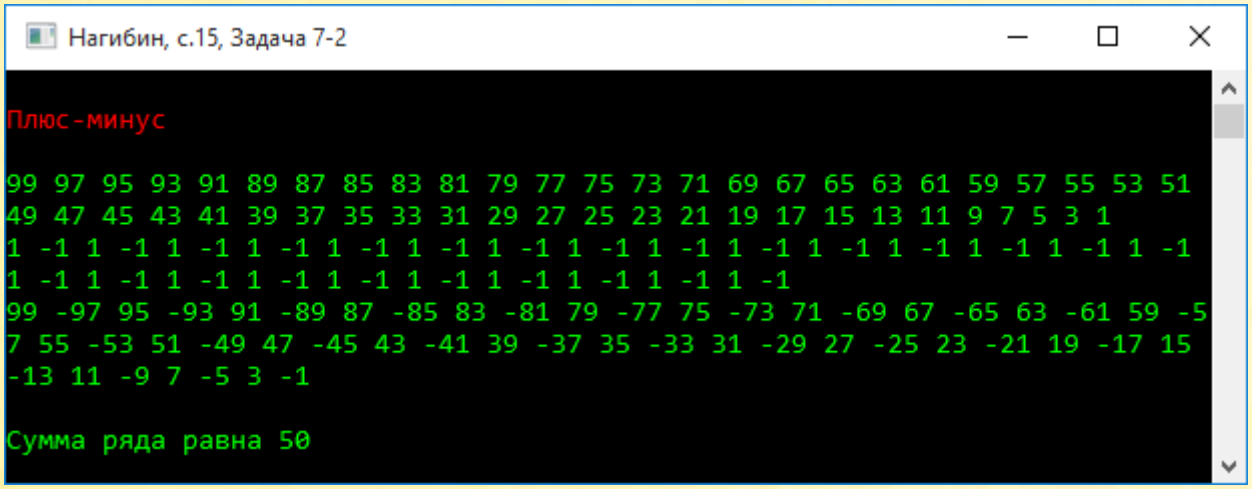
Он получает по одному элементу из каждой последовательности и перемножает их.

Осталось найти сумму третьей последовательности – и задача решена:

```
uses
   System;

// Нагибин, с.15, Задача 7-2


begin
```

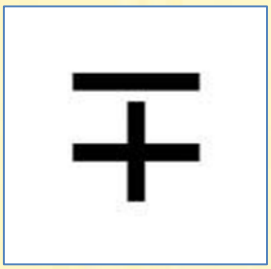
```
    // заголовок окна:
    Console.Title := 'Нагибин, с.15, Задача 7-2';
    Console.WriteLine('');
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Плюс-минус');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    // решаем задачу:
    var seq1 := Range(99, 1, -2).Println;
    var seq2 := Range(1,50).Select(x -> x.IsOdd ? 1 : -1).Println;
    var summa := seq1.Zip(seq2, (x,y)->x*y).Println.Sum;

    // печатаем ответ:
    Console.WriteLine();
    Console.WriteLine('Сумма ряда равна {0}', summa);

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Сумма ряда найдена:



Минус-плюс (Range.Zip)

Задача 482-8 из книги *Математическая шкатулка* [Нагибин88], страница 88:



> Найдите простой приём вычисления:
>
> $100^2 - 99^2 + 98^2 - 97^2 + 96^2 - 95^2 + \ldots + 4^2 - 3^2 + 2^2 - 1^2$

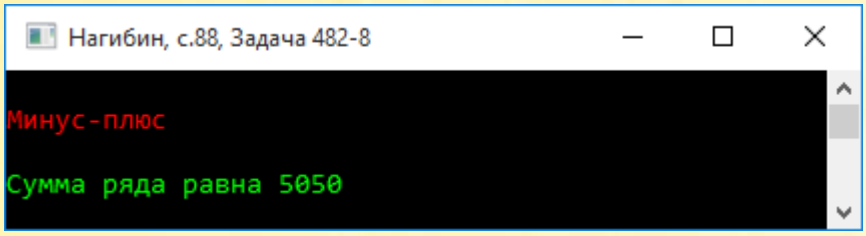
Здесь мы видим практически тот же ряд, что и в предыдущей задаче, только все числа возводятся в квадрат, что легко сделать в методе Select:

```
var seq1 := Range(100, 1, -1).Select(x -> x*x);
```

Дальше задача решается без труда, но довольно интересно, что ответ на неё полностью совпадает с тем ответом, который мы получили, решая задачу Гаусса:

```
uses
   System;

// Нагибин, с.88, Задача 482-8


begin
   // заголовок окна:
   Console.Title := 'Нагибин, с.88, Задача 482-8';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Минус-плюс');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var seq1 := Range(100, 1, -1).Select(x -> x*x);
   var seq2 := Range(1,100).Select(x -> x.IsOdd ? 1 : -1);
   var summa := seq1.Zip(seq2, (x,y)->x*y).Sum;

   // печатаем ответ:
```

```
   Console.WriteLine('Сумма ряда равна {0}', summa);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```



Метод *Distinct*

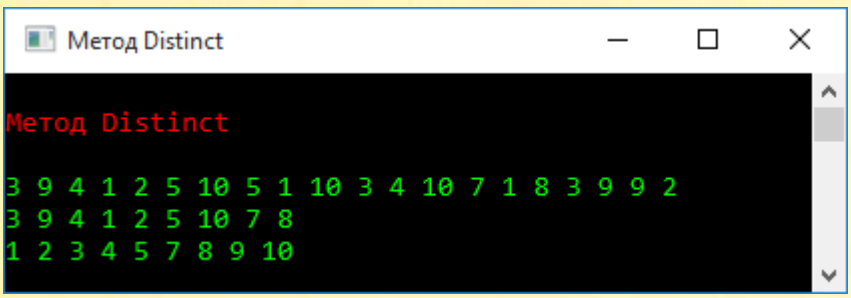
Метод Distinct возвращает исходную последовательность, из которой удалены *повторяющиеся* элементы (то есть все оставшиеся элементы - *уникальны*):

```
function Distinct(): sequence of T;
```

Создаём последовательность целых чисел, а затем напечатаем её без повторов элементов:

```
var sq:= SeqRandom(20,1,10).ToArray.Println;
sq.Distinct().Println;
```

Можно дополнительно *отсортировать* элементы:

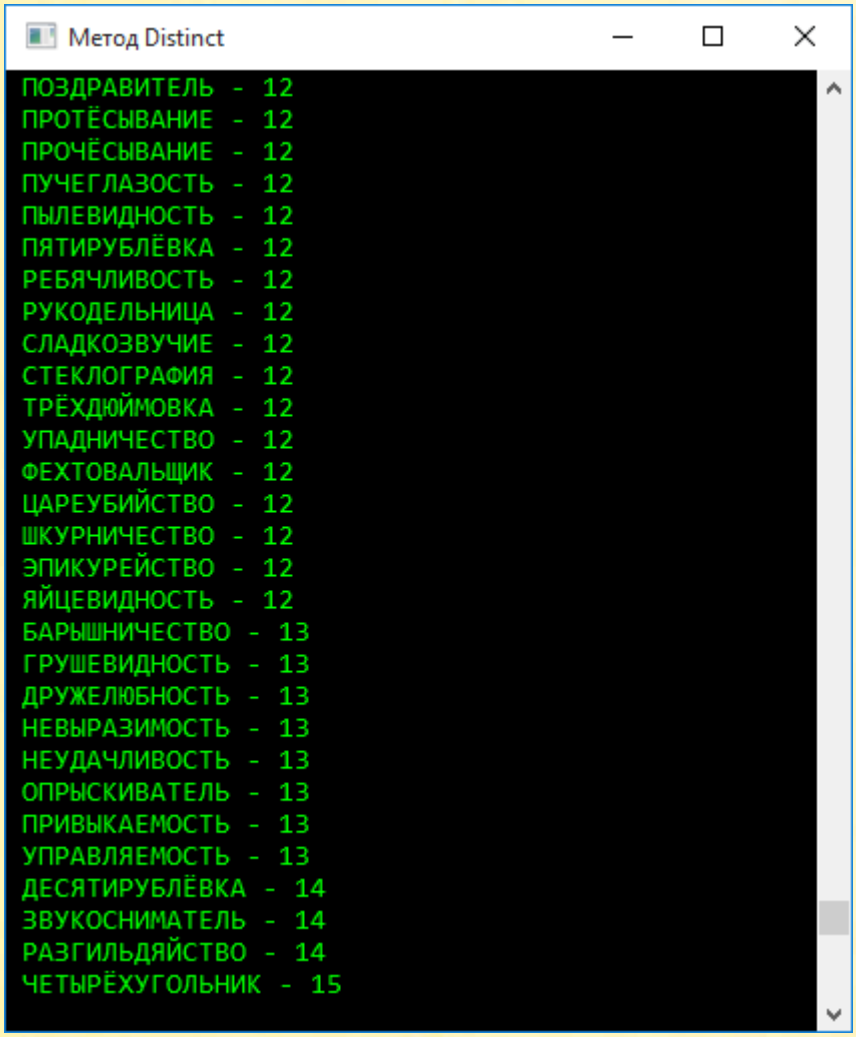


```
sq.Distinct().Sorted.Println;
```

Метод *Distinct* поможет нам найти все **разнобуквицы** русского языка:

```
var lstStrAll := ReadAllLines('OSH-W97.txt');
var razn := lstStrAll
            .Where(s -> s.Length = s.Distinct().Count())
            .OrderBy(s -> s.Length)
            .ThenBy(s -> s)
            .Select(s -> s + ' - ' + s.Length + NewLine)
            .Println;
```

В разнобуквицах буквы не повторяются, поэтому длина исходного слова должна быть равна длине слова, полученного после удаления из него повторяющихся букв:



Второй метод Distinct для сравнения элементов последовательности использует заданный компаратор *comparer* типа *IEqualityComparer<T>*:

```
function Distinct(comparer: IEqualityComparer<T>): sequence of T;
```

Метод *Union*

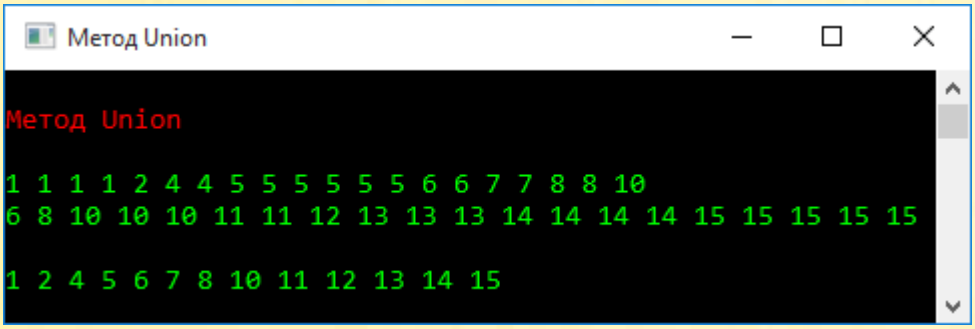
Метод Union *объединяет* две последовательности. При этом в выходной последовательности остаются элементы *без повторов*. Для сравнения элементов используется *компаратор* проверки на равенство по умолчанию:

```
function Union(second: sequence of T): sequence of T;
```

Создаём две последовательности чисел, которые для удобства отсортируем, и находим их объединение:

```
var sq1:= SeqRandom(20,1,10).ToArray.OrderBy(n -> n).Println;
var sq2:= SeqRandom(20,6,16).ToArray.OrderBy(n -> n).Println;

Println;
var union := sq1.Union(sq2)
                .Println;
```

На рисунке показана объединённая последовательность. Из неё исчезли повторяющиеся элементы последовательности *sq1*, после которых следуют элементы последовательности *sq2*, не совпадающие с элементами первой последовательности.



Второй метод Union также объединяет две последовательности, но для сравнения элементов использует компаратор *comparer*:

```
function Union(second: sequence of T; comparer: IEqualityComparer<T>):
                sequence of T;
```

Метод *Intersect*

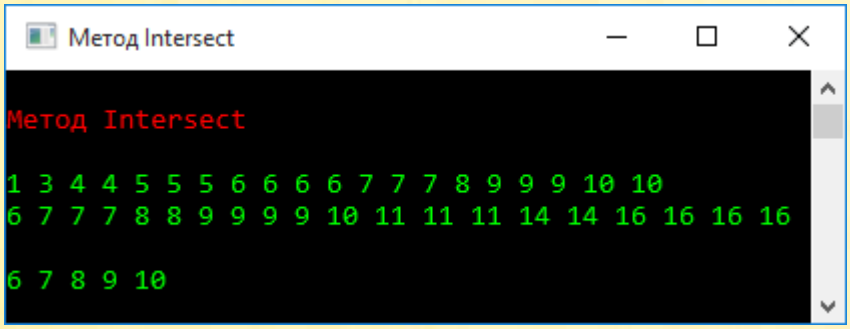
Метод Intersect возвращает *пересечение* двух последовательностей, то есть последовательность, состоящую из элементов, имеющихся в *обеих* последовательностях. Для сравнения элементов используется *компаратор* проверки на равенство по умолчанию:

```
function Intersect(second: sequence of T): sequence of T;
```

Найдём пересечение числовых последовательностей *sq1* и *sq2*:

```
var sq1:= SeqRandom(20,1,10).ToArray.OrderBy(n -> n).Println;
var sq2:= SeqRandom(20,6,16).ToArray.OrderBy(n -> n).Println;

Println;
var inter := sq1.Intersect(sq2)
                .Println;
```

В последовательности inter оказались элементы, которые содержатся в обеих последовательностях:



Во втором методе Intersect для сравнения элементов используется заданный компаратор *comparer*:

```
function Intersect(second: sequence of T;
                   comparer: IEqualityComparer<T>): sequence of T;
```

Метод *Except*

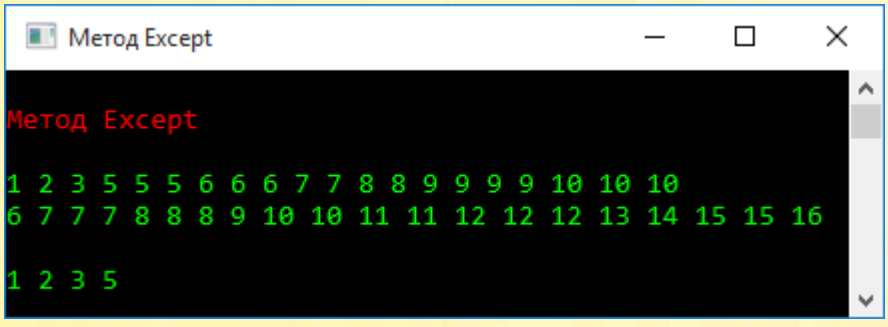
Метод Except возвращает *разность* двух последовательностей, то есть последовательность, состоящую из элементов первой последовательности за исключением тех элементов, которые содержатся и во второй последовательности. Для сравнения элементов используется *компаратор* проверки на равенство по умолчанию:

```
function Except(second: sequence of T): sequence of T;
```

Найдём разность числовых последовательностей *sq1* и *sq2*:

```
var sq1:= SeqRandom(20,1,10).ToArray.OrderBy(n -> n).Println;
var sq2:= SeqRandom(20,6,16).ToArray.OrderBy(n -> n).Println;

Println;
var exc := sq1.Except(sq2)
               .Println;
```

В последовательности exc оказались элементы первой последовательности, которые не содержатся во второй последовательности:



Во втором методе Except для сравнения элементов используется заданный компаратор *comparer*:

```
function Except(second: sequence of T; comparer: IEqualityComparer<T>):
                sequence of T;
```

Метод *Contains*

Метод Contains возвращает *true*, если последовательность содержит элемент с заданным значением *value*, используя *компаратор* проверки на равенство по умолчанию.:

```
function Contains(value: T): boolean;
```

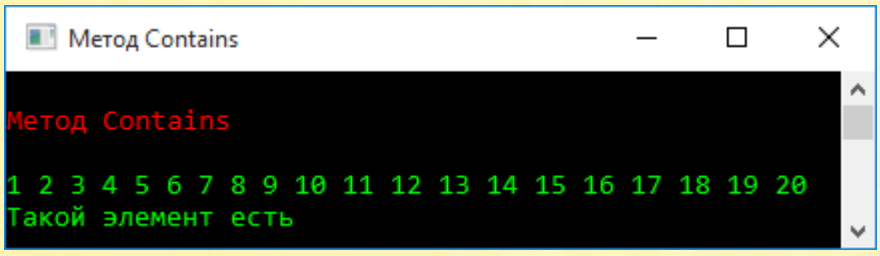
Создаём последовательность натуральных чисел 1..20:

```
var sq := Range(1, 20).Println;
```

И узнаём, есть ли среди них число 12:

```
if sq.Contains(12) then
   Println('Такой элемент есть')
else
   Println('Такого элемента нет');
```

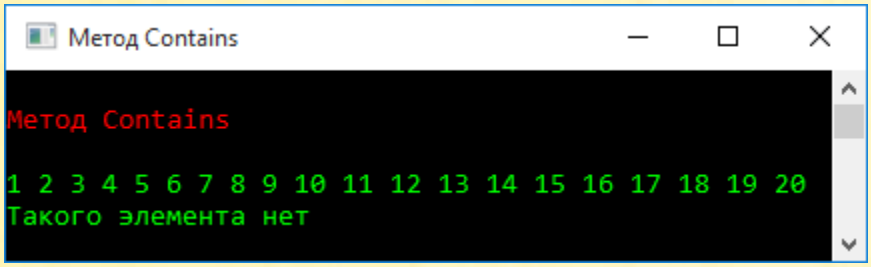
Рисунок подтверждает, что такое число содержится в последовательности:



Изменим условие:

```
if sq.Contains(22) then
   Println('Такой элемент есть')
else
   Println('Такого элемента нет');
```

А числа 22 в последовательности нет:



Второй метод Contains использует для сравнения элементов заданный компаратор *comparer*:

```
function Contains(value: T; comparer: IEqualityComparer<T>): boolean;
```

Методы *Single* и *SingleOrDefault*

Метод Single возвращает первый и *единственный* элемент последовательности:

```
function Single(): T;
```

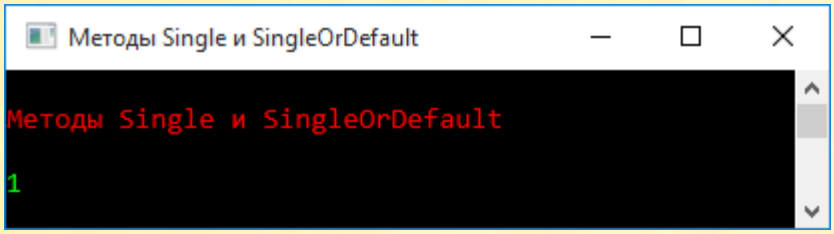
Если в исходной коллекции больше одного элемента или вообще нет элементов, генерируется *исключение*.

Создаём последовательность с *единственным* элементом:

```
var res := Range(1, 1)
           .Single;
```

И печатаем его:

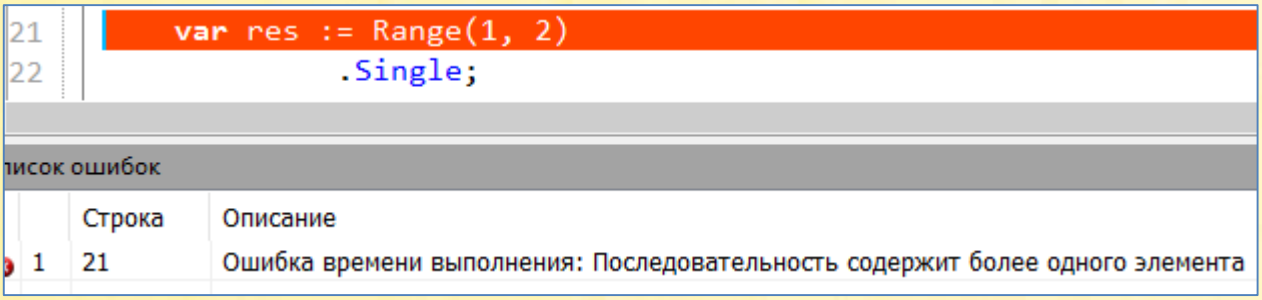
```
Println(res);
```



Добавим в последовательность *второй* элемент:

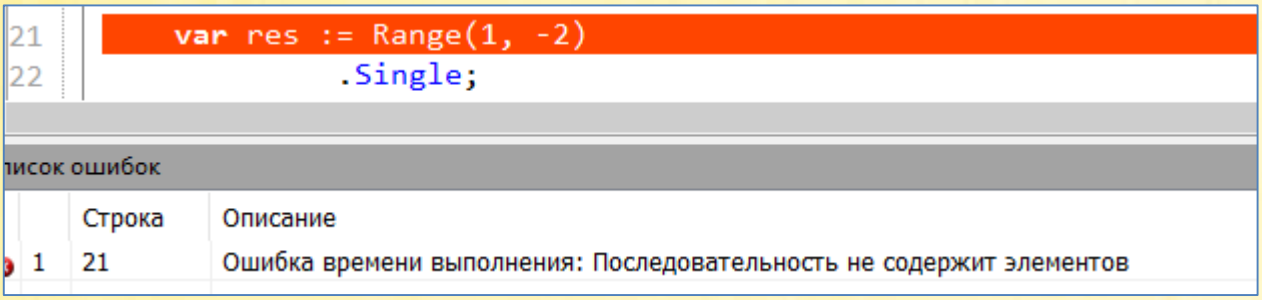
```
var res := Range(1, 2)
           .Single;
```

При запуске программы получаем **сообщение**:



Если последовательность не содержит элементов, то также возникает **ошибка**:

```
var res := Range(1, -2)
           .Single;
```



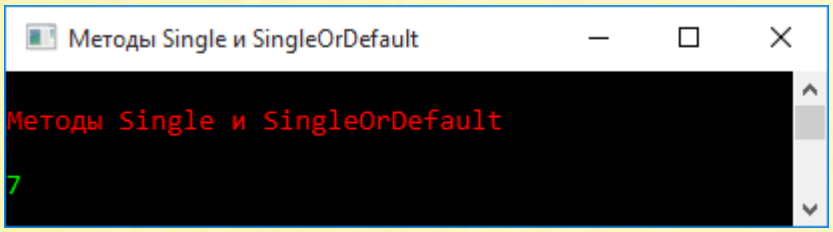
Второй метод Single проверяет элементы исходной коллекции. Если единственный элемент удовлетворяет условию, то он будет возвращён, в противном случае возникает исключение:

```
function Single(predicate: T->boolean): T;
```

Создаём последовательность из 10 натуральных чисел и проверим, имеется ли в ней *единственная* семёрка:

```
var res := Range(1, 10)
          .Single(n -> n = 7);
Println(res);
```

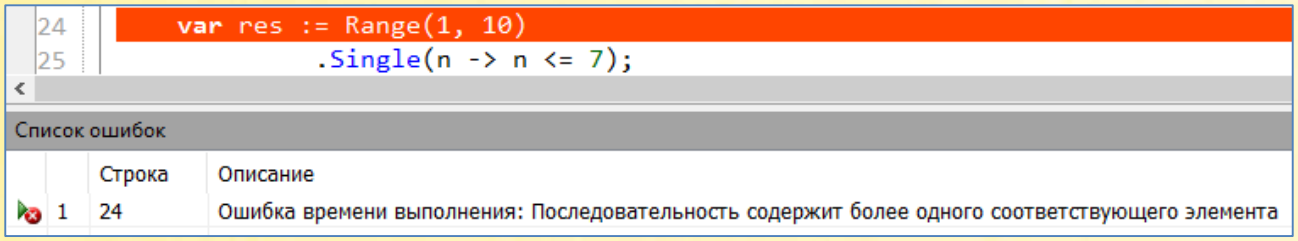
Всё верно:



Изменяем условие:

```
var res := Range(1, 10)
          .Single(n -> n <= 7);
Println(res);
```

Оно вызывает **ошибку**:



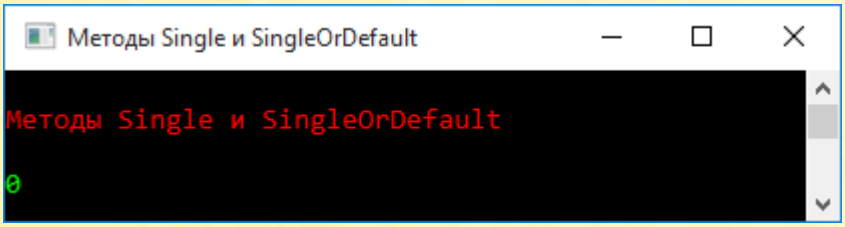
Чтобы избежать ошибок в работе программы, используйте метод **SingleOrDefault**. Он действует аналогично первому методу *Single*. Но если исходная коллекция *пустая*, то возвращается значение по *умолчанию* для типа *T*. Если последовательность содержит более одного элемента, генерируется исключение:

```
function SingleOrDefault(): T;
```

Создаём *пустую* последовательность и печатаем её первый элемент:

```
var res := Range(1, -1)
          .SingleOrDefault;
Println(res);
```

Так как последовательность не содержит элементов, то метод **SingleOrDefault** возвращает значение по умолчанию для типа *integer*, то есть **0**:



Но если последовательность содержит более одного элемента, возникает **ошибка**:

```
var res := Range(1, 2)
          .SingleOrDefault;
Println(res);
```

```
28      var res := Range(1, 2)
29                 .SingleOrDefault;
```

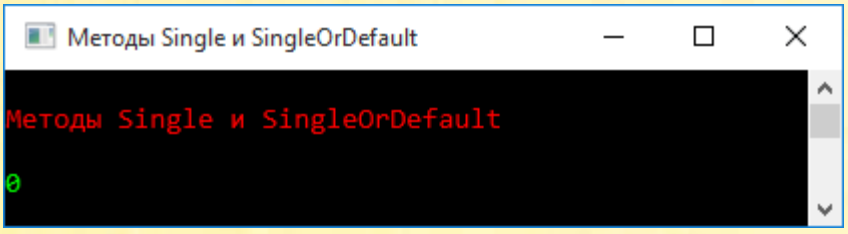
Список ошибок

	Строка	Описание
1	28	Ошибка времени выполнения: Последовательность содержит более одного элемента

Второй метод SingleOrDefault также возвращает единственный элемент или значение по умолчанию, если исходная коллекция пуста или в ней не нашлось ни одного элемента, удовлетворяющего *условию*:

```
function SingleOrDefault(predicate: T->boolean): T;
```

```
var res := Range(1, 10)
          .SingleOrDefault(n -> n > 10);
Println(res);
```



Если совпадение не единственное, возникает исключение:

```
28      var res := Range(1, 2)
29                 .SingleOrDefault;
30      Println(res);}
31
32      var res := Range(1, 10)
33                 .SingleOrDefault(n -> n > 7);
34      Println(res);
```

Список ошибок

	Строка	Описание
1	32	Ошибка времени выполнения: Последовательность содержит более одного соответствующего элемента

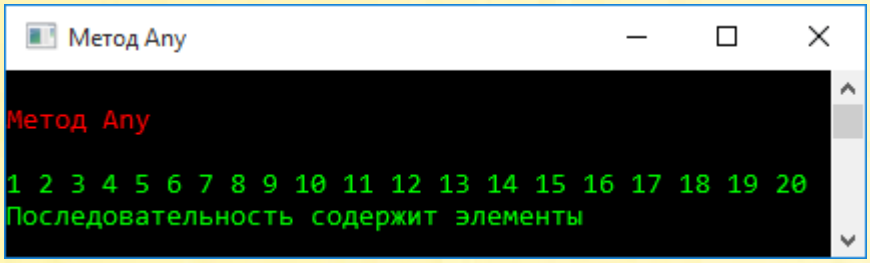
Метод *Any*

Метод Any возвращает *true*, если последовательность содержит элементы (не пустая):

```
function Any(): boolean;
```

Создаём последовательность из 20 элементов и проверяем её методом Any:

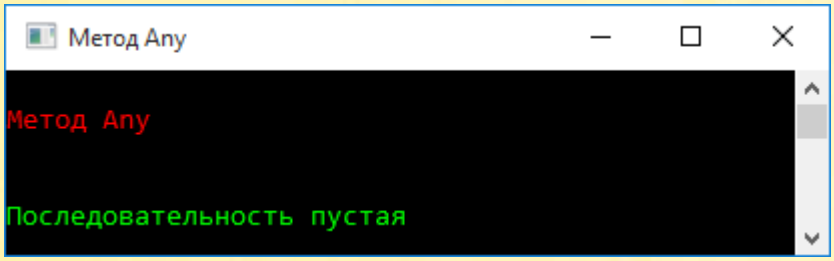
```
var sq := Range(1, 20).Println;

if sq.Any() then
   Println('Последовательность содержит элементы')
else
   Println('Последовательность пустая');
```



Создаём пустую последовательность:

```
var sq := Range(1, -20).Println;
```

Метод Any «узнал» её:



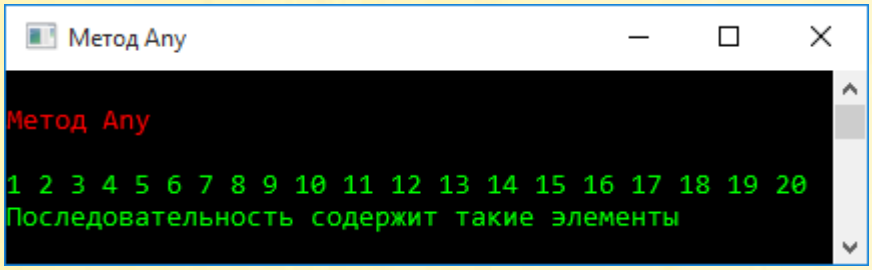
Второй метод Any возвращает *true*, если последовательность содержит хотя бы один элемент, удовлетворяющий условию *predicate*:

```
function Any(predicate: T->boolean): boolean;
```

Проверяем, имеется ли в последовательности *sq* хотя бы 1 элемент, который *меньше 20*:

```
var sq := Range(1, 20).Println;

if sq.Any(n -> n < 20) then
   Println('Последовательность содержит такие элементы')
else
   Println('Последовательность не содержит таких элементов');
```

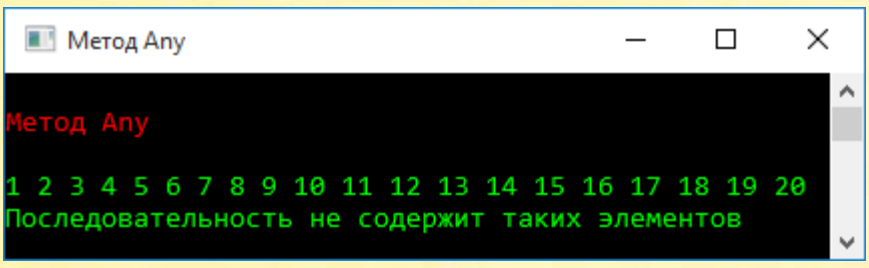
Метод Any утверждает правильно:



А больше 20?

```
var sq := Range(1, 20).Println;

if sq.Any(n -> n > 20) then
   Println('Последовательность содержит такие элементы')
else
   Println('Последовательность не содержит таких элементов');
```

А таких элементов нет:



Метод *All*

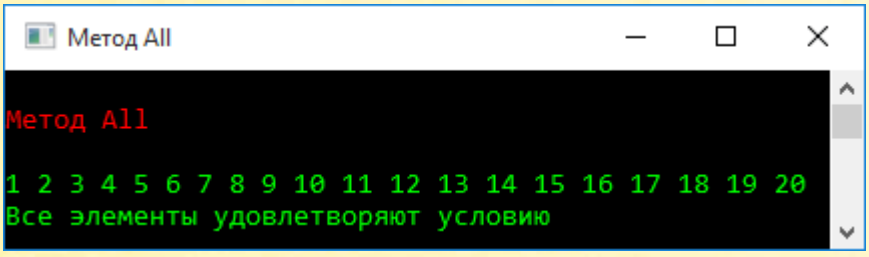
Метод All возвращает *true*, если *все* элементы последовательности удовлетворяют заданному условию *predicate*:

```
function All(predicate: T->boolean): boolean;
```

Создаём последовательность натуральных чисел 1..20 и проверяем, все ли элементы *не больше 20*:

```
var sq := Range(1, 20).Println;

if sq.All(n -> n <= 20) then
   Println('Все элементы удовлетворяют условию')
else
   Println('Не все элементы удовлетворяют условию');
```

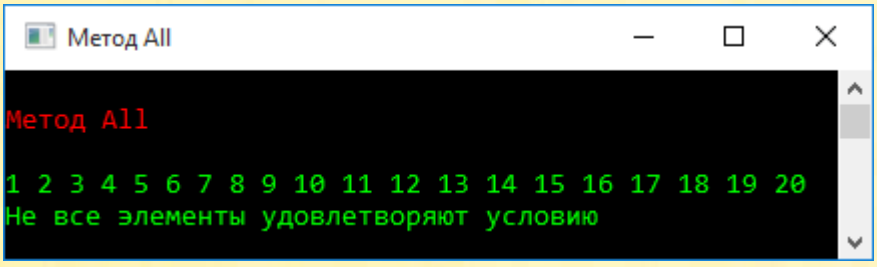
Метод All подтверждает, что все элементы последовательности удовлетворяют заданному условию:



А все ли элементы последовательности не больше 10?

```
if sq.All(n -> n <= 10) then
   Println('Все элементы удовлетворяют условию')
else
   Println('Не все элементы удовлетворяют условию');
```

Разумеется, не все:



Метод *DefaultIfEmpty*

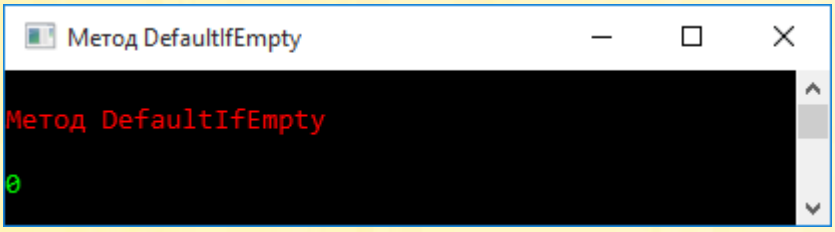
Метод DefaultIfEmpty возвращает исходную коллекцию *source*, если она не пуста, или одноэлементную последовательность со значением элемента по умолчанию в противном случае:

```
function DefaultIfEmpty(): sequence of T;
```

Создаём пустую последовательность и пропускаем её через метод DefaultIfEmpty:

```
var sq := Range(1, -20).DefaultIfEmpty.Println;
```

Он возвращает последовательность из одного элемента, значение которого равно нулю:



Второй метод DefaultIfEmpty позволяет задать любое значение *defaultValue* по умолчанию для пустой коллекции:

```
function DefaultIfEmpty(defaultValue: T): sequence of T;
```

Если последовательность пустая, то метод DefaultIfEmpty вернёт последовательность из одного элемента, значение которого равно *-1*:

```
var sq := Range(1, -20).DefaultIfEmpty(-1).Println;
```

Метод *SequenceEqual*

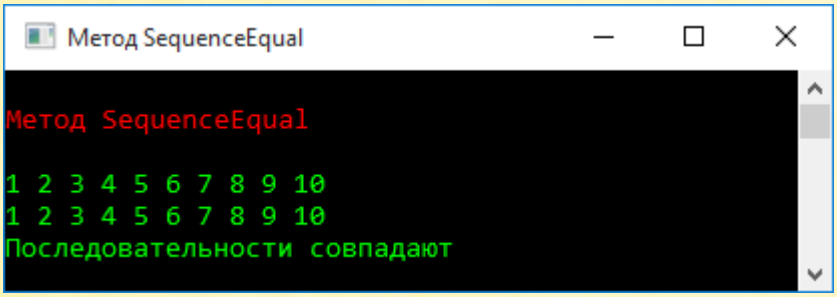
Метод SequenceEqual возвращает *true*, если две последовательности совпадают по числу элементов, по их значению и порядку. При сравнении элементов последовательности используется *компаратор* по умолчанию для типа *T*:

```
function SequenceEqual(second: sequence of T): boolean;
```

Создаём 2 одинаковые последовательности:

```
var sq1 := Range(1, 10).Println;
var sq2 := Range(1, 10).Println;

if sq1.SequenceEqual(sq2) then
   Println('Последовательности совпадают')
else
   Println('Последовательности не совпадают');
```

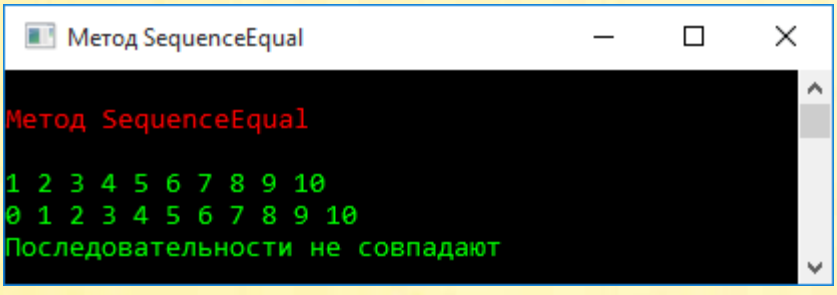
Метод SequenceEqual показывает, что они *совпадают*:



Добавим ко второй последовательности ещё 1 элемент:

```
var sq2 := (0 + Range(1, 10)).Println;
```

Теперь последовательности не совпадают:



Второй метод SequenceEqual использует для сравнения элементов заданный компаратор *comparer*:

```
function SequenceEqual(second: sequence of T;
                       comparer: IEqualityComparer<T>): boolean;
```

Метод *OfType*

Метод OfType возвращает последовательность элементов заданного типа *Res* из исходной последовательности:

```
function OfType<Res>(): sequence of Res;
```

Этот метод имеет смысл применять к коллекциям, которые могут содержать разнотипные элементы.

Создаём массив элементов разных типов:

```
var arr := new object[](1,2,3,'один','два','три', 1.0, 2.0, 3.0);
```

И выбираем из них строки:

```
arr.OfType&<string>.Println;
```

Метод *Cast*

Метод Cast действует аналогично методу *OfType,* но, если в коллекции встретятся элементы другого типа, возникает *исключение*:

```
function Cast<Res>(): sequence of Res;
```

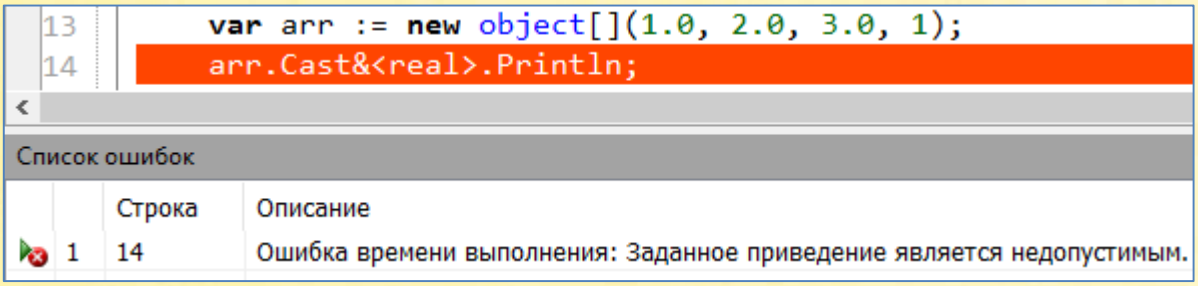
Создаём массив вещественных чисел:

```
var arr := new object[](1.0, 2.0, 3.0);
arr.Cast&<real>.Println;
```

Добавляем к ним число типа *integer*:

```
var arr := new object[](1.0, 2.0, 3.0, 1);
arr.Cast&<real>.Println;
```

И получаем сообщение об ошибке:

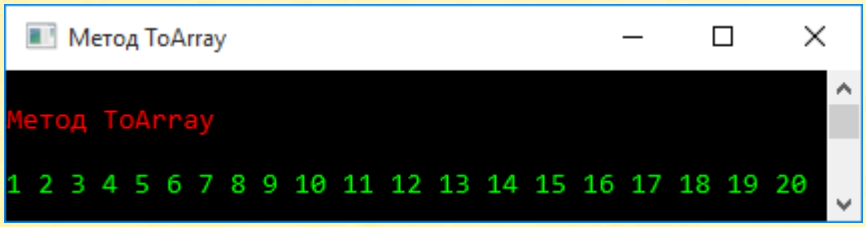


Метод *ToArray*

Метод ToArray возвращает *массив* элементов типа *T*, созданный из последовательности элементов типа *T*:

```
function ToArray(): array of T;
```

```
var arr := Range(1, 20)
           .ToArray
           .Println;
```

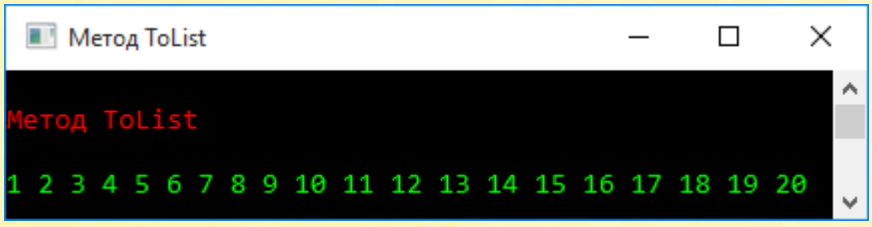


Метод *ToList*

Метод ToList возвращает **список** элементов типа *T*, созданный из последовательности того же типа:

```
function ToList(): List<T>;
```

```
var lst := Range(1, 20)
           .ToList
           .Println;
```



Метод *ToDictionary*

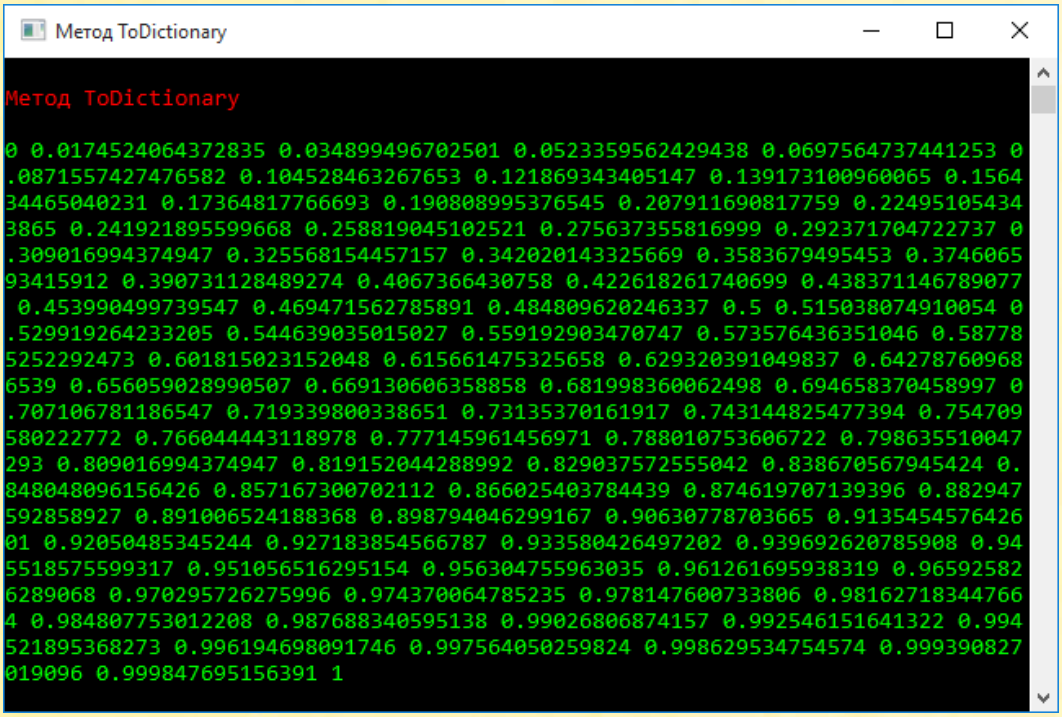
Метод ToDictionary возвращает *словарь*, созданный из последовательности. Поскольку словарь состоит из пар *ключ-значение*, а в последовательности имеются только значения, то с помощью метода *keySelector* необходимо создать ключ типа *Key*. Для сравнения ключей метод *ToDictionary* использует компаратор проверки на равенство по умолчанию:

```
function ToDictionary<Key>(keySelector: T->Key): Dictionary<Key,T>;
```

Все ключи в словаре должны быть разными!

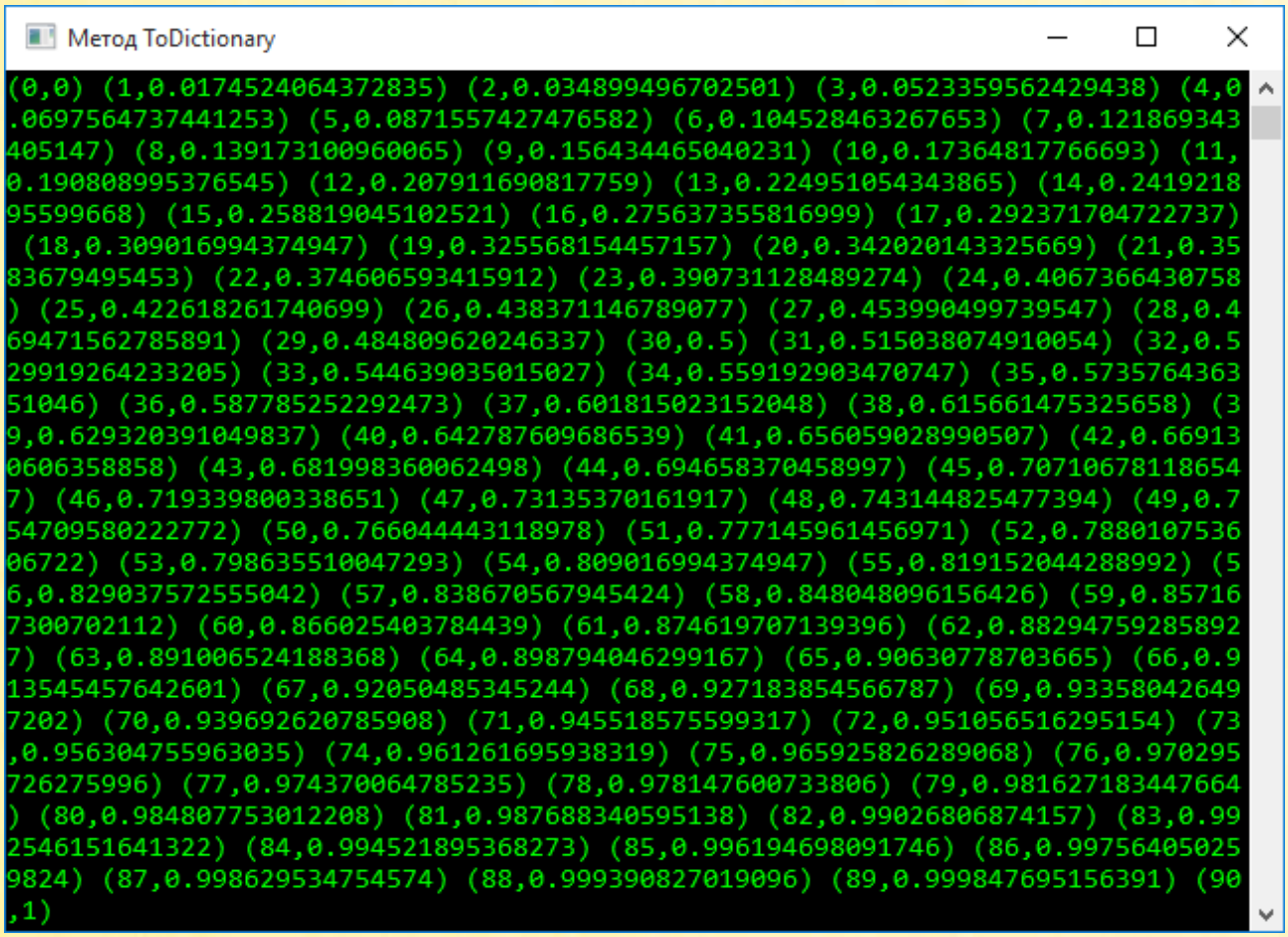
Пусть у нас имеется *таблица синусов* от 0 до 90 градусов, сохранённая в последовательности sq:

```
// создаём последовательность синусов:
var sq := Range(0, 90)
          .Select(i -> Sin(i * PI / 180.0))
          .Println;
```



Мы хотим создать **словарь**, в котором по значению ключа в градусах мы можем **сразу** получить соответствующее значение синуса, без повторного его вычисления. Вся сложность состоит в том, чтобы по значению синуса вычислить его угол. Проблема решается «тригонометрически»:

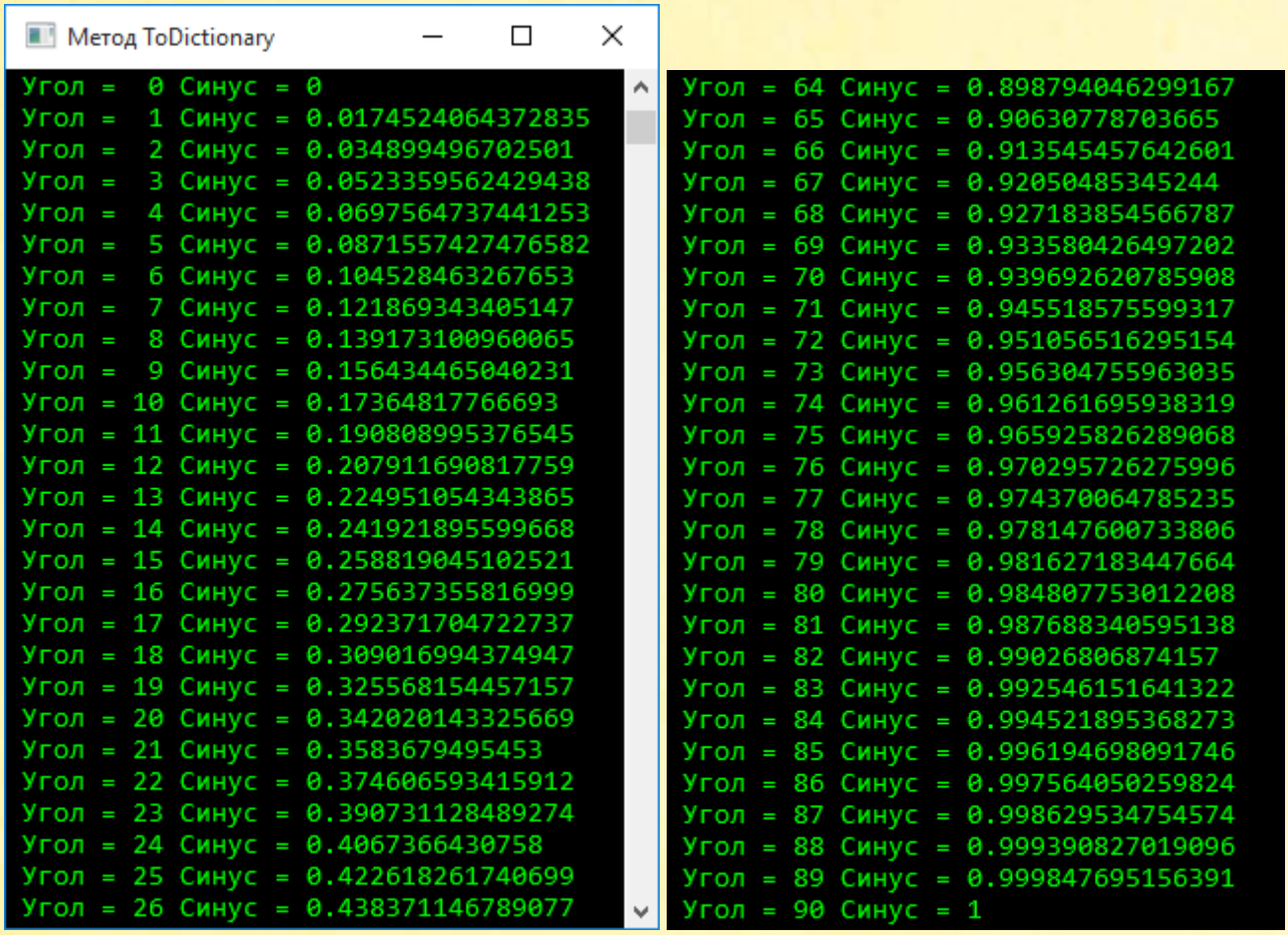
```
// создаём словарь:
var dic := sq
          .ToDictionary(i ->
           Round(ArcSin(i) * 180.0 / PI))
.Println;
```



Пользуясь словарём, мы быстро найдём синус любого угла от 0 до 90 градусов:

```
Println;
// печатаем таблицу:
```

```
for var i := 0 to 90 do
begin
    Console.Write(' Угол = ' + i.ToString().PadLeft(2));
    Console.WriteLine(' Синус = ' + dic.ElementAt(i).Value);
end;
```



В паскале *PascalABC.NET* методы **ToDictionary** и **ToLookup** возвращают последовательность типа *KeyValuePair*!

Метод ToDictionary имеет ещё три, более сложные разновидности:

```
function ToDictionary<Key>(keySelector: T->Key;
                           comparer: IEqualityComparer<Key>):
                           Dictionary<Key,T>;
```

```
function ToDictionary<Key,Element>(keySelector: T->Key;
                                   elementSelector: T->Element):
                                   Dictionary<Key,Element>;
```

```
function ToDictionary<Key,Element>(keySelector: T->Key;
                                   elementSelector: T->Element;
                                   comparer: IEqualityComparer<Key>):
                                   Dictionary<Key,Element>;
```

В **первой** из них для сравнения ключей использует заданный компаратор проверки на равенство **comparer**.

Во **второй** для вычисления значения элементов используется функция **elementSelector**.

Третья перегрузка используется одновременно и **comparer**, и **elementSelector**.

Если значение *comparer* равно *null*, то для сравнения ключей используется компаратор проверки на равенство по умолчанию.

Метод *ToLookup*

Метод ToLookup возвращает *множественный словарь*, созданный из последовательности с помощью метода *keySelector:*

```
function ToLookup<Key>(keySelector: T->Key):
                       System.Linq.ILookup<Key,T>;
```

Загружаем словарь в строковый массив:

```
var lstStrAll := ReadAllLines('OSH-W97.txt');
```

Создаём из него множественный словарь lookup, последовательно применяя методы:

- Where – чтобы отбросить слова, которые длиннее 6 букв
- OrderBy – чтобы отсортировать слова по длине
- ToLookup – чтобы создать множественный словарь, в котором роль ключа играет длина слов:

```
var lookup := lstStrAll
              .Where(s -> s.Length < 6)
              .OrderBy(s -> s.Length)
              .ToLookup(s -> s.Length)
              .Println;
```

Таким образом, в словаре lookup для каждой длины слов создана последовательность строк. Получить эту последовательность можно по *ключу*. Например, если мы хотим напечатать список слов из 5 букв, то сначала получаем коллекцию слов, а затем печатаем её на экране (рис. на следующей странице):

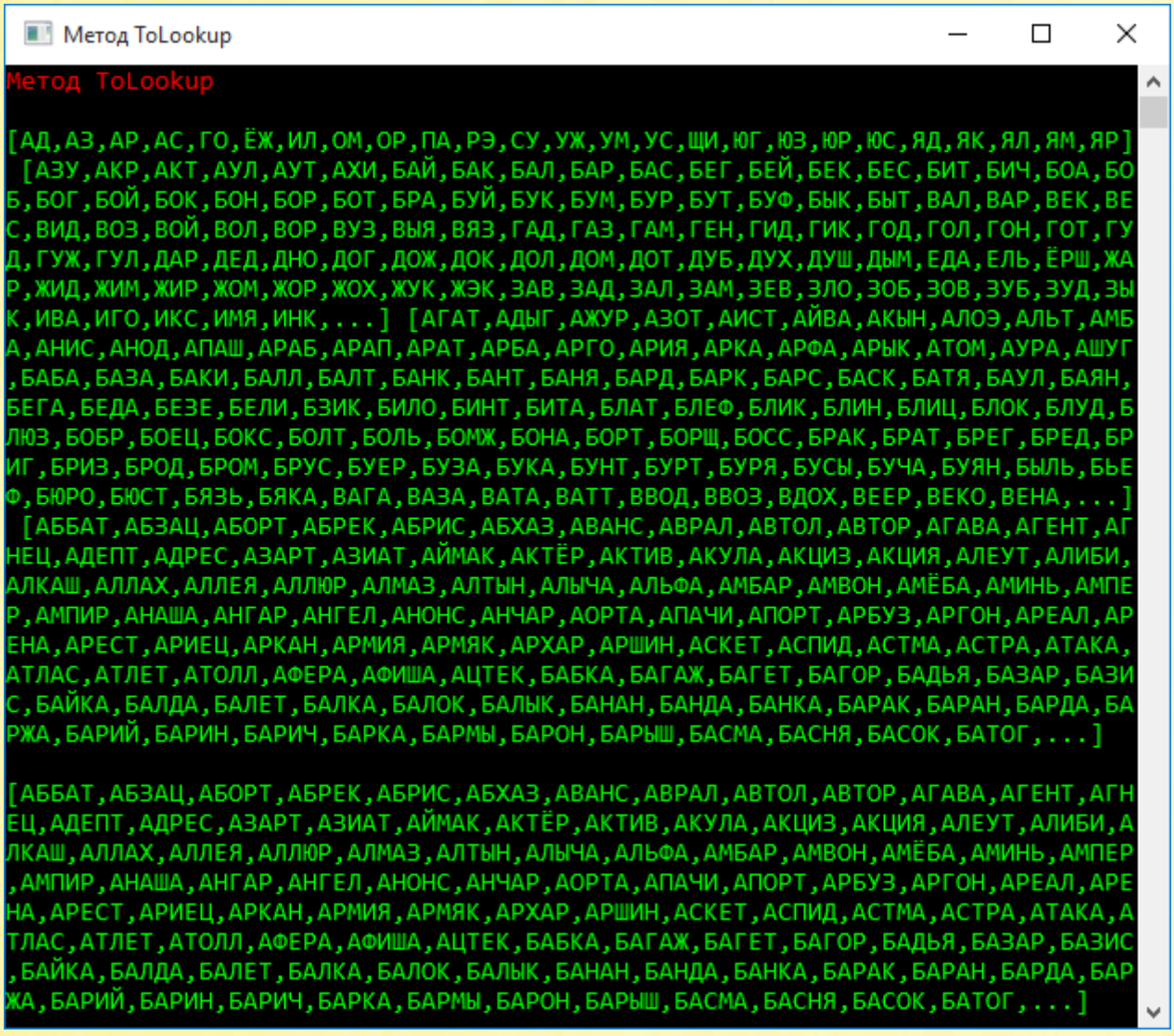
```
var words := lookup.ElementAt(3);
Println(words);
```

Метод ToLookup имеет ещё три разновидности. Они в точности повторяют разновидности метода *ToDictionary*:

```
function ToLookup<Key>(keySelector: T->Key;
                       comparer:
                       IEqualityComparer<Key>):
                       System.Linq.ILookup<Key,T>;
```

```
function ToLookup<Key,Element>(keySelector: T->Key;
                               elementSelector: T->Element):
                               System.Linq.ILookup<Key,Element>;
```

```
function ToLookup<Key,Element>(keySelector: T->Key;
                               elementSelector: T->Element;
                               comparer: IEqualityComparer<Key>):
                               System.Linq.ILookup<Key,Element>;
```



Метод *AsEnumerable*

Метод AsEnumerable возвращает последовательность того же типа, что и заданная:

```
function AsEnumerable(): sequence of T;
```

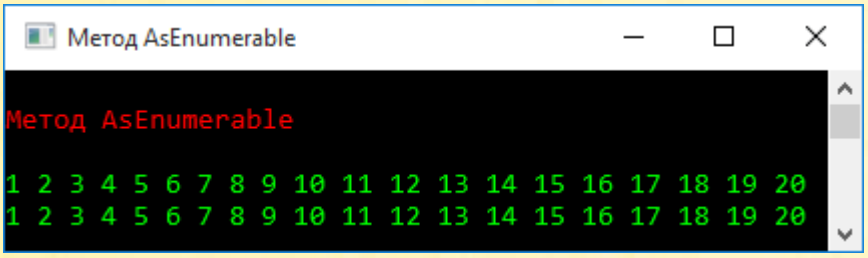
Создаём **список** из 20 элементов типа *integer*:

```
var list := Lst(Range(1, 20))
            .Println;
```

И конвертируем его в последовательность того же типа:

```
var sq:= list.AsEnumerable()
          .Println;
```

Элементы списка и последовательности, естественно, одинаковые:



Метод *Aggregate*

Метод Aggregate возвращает значение типа *T*, вычисленное аккумулирующей функцией *func*, которая применяется ко всем элементам заданной последовательности:

```
function Aggregate(func: (T,T)->T): T;
```

Все параметры функции **func** имеют один и тот же тип. **Первый** параметр – это **аккумулятор**. Вначале он равен первому элементу заданной последовательности.

Второй параметр – это следующий элемент последовательности: второй, третий,..., последний. Он добавляется к аккумулятору. Конечное значение аккумулятора возвращается методом *Aggregate*. Тип возвращаемого значения определяется типом функции.

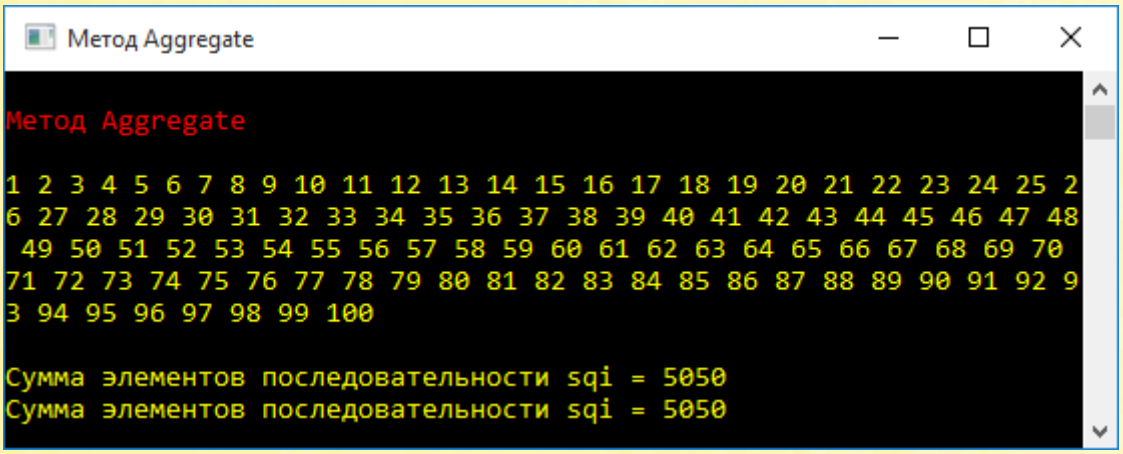
Найдём сумму первых 100 натуральных чисел в последовательности *sqi*, которую напечатаем на экране:

```
var sqi := Range(1, 100).Println;
```

Сумму элементов последовательности мы получим от метода *Aggregate*. Значение переменной **res** – это промежуточная сумма элементов заданной последовательности. Она инициализируется *первым* элементом последовательности, а затем мы друг за другом добавляем к промежуточной сумме *res* все остальные числа последовательности **next**, начиная со второго и заканчивая последним. Окончательное значение переменной *res* присваивается переменной **sum**, которую мы и печатаем на экране:

```
Console.WriteLine('Сумма элементов последовательности sqi = {0}',
                          sqi.Sum);
var sum := sqi.Aggregate((res, next) -> res + next);
Console.WriteLine('Сумма элементов последовательности sqi = {0}',
                          sum);
```

Для проверки печатаем сумму элементов, вычисленную методом расширения *Sum*. Рисунок показывает, что метод *Aggregate* правильно нашёл сумму:

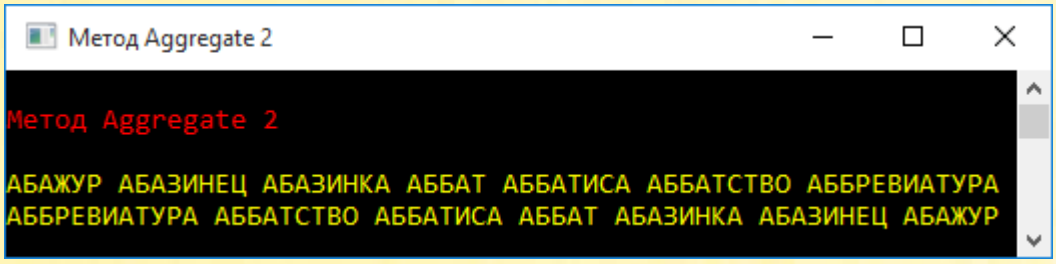


Второй метод Aggregate позволяет задать *начальное значение* аккумулятора *seed*:

```
function Aggregate<Accum>(seed: T; func: (Accum,T)->Accum): T;
```

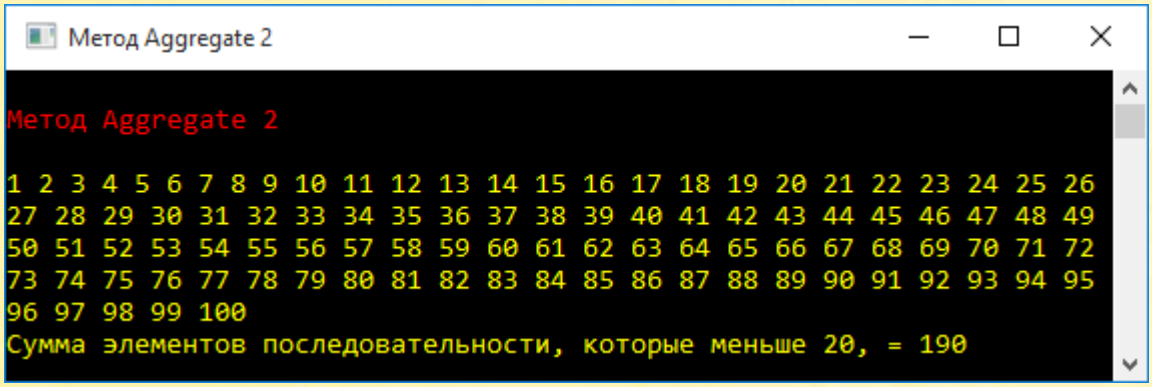
Создаём из строковой последовательности *lstStr* строку rev2, в которой слова расположены в *обратном* направлении.

```
// второй пример:

var lstStrAll := ReadAllLines('OSH-W97.txt').ToList();
var lstStr := lstStrAll.Take(7).Println;
var rev := lstStr.Aggregate('', (res, next) -> next + ' ' +
                                  res).Println;
Println;
```



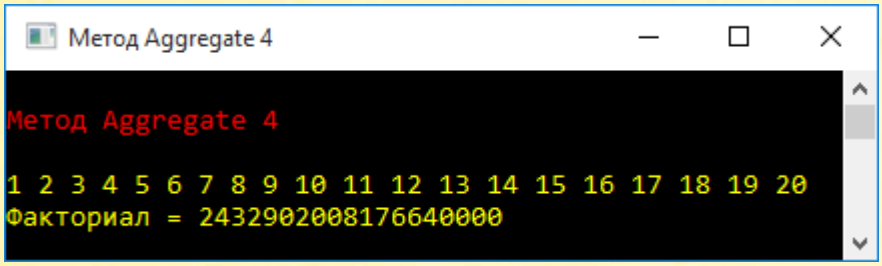
Найдём сумму чисел последовательности *sqi*, которые меньше 20:

```
var sqi := Range(1, 100).Println;
var sum := sqi.Aggregate(0, (res, next) -> next < 20 ?
                           res + next : res);
Console.WriteLine('Сумма элементов последовательности, ' +
                        'которые меньше 20, = {0}', sum);
```



Для больших целых чисел используйте тип decimal:

```
// факториал:
var sqi := Range(1, 20).Println;
var fact := sqi.Aggregate(decimal(1), (res, next) -> res * next);
Console.WriteLine('Факториал = {0}', fact);
```



Третий метод Aggregate дополнен ещё одной функцией - *resultSelector*, которая вычисляет возвращаемое значение:

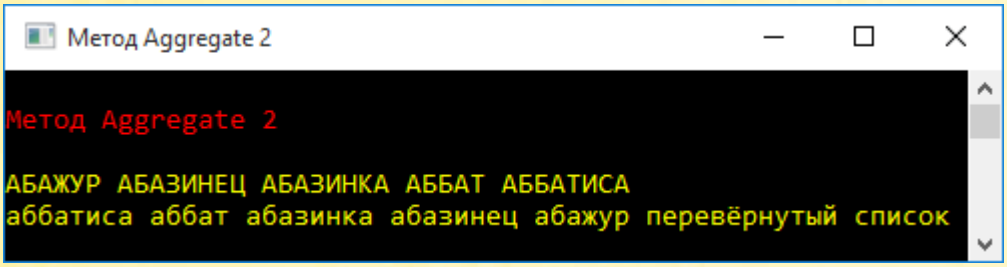
```
function Aggregate<Accum,Res>(seed: T; func: (Accum,T)->Accum;
                              resultSelector: Accum->Res): T;
```

```
// третий пример:

var lstStrAll := ReadAllLines('OSH-W97.txt').ToList();
var lstStr := lstStrAll.Take(5).Println;
```

Если строка длинная, то можно использовать экземпляр типа *StringBuilder*:

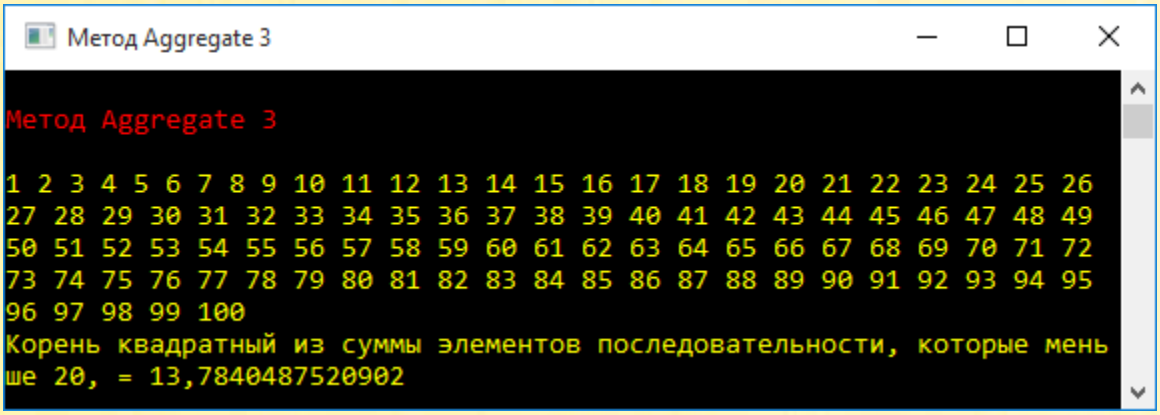
```
var rev := lstStr.Aggregate(new StringBuilder('Перевёрнутый список '),
                            (sb, next) -> sb.Insert(0, (next + ' ')),
                             sb -> sb.ToString().ToLower()).Println;
```

Здесь функция resultSelector конвертирует строку в нижний регистр:



Снова найдём сумму элементов последовательности *sqi*, которые меньше 20, но вернём не сумму, а *квадратный корень* из неё, применив к найденному значению *res* математическую операцию извлечения квадратного корня. Если результат – вещественное число, то начальное значение аккумулятора должно иметь тип *real*:

```
var sqi := Range(1, 100).Println;
var sum := sqi.Aggregate(0.0, (res, next) -> next < 20 ?
                          res + next : res, res -> Sqrt(res));
Console.WriteLine('Корень квадратный из суммы элементов
                   последовательности, ' +
                  'которые меньше 20, = {0}', sum);
```



Метод *Join*

Метод Join объединяет две последовательности типа *Res* (заданная, внешняя) и *TInner* (внутренняя) по ключам *outerKeySelector* и *innerKeySelector*. Для каждого элемента выходной последовательности вызывается метод *resultSelector*:

```
function Join<TInner,Key,Res>(inner: sequence of TInner;
                              outerKeySelector: T->Key;
                              innerKeySelector: TInner->TKey;
                              resultSelector: (T,TInner)->Res):
                              sequence of Res;
```

Для сравнения ключей используется *компаратор* проверки на равенство по умолчанию.

Разберём действие метода *Join* на примере.

Мы будем искать **слова-палиндромоиды**, то есть такие пары слов, что если одно слово из пары написать задом наперёд, то они совпадут. Например:

СКЕЛЕТ – ТЕЛЕКС
ОСЕЛОК - КОЛЕСО
ШРАМ – МАРШ

Для переворота слов используем строковый **метод расширения**:

```
function  Reverse(Self: String): string; extensionmethod;
begin
    Result := new string(Self.ToCharArray()
                              .Reverse()
                              .ToArray());
end;
```

Загружаем список слов lstStrAll:

```
var lstStrAll := ReadAllLines('OSH-W97.txt');
```

Просматриваем его и каждое слово переворачиваем с помощью метода Reverse. Снова просматриваем список слов *lstStrAll* и ищем в нём слово, которое совпа-

дает с перевёрнутым. Если такое слово обнаружится, то пара слов-палиндромоидов найдена, и мы добавляем строку с обоими словами в выходную последовательность.

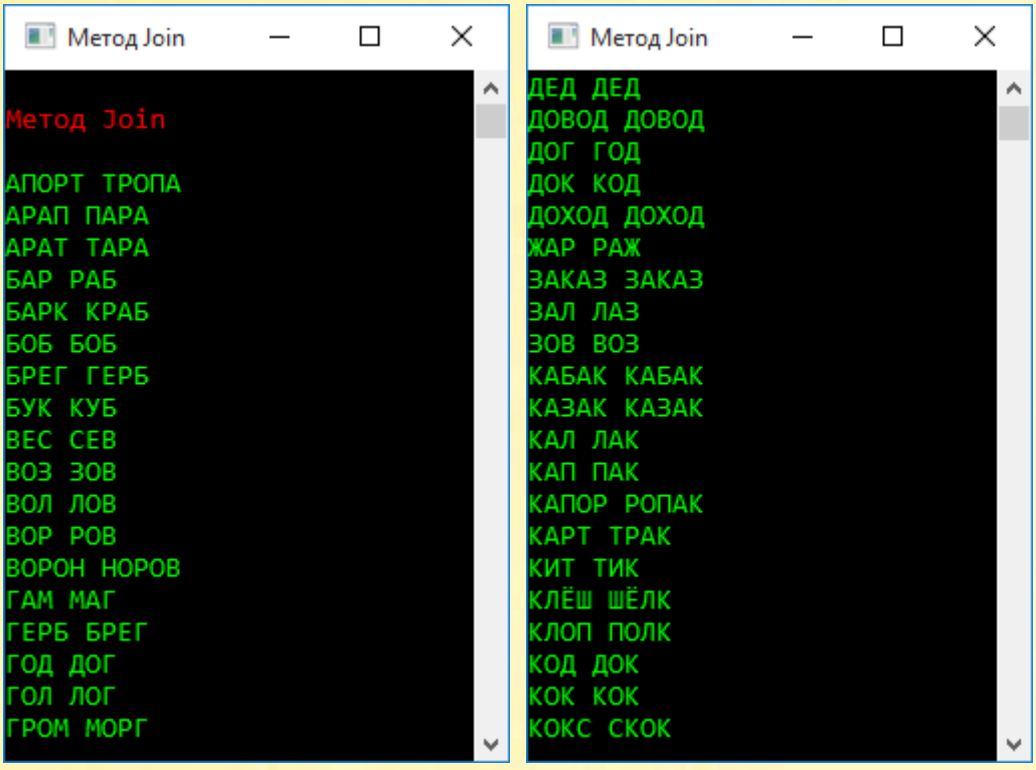
Из этого «алгоритма» можно сделать такие выводы:

- **Внешняя последовательность** – это список слов *lstStrAll*
- **Внутренняя последовательность** – это также список слов *lstStrAll*, поскольку все слова содержатся в одной и той же последовательности. В других задачах это может быть, конечно, другая последовательность – и даже не обязательно строковая.
- **Ключ для слов первой последовательности** – перевёрнутое слово. Его мы получаем от метода *outerKeySelector*.
- **Ключ для слов второй последовательности** – прямое слово. Его мы получаем от метода *innerKeySelector*.
- Если ключи (то есть слова – перевёрнутое и прямое) совпадут, то метод *resultSelector* формирует результат операции, который будет добавлен в выходную последовательность. Мы хотим получить строку, состоящую из слов-палиндромоидов, разделённых пробелом, поэтому выходная последовательность имеет тип *String*.

Для визуализации найденного списка палиндромоидов мы **печатаем** его в консольном окне:

```
//палиндромоиды и палиндромы:
var lst := lstStrAll.Join(lstStrAll,
                          s1 -> s1.Reverse(),
                          s2 -> s2,
                          (s1, s2) -> s1 + ' ' + s2)
                    .Println(NewLine);
```

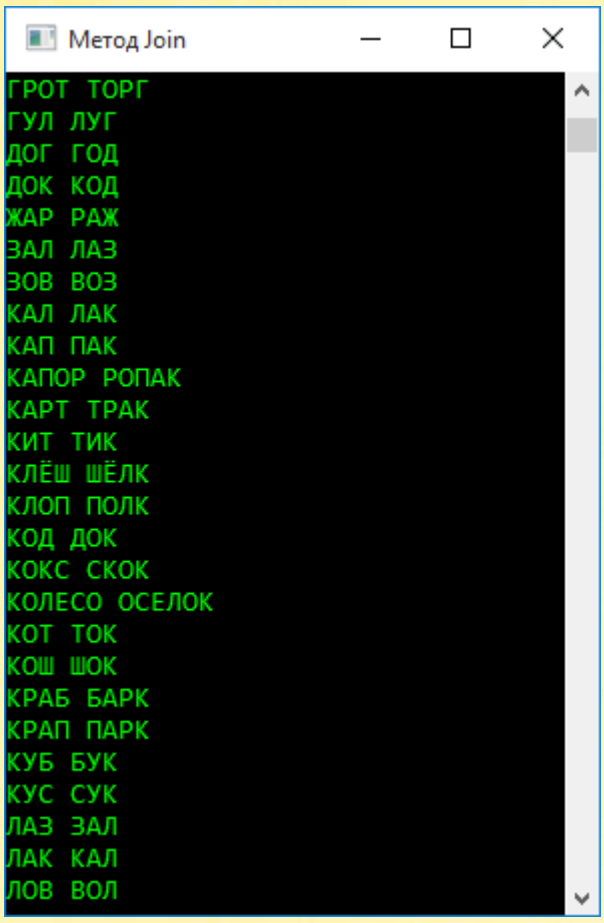
Пристальный взгляд на рисунок убеждает нас, что мы нашли не только собственно палиндромоиды, но и **палиндромы**:



От палиндромов можно избавиться, сравнив дополнительно первое (прямое!) слово со вторым: если они совпадут, значит, это палиндромы:

```
Println;
//палиндромоиды:
var lst2 := lstStrAll.Join(lstStrAll,
                           s1 -> s1.Reverse(),
                           s2 -> s2,
                          (s1, s2) -> s1 = s2 ? nil : s1 + ' ' + s2)
                    .Where(s -> s <> nil)
                    .Println(NewLine);
```

На рисунке вы уже не увидите палиндромов ДЕД, ДОВОД, ДОХОД, ЗАКАЗ, КАБАК, КАЗАК и КОК:



Аналогично мы можем получить только список палиндромов:

```
Println;
//палиндромы:
var lst3 := lstStrAll.Join(lstStrAll,
                           s1 -> s1.Reverse(),
                           s2 -> s2,
                          (s1, s2) -> s1 <> s2 ? nil : s1)
                     .Where(s -> s <> nil)
                     .Println(NewLine);
```

Второй метод Join для сравнения ключей используется заданный компаратор *comparer* проверки на равенство:

```
function Join<TInner,Key,Res>(inner: sequence of TInner;
```

```
                                outerKeySelector: T->Key;
                                innerKeySelector: TInner->TKey;
                                resultSelector: (T,TInner)->Res;
                                comparer:IEqualityComparer<Key>):
                                sequence of Res;
```

Метод *GroupJoin*

Метод GroupJoin действует так же, как метод *Join*, но возвращает группы последовательностей:

```
function GroupJoin<TInner,Key,Res>(inner: sequence of TInner;
                                   outerKeySelector: T->Key;
                                   innerKeySelector: TInner->TKey;
                    resultSelector: (T,sequence of TInner)->Res):
                                   sequence of Res;
```

Для сравнения ключей используется *компаратор* проверки на равенство по умолчанию.

Займёмся поиском анаграмм. Внимательный взор на список анаграмм обнаружит слова, которые имеют не одну анаграмму, а больше – от двух до шести. Например, слову ТОРС принадлежат анаграммы РОСТ и СОРТ, слову ТОРФ – ФОРТ и ФТОР, слову ТРОПИК – ПОРТИК, ПРИТОК и ПОРТКИ.

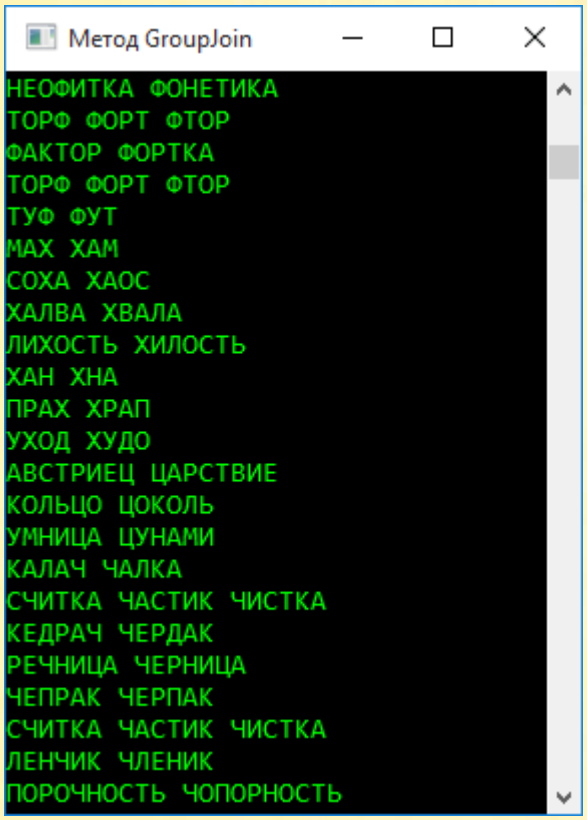
Лучше и правильнее объединить такие родственные анаграммы в одну группу, заменив метод *Join* групповым методом объединения последовательностей GroupJoin:

```
//анаграмммы:
var groups := lstStrAll.GroupJoin(lstStrAll,
              s1 -> new string(s1.OrderBy(ch -> ch).ToArray),
              s2 -> new string(s2.OrderBy(ch -> ch).ToArray),
              (s1, g) -> s1 = g.ElementAt(0) ?
                         nil : String.Join(' ', g))
                        .Where(s -> s <> nil)
```

```
                .Println(NewLine);
```

Второй параметр метода **resultSelector** имеет тип не *TInner*, как раньше, а **sequence of** TInner, то есть последовательность всех слов второй последовательности, имеющих ключ, который возвращает метод *outerKeySelector*. В нашем случае в группе слов окажутся все слова, состоящие из тех же букв, что и отсортированное слово *s1*, включая и само первое слово.

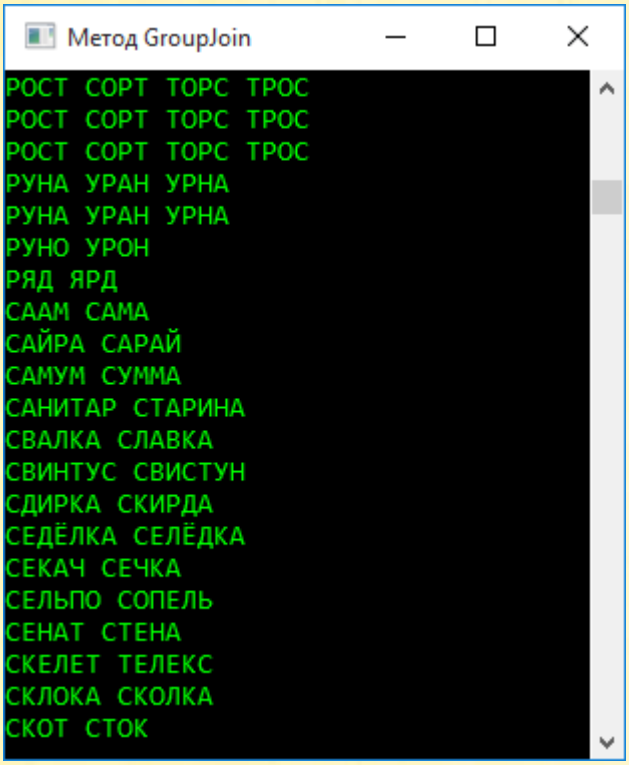
Изъян этого способа состоит в том, что большие группы анаграмм будут повторяться в выходной последовательности (они располагаются не в алфавитном порядке):



Для ясности восприятия отсортируем группы по алфавиту:

```
Println;
var groups2 := groups.OrderBy(s -> s)
                     .Println(NewLine);
```

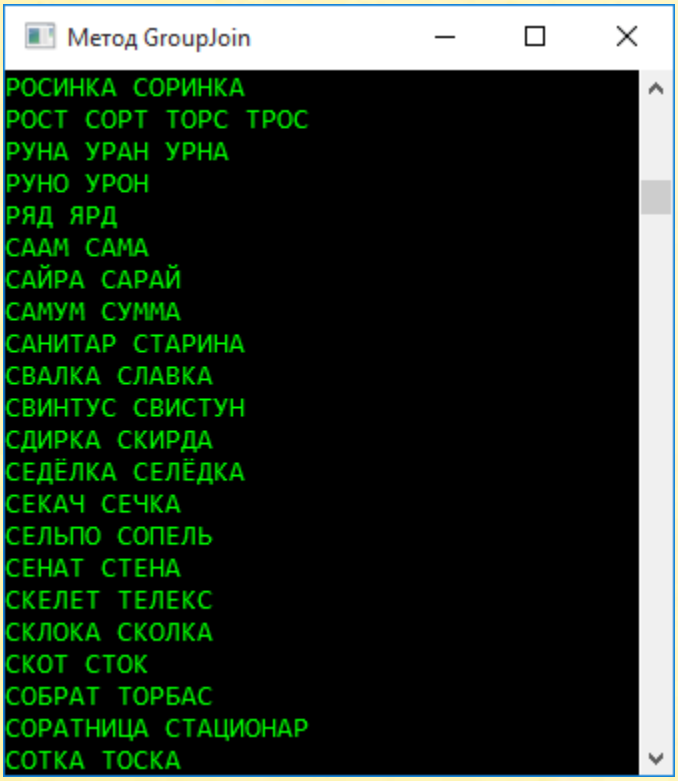
На рисунке вы видите, что группа РОСТ-СОРТ-ТОРС-ТРОС присутствует в списке трижды, а группа РУНА-УРАН-УРНА – дважды:



С помощью метода Distinct мы отсеем одинаковые строки в группах:

```
var groups2 := groups.Dis-
tinct().OrderBy(s -> s)

.Println(NewLine);
```

Теперь все группы в последовательности *groups2* содержатся однократно:



Второй метод GroupJoin для сравнения ключей использует заданный компаратор *comparer* проверки на равенство:

```
function GroupJoin<TInner,Key,Res>(inner: sequence of TInner;
                                   outerKeySelector: T->Key;
                                   innerKeySelector: TInner->TKey;
                        resultSelector: (T,sequence of TInner)->Res;
                                   comparer: IEqualityComparer<Key>):
                                   sequence of Res;
```

Метод *GroupBy*

Метод GroupBy возвращает последовательность, в которой элементы разбиты на *группы* по значению ключа. Ключ для каждого элемента заданной последовательности вычисляет метод *keySelector*:

```
function GroupBy<Key>(keySelector: T->Key):
                        IEnumerable<IGrouping<Key,T>>;
```

Для сравнения ключей используется *компаратор* проверки на равенство по умолчанию.

В новом проекте мы разобьём последовательность чисел *sq* на 2 группы – **чётные** числа и **нечётные** числа. Для чётных чисел остаток от деления на 2 равен нулю, для нечётных - единице. Таким образом, **ключом** может служить остаток от деления числа на двойку:

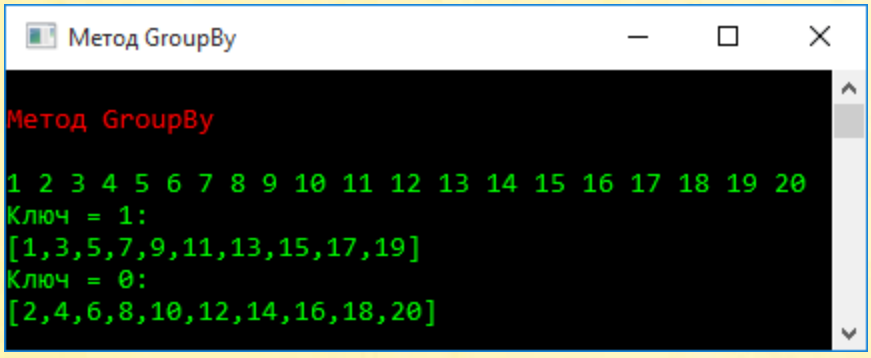
```
var sq := Range(1, 20)
          .Println;

var groupi := sq.GroupBy(n -> n mod 2);
```

Метод **keySelector** получает число и находит для него *ключ* – 0 или 1. В зависимости от значения ключа метод **GroupBy** помещает число в соответствующую группу – чётных или нечётных чисел. В итоге мы получаем результирующую последовательность, по структуре напоминающую словарь *Dictionary* и состоящую из нескольких групп (возможно, ни из одной). Тип каждой группы - *IGrouping<Key, T>,* в данном примере - *IGrouping<integer, integer>.*

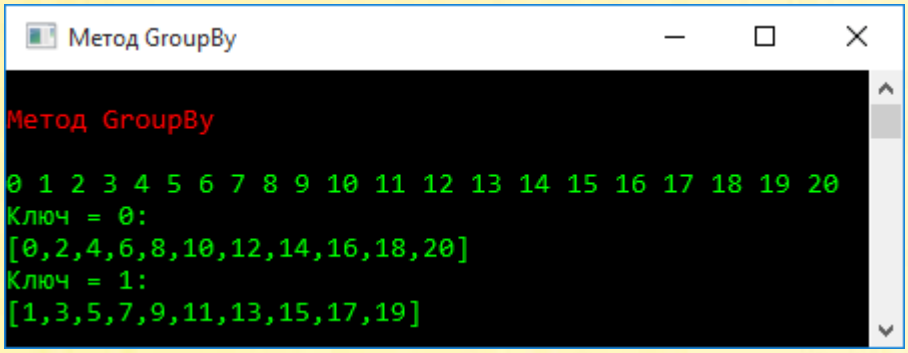
В переменной **groupi** мы получаем последовательность групп, которые можно перебрать в цикле *foreach*. Каждая группа имеет одно и то же значение *ключа*:

```
foreach var g in groupi do
begin
   Console.WriteLine('Ключ = {0}:', g.Key);
   Print(g);
   Console.WriteLine();
end;
```

На рисунке вы видите, что числа правильно разбиты на группы, но порядок групп в выходной последовательности зависит от значения первого элемента исходной последовательности. На верхнем рисунке последовательность начинается с группы *нечётных* чисел, а на нижнем – с группы *чётных* чисел:



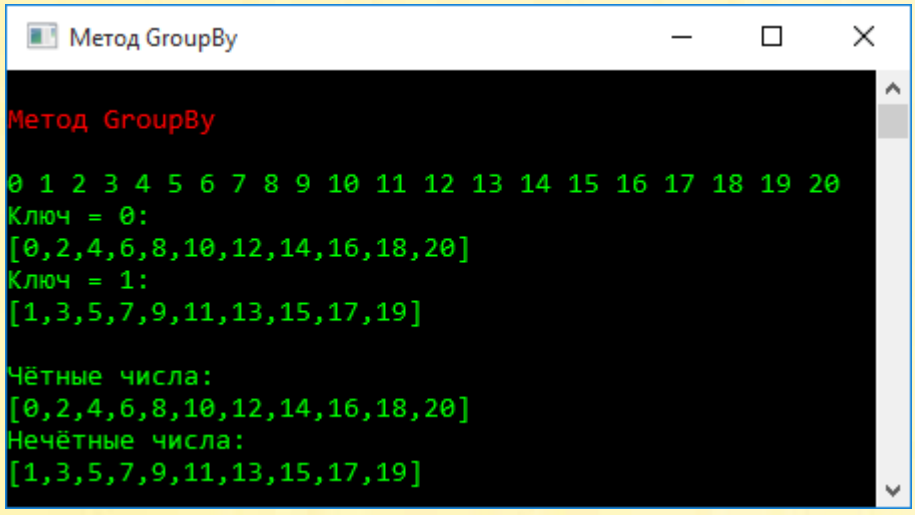

```
var sq := (0 + Range(1, 20))
          .Println;
```



Вполне разумно **отсортировать** группы в выходной последовательности по возрастанию значения ключа:

```
var groupi := sq
              .GroupBy(n -> n mod 2)
              .OrderBy(g -> g.Key);
```

Теперь в выходной последовательности группа чётных чисел будет предшествовать группе нечётных чисел – как на рисунке выше. Это значит, что вы можете гарантированно получить нужную группу чисел по **ключу** и, например, напечатать её:

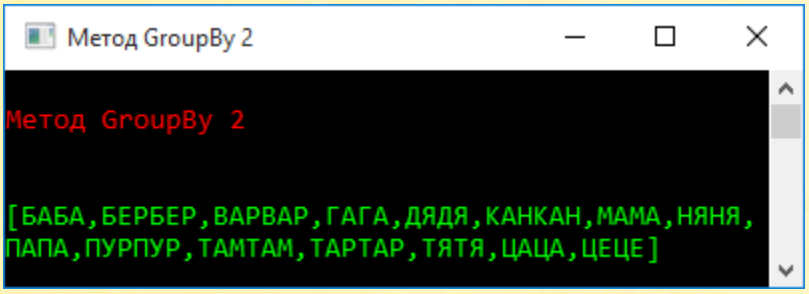



```
Println;
Console.WriteLine('Чётные числа:');
var gg := groupi.ElementAt(0);
Print(gg);
Console.WriteLine();
Console.WriteLine('Нечётные числа:');
gg := groupi.ElementAt(1);
Print(gg);
Println;
```

Во **втором примере** мы отыщем в словаре Ожегова все **слова-мамы**, то есть такие слова, которые состоят из двух одинаковых частей.

И здесь ключ может принимать два значения - **true** и **false**. Словам-мамам соответствует ключ со значением *true*:

```
var lstStrAll := ReadAllLines('OSH-W97.txt');

var groups := lstStrAll
              .GroupBy(s -> (s.Length mod 2 = 0) and
                            (s.Substring(0, s.Length div 2) =
                             s.Substring(s.Length div 2, s.Length div
2)));

foreach var g in groups do
begin
   if (g.Key) then
       Println(g, NewLine);
end;
```



Все остальные слова (а их больше 20 тысяч!) попадут в группу со значением ключа *false*. Поскольку нас эта группа не интересует, то с помощью метода *Where* мы можем оставить только группу с ключом *true*:

```
var groups := lstStrAll
              .GroupBy(s -> (s.Length mod 2 = 0) and
                            (s.Substring(0, s.Length div 2) =
                             s.Substring(s.Length div 2, s.Length div 2)))
                            .Where(g -> g.Key);
```

Цикл foreach напечатает тот же список слов-мам, но уже без дополнительной проверки ключа:

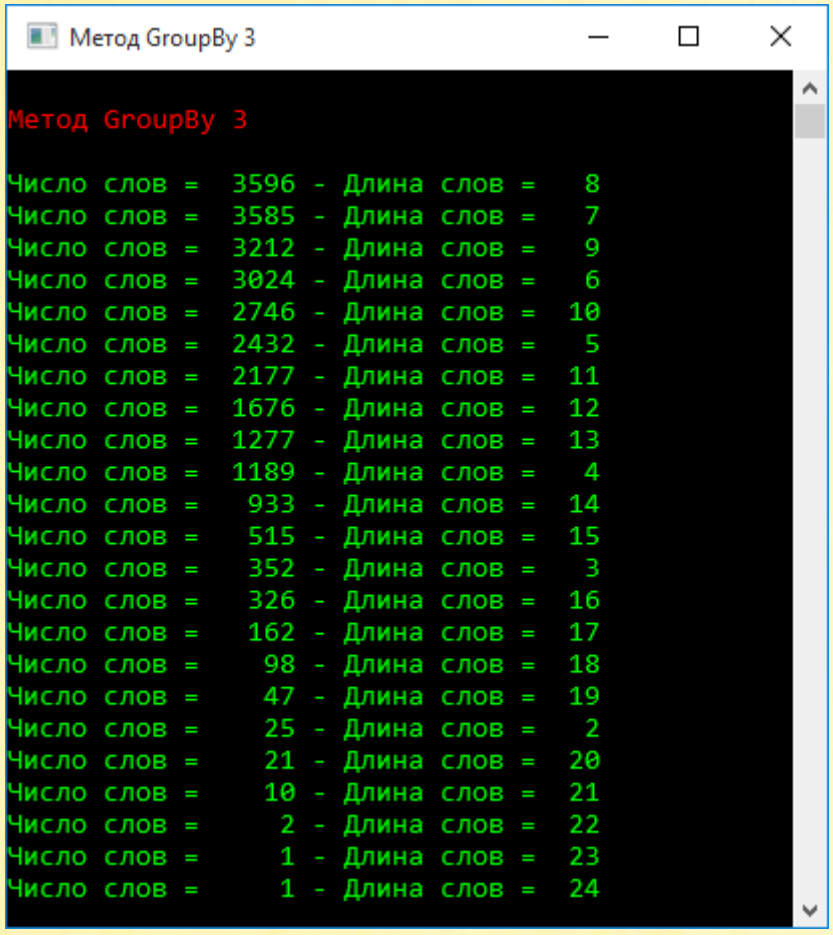
```
foreach var g in groups do
 begin
    //if (g.Key) then
        Println(g, NewLine);
 end;
```

Чтобы подсчитать число слов каждой длины, мы можем разбить их на группы по длине, а затем отсортировать группы по числу элементов:

```
 var lstStrAll := ReadAllLines('OSH-W97.txt');

 var groups := lstStrAll
               .GroupBy(s -> s.Length)
               .OrderByDescending(g -> g.Count());

 foreach var g in groups do
 begin
    Console.WriteLine('Число слов = {1} - Длина слов = {0}',
                      g.Key.ToString().PadLeft(3),
                      g.Count().ToString().PadLeft(5));
    //Print(g);
    //Console.WriteLine();
end;
```

Рисунок неопровержимо свидетельствует, что в русском языке больше всего существительных из 8 и 7 букв.

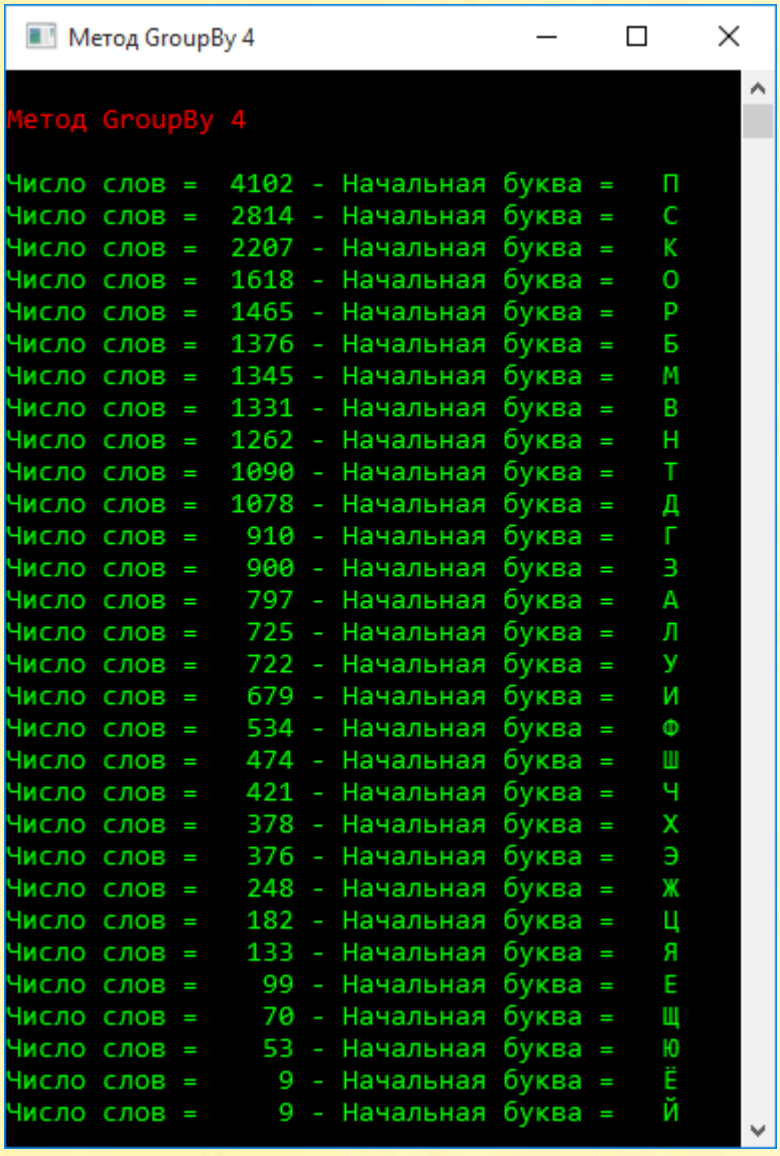
> Чтобы напечатать все группы слов, раскомментируйте строчки кода!



Изменив условие в методе *keySelector*, мы узнаем, сколько существительных начинается на каждую букву:

```
var lstStrAll := ReadAllLines('OSH-W97.txt');

var groups := lstStrAll
            .GroupBy(s -> s.First())
            .OrderByDescending(g -> g.Count());

 foreach var g in groups do
 begin
    Console.WriteLine('Число слов = {1} - Начальная буква = {0}',
                      g.Key.ToString().PadLeft(3),
                      g.Count().ToString().PadLeft(5));
    //Print(g);
    //Console.WriteLine();
end;
```

Оказывается, русские слова предпочитают начинаться на букву **П**:



Остальные методы GroupBy:

```
function GroupBy<Key>(keySelector: T->Key;
        comparer: System.Collections.Generic.IEqualityComparer<Key>):
                        IEnumerable<IGrouping<Key,T>>;
```

Как первый метод, но для сравнения ключей используется заданный компаратор проверки на равенство comparer.

```
function GroupBy<Key,Element>(keySelector: T->Key;
                              elementSelector: T->Element):
                              IEnumerable<IGrouping<Key,T>>;
```

Как первый метод, но к каждому исходному элементу применяется метод elementSelector . Для сравнения ключей используется компаратор проверки на равенство по умолчанию.

```
function GroupBy<Key,Element>(keySelector: T->Key;
                              elementSelector: T->Element;
                              comparer: IEqualityComparer<Key>):
                              IEnumerable<IGrouping<Key,Element>>;
```

Как предыдущий метод, но для сравнения ключей используется заданный компаратор проверки на равенство comparer.

```
function GroupBy<Key,Res>(keySelector: T->Key;
                          resultSelector: (Key,sequence of T)->Res):
                          sequence of Res;
```

Как первый метод, но результирующее значение для каждой группы вычисляет метод resultSelector. Для сравнения ключей используется компаратор проверки на равенство по умолчанию.

```
function GroupBy<Key,Element,Res>(keySelector: T->Key;
                                  elementSelector: T->Element;
                      resultSelector: (Key,sequence of Element)->Res):
                                  sequence of Res;
```

Как предыдущий метод, но для сравнения ключей используется заданный компаратор проверки на равенство comparer.

```
function GroupBy<Key,Res>(keySelector: T->Key;
                          resultSelector: (Key,sequence of T)->Res;
                          comparer: IEqualityComparer<Key>):
                          sequence of Res;
```

Как третий и пятый метод вместе взятые.

```
function GroupBy<Key,Element,Res>(keySelector: T->Key;
                                  elementSelector: System.T->Element;
                        resultSelector: (Key,sequence of Element)->Res;
                                  comparer: IEqualityComparer<Key>):
                                  sequence of Res;
```

Как предыдущий метод, но для сравнения ключей используется заданный компаратор проверки на равенство comparer.

Дополнительные методы расширения для последовательностей

Кроме методов расширения для последовательностей, позаимствованных у платформы *.NET, паскаль PascalABC.NET* имеет немало новых, собственных.

Методы *Print* и *Println*

Рассмотрены нами ранее.

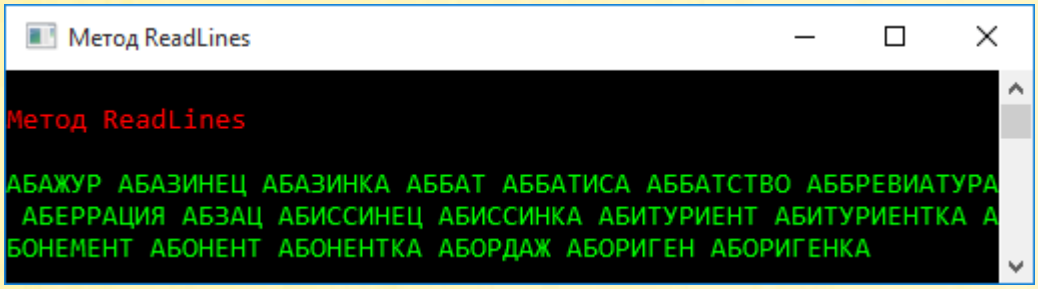
Метод *ReadLines*

Метод ReadLines возвращает последовательность строк из заданного файла:

```
function ReadLines(path: string): sequence of string;
```

Файл открывается и закрывается автоматически. Кодировка строк – *Windows*.

```
var sqStrAll := ReadLines('OSH-W97.txt')
                .Take(19)
                .Println;
```



Второй метод ReadLines действует так же, но дополнительно можно задать кодировку строк в файле:

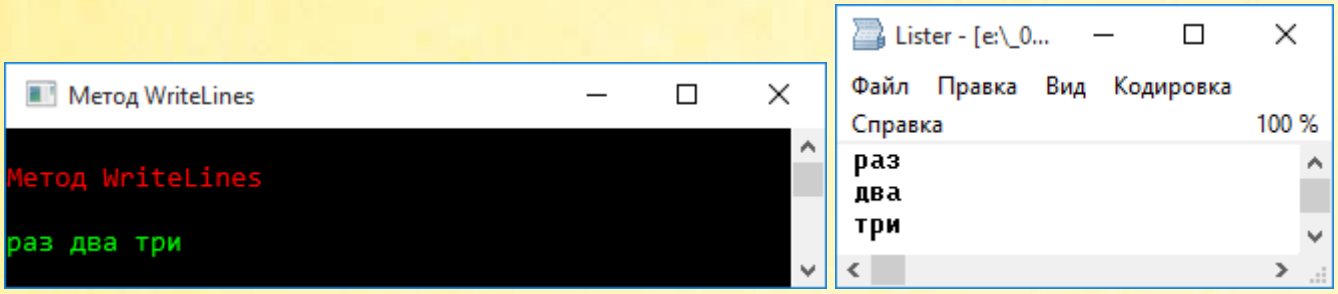
```
function ReadLines(path: string; en: Encoding): sequence of string;
```

Метод *WriteLines*

Метод WriteLines выводит строковую последовательность в заданный файл:

```
function WriteLines(Self: sequence of string; fname: string):
                    sequence of string; extensionmethod;
```

```
var sq := Seq('раз','два','три')
          .Println;
Println;
sq.WriteLines('Метод WriteLines.txt');
```



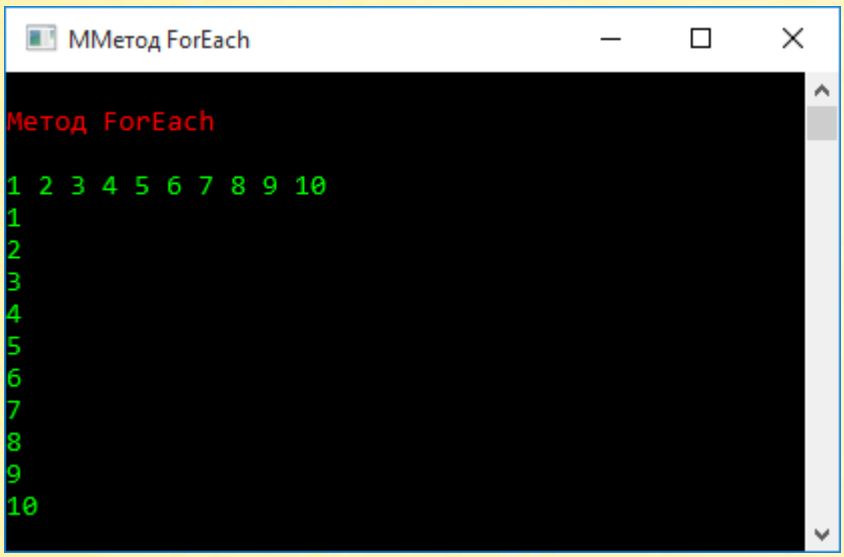
Метод *ForEach*

Метод ForEach – это процедура, поэтому он не возвращает элементов последовательности, а только *применяет к ним указанные действия*:

```
procedure ForEach(action: T->());
```

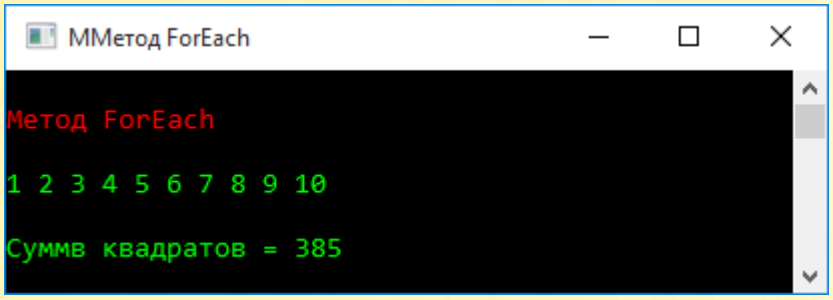
Например, мы можем напечатать все элементы последовательности:

```
var sq := Range(1,10).Println;
sq.ForEach(n -> Println(n));
```



Чтобы «вернуть» значение, необходимо использовать внешнюю переменную:

```
Println;
var summa:= 0;
sq.ForEach(procedure(n) -> summa := summa + n * n);
Println('Сумма квадратов = ' + summa);
```

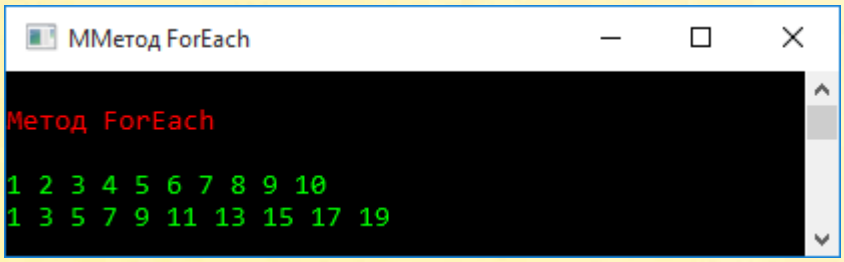


Второй метод ForEach применяет к элементам последовательности указанные действия с учётом *индекса*:

```
procedure ForEach<T>(Self: sequence of T; action: (T,integer) -> ());
                     extensionmethod;
```

Печатаем суммы элементов и их индексов в последовательности:

```
var sq := Range(1,10).Println;
sq.ForEach((n, i) -> Print(i+n));
```



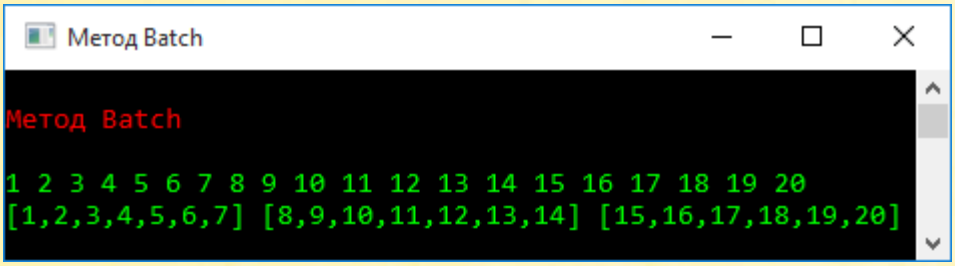
Метод *Batch*

Метод Batch разбивает последовательность на последовательности (серии) заданной длины *size*:

```
function Batch<T>(Self: sequence of T; size: integer):
                  sequence of sequence of T; extensionmethod;
```

Если число элементов в последовательности не кратно *size*, то в последнюю серию помещаются оставшиеся элементы:

```
var sq := Range(1, 20)
          .Println;
sq.Batch(7).Println;
```



Второй метод Batch разбивает последовательность на последовательности заданной длины *size* и применяет к каждой серии заданную функцию *proj*:

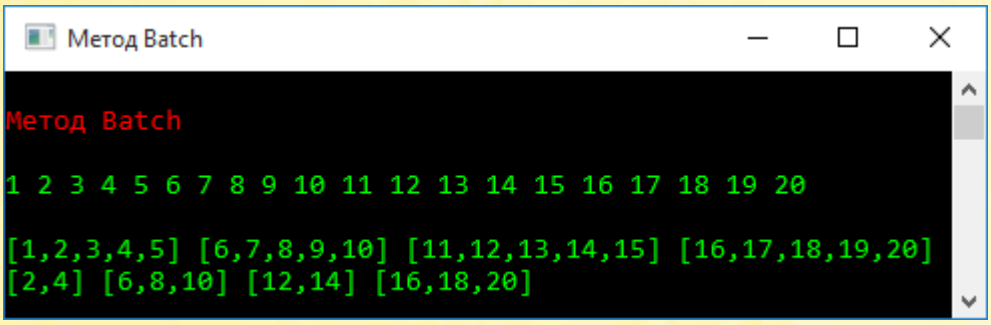
```
function Batch<T,Res>(Self: sequence of T; size: integer;
                      proj: Func<IEnumerable<T>,Res>):
                      sequence of Res; extensionmethod;
```

Так как каждая серия элементов – это последовательность, то к ней можно применить любой из рассмотренных выше методов. Например, отберём из каждой серии только чётные числа:

```
Println;
```

```
sq.Batch(5).Println;
sq.Batch(5, n -> n.Where(n -> n mod 2 = 0)).Println;
```

Мы получили 4 серии чётных чисел:

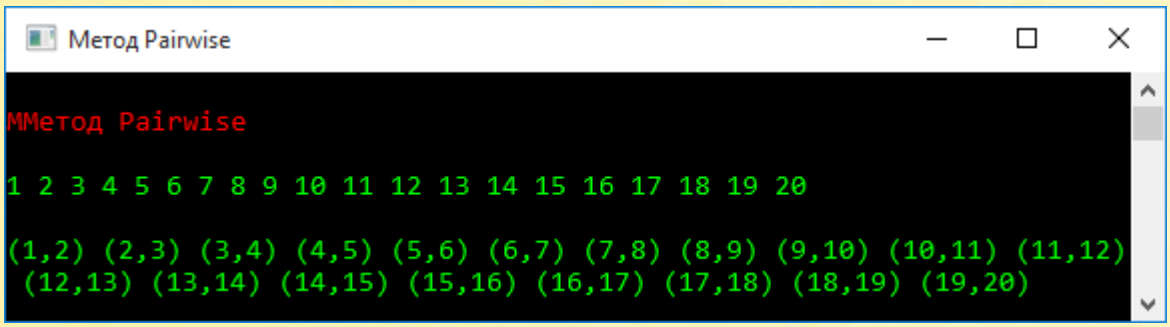


Метод *Pairwise*

Метод Pairwise возвращает последовательность кортежей, составленных изо всех пар соседних элементов:

```
function Pairwise<T>(Self: sequence of T): sequence of (T,T);
                     extensionmethod;
```

```
var sq := Range(1, 20)
          .Println;
Println;
sq.Pairwise.Println;
```

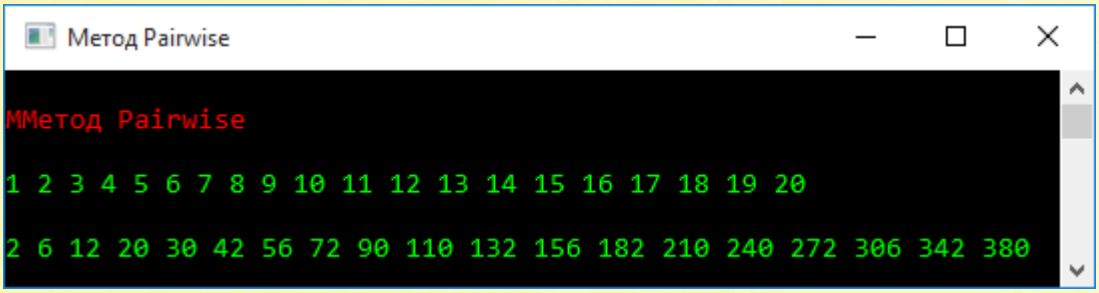


Второй метод Pairwise возвращает последовательность кортежей, составленных изо всех пар соседних элементов, и к каждому кортежу применяет заданную функцию *func*:

```
function Pairwise<T,Res>(Self: sequence of T; func:(T,T)->Res):
                         sequence of Res; extensionmethod;
```

Найдём произведение пар чисел в каждом кортеже:

```
sq.Pairwise((x,y) -> x*y).Println;
```



Метод *Partition*

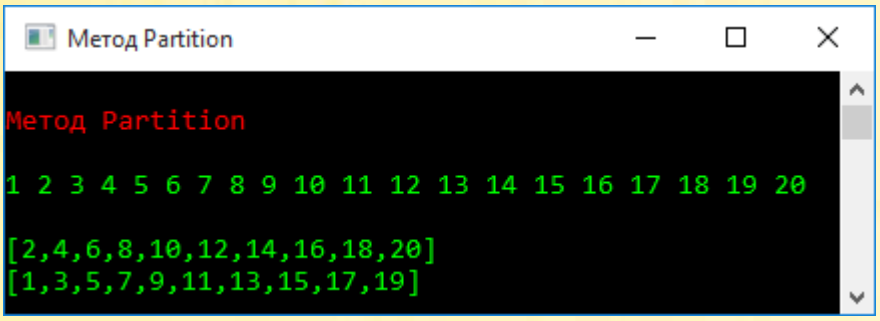
Метод Partition возвращает кортеж из двух последовательностей. Функция *cond* разбивает исходную последовательность на 2 части по заданному условию:

```
function Partition<T>(Self: sequence of T; cond: T->boolean):
                      (sequence of T,sequence of T); extensionmethod;
```

Разобьём числовую последовательность *sq* на 2 части так, чтобы в одной оказались чётные числа, а в другой - нечётные:

```
var sq := Range(1, 20)
          .Println;
Println;
var res := sq.Partition(n -> n mod 2 = 0);
```

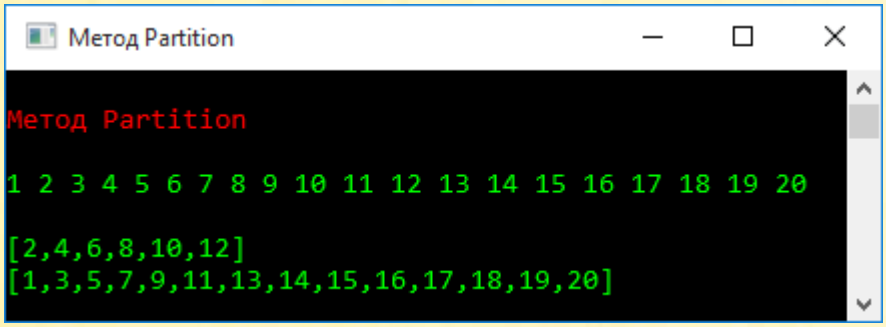
```
Println(res.Item1);
Println(res.Item2);
```



Второй метод Partition возвращает кортеж из двух последовательностей. Функция *cond* разбивает исходную последовательность на 2 части по заданному условию с учётом *индексов*:

```
function Partition<T>(Self: sequence of T;
                      cond: (T,integer)->boolean):
                     (sequence of T,sequence of T); extensionmethod;
```

Пусть в первой последовательности будут чётные числа, индексы которых меньше 12:



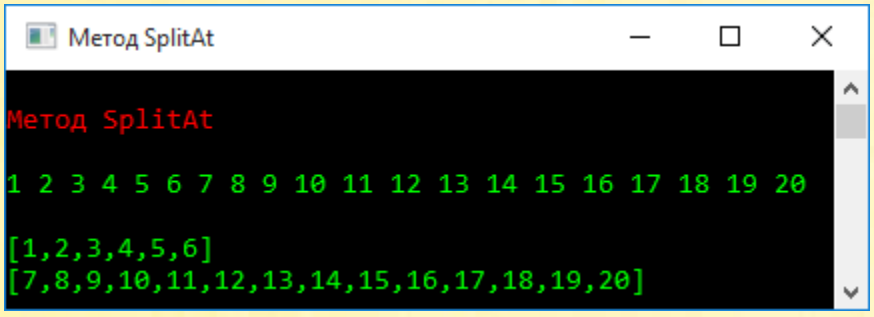

```
var sq := Range(1, 20)
          .Println;
Println;
var res := sq.Partition((n, i) -> (n mod 2 = 0) and (i < 12));

Println(res.Item1);
Println(res.Item2);
```

Метод *SplitAt*

Метод SplitAt возвращает кортеж из двух последовательностей. Параметр *ind* - это индекс элемента, начинающего вторую последовательность:

```
function SplitAt<T>(Self: sequence of T; ind: integer):
                    (sequence of T,sequence of T); extensionmethod;
```

```
var sq := Range(1, 20)
          .Println;
Println;
var res := sq.SplitAt(6);

Println(res.Item1);
Println(res.Item2);
```



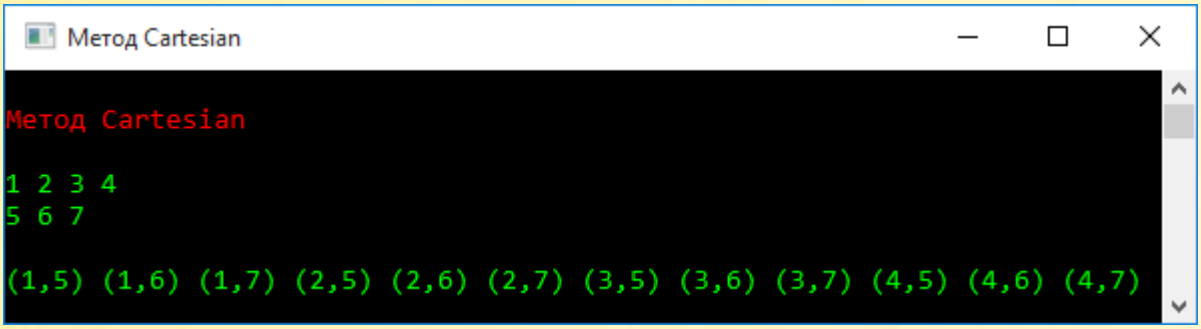
Метод *Cartesian*

Метод Cartesian возвращает *декартово произведение* двух последовательностей:

```
function Cartesian<T,T1>(Self: sequence of T; b: sequence of T1):
                         sequence of (T,T1); extensionmethod;
```

```
var sq1 := Range(1, 4)
           .Println;
var sq2 := Range(5, 7)
```

```
          .Println;
Println;
var res := sq1.Cartesian(sq2)
             .Println;
```

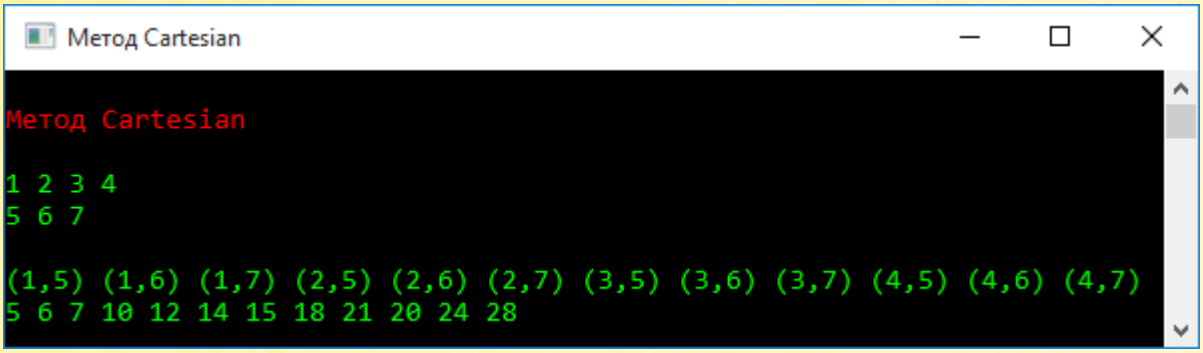


Второй метод Cartesian возвращает декартово произведение двух последовательностей. К каждой паре элементов в кортежах применяется функция *func*:

```
function Cartesian<T,T1,T2>(Self: sequence of T; b: sequence of T1;
                            func: (T,T1)->T2):
                            sequence of T2; extensionmethod;
```

Найдём произведение всех пар чисел в кортежах:

```
var sq1 := Range(1, 4)
          .Println;
var sq2 := Range(5, 7)
          .Println;
Println;
sq1.Cartesian(sq2).Println;
var res := sq1.Cartesian(sq2, (x,y) -> x * y)
             .Println;
```



Декартово произведение двух последовательностей и цикл *foreach* вполне могут заменить традиционные вложенные циклы *for*.

Сколько мне лет? (Range.Cartesian)

Дядя Алёша вдвое старше меня, а цифры числа, выражающего мой возраст, равны сумме и разности цифр его возраста.

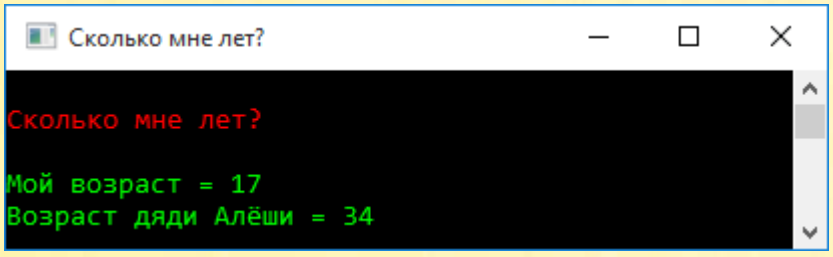
Сколько мне лет?

Из условия задачи следует, что мне (автору задачи) не меньше 10 лет. Чтобы решить задачу, достаточно **перебирать** цифры, составляющие мой возраст и сверяться с текстом задачи:

```
uses
   System;


begin
   // заголовок окна:
   Console.Title := 'Сколько мне лет?';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Сколько мне лет?');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();
```

```
   // цифры:
   var digits := Range(0, 10);

   // решаем задачу:
   foreach var t in digits.Cartesian(digits) do
   begin
      var a := t.Item1;
      var b := t.Item2;
      var i := a * 10 + b;
      // мой возраст - двузначное число:
      if (i < 10) then
         continue;
      var onkel := i * 2;
      var dig1 := onkel div 10;
      var dig2 := onkel mod 10;
      if (((dig1 + dig2 = a) and
         (Abs(dig1 - dig2) = b)) or
         ((dig1 + dig2 = b) and
         (Abs(dig1 - dig2) = a))) then
      begin
         Writeln('Мой возраст = ' + i);
         Writeln('Возраст дяди Алёши = ' + onkel);
      end
   end;

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Задача несложная, но для упражнения в программировании вполне годится:



Великолепная семёрка (Range.Cartesian)

В журнале *Квантик* №10 за 2015 год напечатана такая конкурсная задача:

46. Хулиган Семён, любимое число которого – семь, забрался ночью через окно в гостиницу «Караван-Сарай» и с дверей всех номеров снял семёрки. Утром Семёна поймал полицейский Пронькин, который заявил, что за каждую снятую цифру полагается платить штраф в размере одного доллара. Сколько долларов придётся заплатить Семёну, если в гостинице 1000 номеров и они нумеруются подряд, начиная с 1?

Заметим, что в числе 1000 семёрок нет, значит, нам достаточно проверить числа от 1 до 999. Чтобы не выделять цифры из чисел, мы найдём декартово произведение всех цифр от 0 до 9:

```
// цифры:
var digits := Range(0, 9);
var res := digits.Cartesian(digits);
var cart3 := res.Cartesian(digits);
```

Теперь мы можем перебирать все комбинации из трёх цифр в цикле foreach:

```
foreach var t in cart3 do
begin
```

Текущие значения элементов кортежа дадут нам цифры очередного числа:

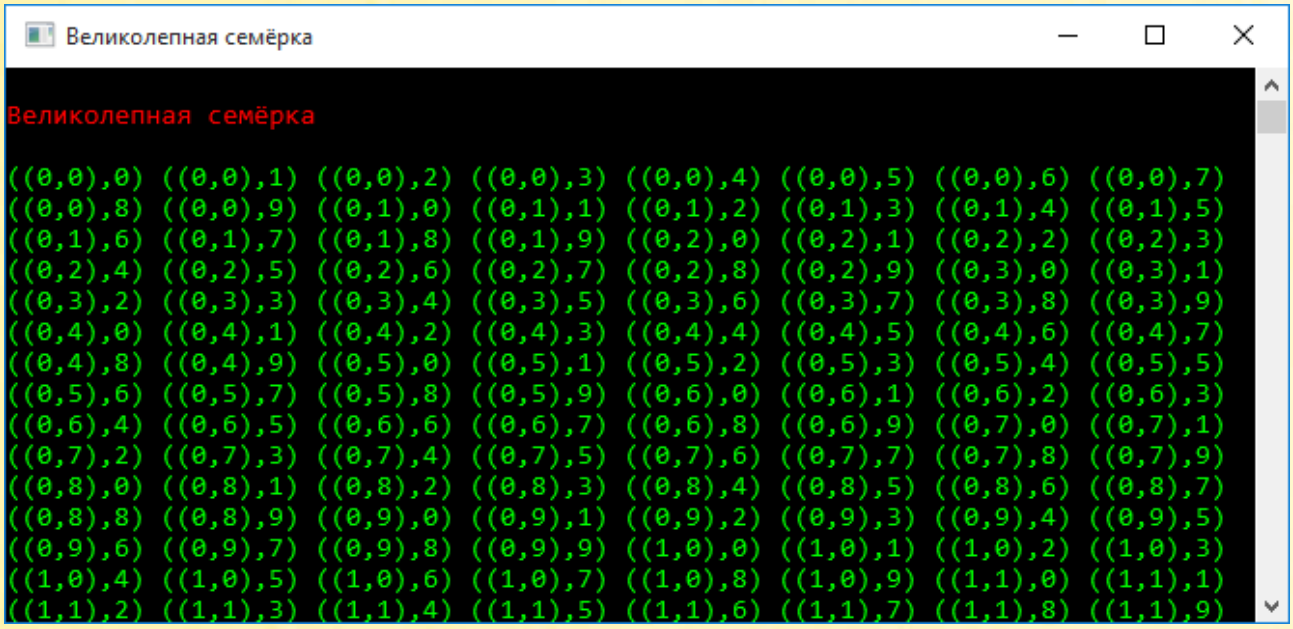
```
var a := t.Item1.Item1;
```

```
var b := t.Item1.Item2;
var c := t.Item2;
```

Каждый кортеж состоит из двухэлементного кортежа и одиночного элемента:

((0,0), 0)

Вот так выглядит часть последовательности кортежей:



Item1 – это двухэлементный кортеж. Первый элемент этого кортежа равен *Item1. Item1*, а второй - *Item1. Item2*.

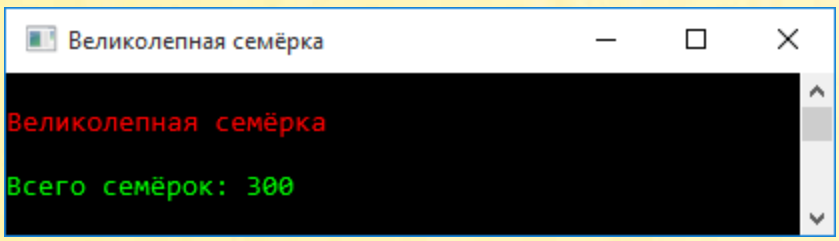
Item2 – одиночный элемент.

Если какая-либо из цифр – **семёрка**, то мы увеличиваем на единицу счётчик семёрок n7. И наконец, для сокращения кода мы используем 3 условных оператора ?: - по одному для каждого элемента кортежа:

```
uses
   System;


begin
```

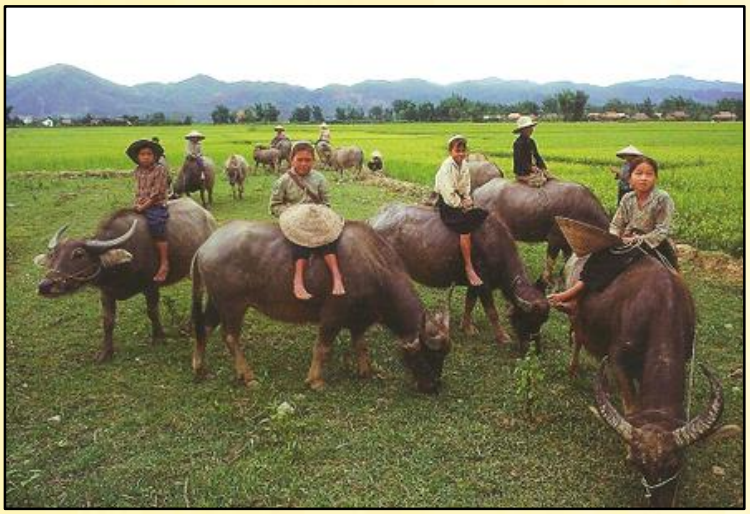
```
    // заголовок окна:
    Console.Title := 'Великолепная семёрка';
    Console.WriteLine('');
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Великолепная семёрка');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    // цифры:
    var digits := Range(0, 9);
    var res := digits.Cartesian(digits);
    var cart3 := res.Cartesian(digits);

    //решаем задачу

    //число семёрок:
    var n7 := 0;
    foreach var t in cart3 do
    begin
       var a := t.Item1.Item1;
       var b := t.Item1.Item2;
       var c := t.Item2;
       var s := 0;
       s += a = 7 ? 1 : 0;
       s += b = 7 ? 1 : 0;
       s += c = 7 ? 1 : 0;
       n7 += s;
    end;
    Println('Всего семёрок: ' + n7);

    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
end.
```

Всё готово! Запускаем программу и тут же получаем точный и круглый ответ – в числах от 1 до 1000 семёрка встречается ровно 300 раз:



Вьетнамские буйволы (Range.Cartesian)

Вот очень старая вьетнамская задача:



Для кормления 100 буйволов заготовили 100 охапок сена.

Стоящий молодой буйвол съедает 5 охапок сена.
Лежащий молодой буйвол съедает 3 охапки сена.
Старые буйволы втроём съедают 1 охапку сена.

Сколько молодых буйволов стоят, сколько лежат и сколько буйволов старых?

Решение задачи с буйволами не очень сильно отличается от решения предыдущей.

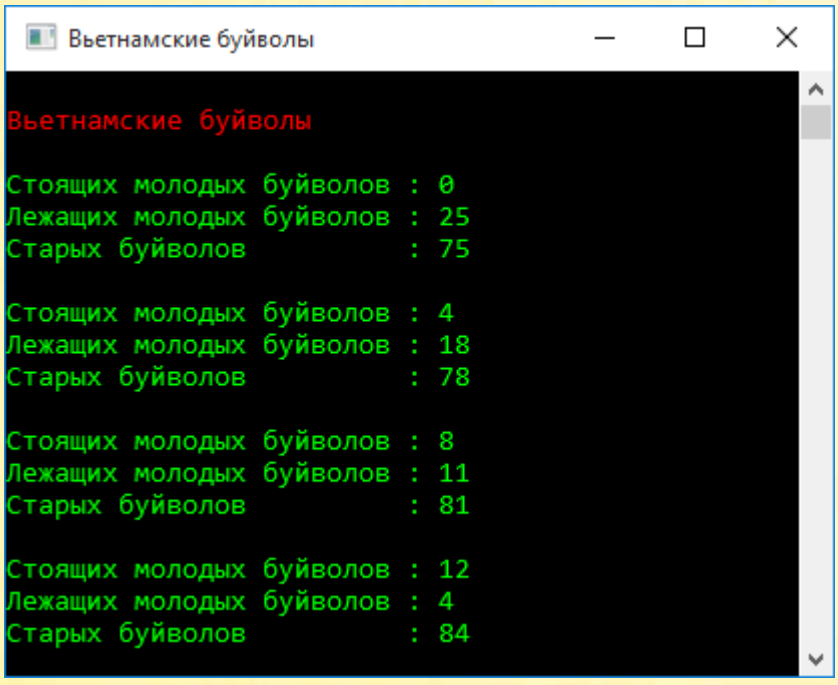
Буйволов каждого вида не меньше нуля и не больше (100 поделить на число съедаемых охапок сена). Некоторую нервозность вносят старые буйволы, которых следует считать тройками.

Когда общее число буйволов и съедаемых охапок сена будет по 100, мы напечатаем ответ:

```
uses
   System;


begin
   // заголовок окна:
   Console.Title := 'Вьетнамские буйволы';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Вьетнамские буйволы');
   Console.ForegroundColor := ConsoleColor.Green;
```

```
   Console.WriteLine();

   // буйволы:
   var sqBuffaloS := Range(0, 100 div 5);
   var sqBuffaloSL := sqBuffaloS.Cartesian(Range(0, 100 div 3));
   var cart3 := sqBuffaloSL.Cartesian(Range(0, 100 div 3));

   // решаем задачу:
   foreach var t in cart3 do
   begin
      var buffaloS := t.Item1.Item1;
      var buffaloL := t.Item1.Item2;
      var bO := t.Item2;
      var buffaloO := bO * 3;
      if ((buffaloS + buffaloL + buffaloO = 100) and
               (buffaloS * 5 + buffaloL * 3 +
                buffaloO div 3 = 100)) then
      begin
         Println('Стоящих молодых буйволов : '
                  + buffaloS);
         Println('Лежащих молодых буйволов : '
                  + buffaloL);
         Println('Старых буйволов          : '
                  + buffaloO);
         Println;
      end
   end;

   Console.Write-
Line();
   Console.Fore-
groundColor := Con-
soleColor.Red;
end.
```

Задача имеет 4 решения:



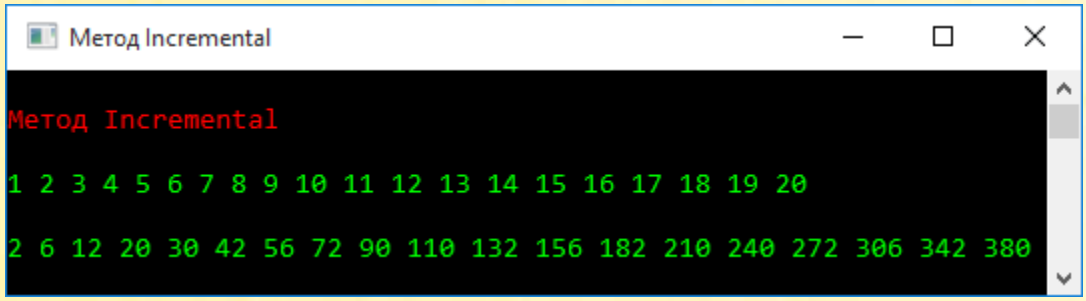
Метод *Incremental*

Метод Incremental возвращает последовательность, составленную изо всех пар соседних элементов исходной последовательности, к которым применена функция *func*:

```
function Incremental<T,T1>(Self: sequence of T; func: (T,T)->T1):
                           sequence of T1; extensionmethod;
```

Найдём произведения всех пар соседних элементов последовательности *sq*:

```
var sq := Range(1, 20)
          .Println;
Println;
var res := sq.Incremental((x,y) -> x*y)
             .Println;
```

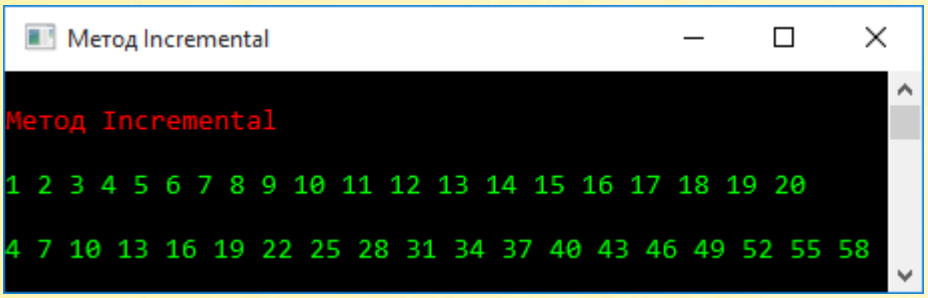


Второй метод Incremental возвращает последовательность, составленную изо всех пар соседних элементов исходной последовательности, к которым применена функция *func* с учётом *индекса пары* (отсчёт начинается с 1):

```
function Incremental<T,T1>(Self: sequence of T;
                           func: (T,T,integer)->T1):
                           sequence of T1; extensionmethod;
```

Найдём сумму всех соседних пар чисел и индекса пары:

```
var sq := Range(1, 20)
          .Println;
Println;
var res := sq.Incremental((x,y,n) -> x + y + n)
             .Println;
```



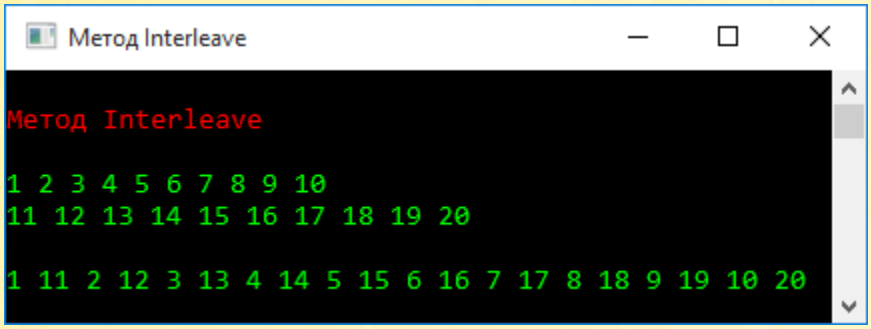
Метод *Interleave*

Метод Interleave возвращает последовательность, составленную из чередующихся элементов *двух* заданных последовательностей:

```
function Interleave<T>(Self: sequence of T; a: sequence of T):
                       sequence of T; extensionmethod;
```

Результирующая последовательность заканчивается, когда в более короткой последовательности не останется элементов.

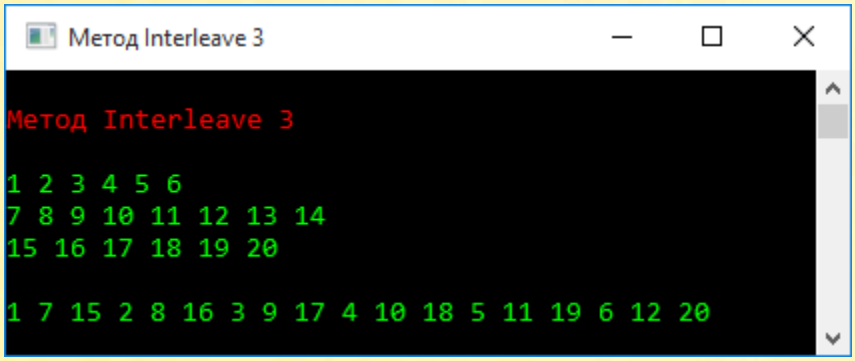
```
var sq1 := Range(1, 10)
           .Println;
var sq2 := Range(11, 20)
           .Println;
Println;
sq1.Interleave(sq2)
   .Println;
```



Второй метод Interleave возвращает последовательность, составленную из чередующихся элементов *трёх* заданных последовательностей:

```
function Interleave<T>(Self: sequence of T; a,b: sequence of T):
                       sequence of T; extensionmethod;
```

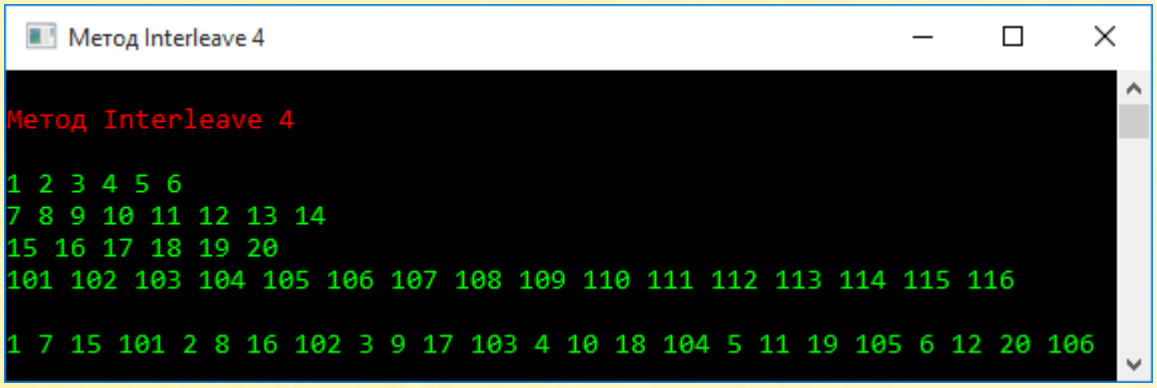
```
var sq1 := Range(1, 6)
           .Println;
var sq2 := Range(7, 14)
           .Println;
var sq3 := Range(15, 20)
           .Println;
Println;
sq1.Interleave(sq2, sq3)
   .Println;
```



Третий метод Interleave возвращает последовательность, составленную из чередующихся элементов *четырёх* заданных последовательностей:

```
function Interleave<T>(Self: sequence of T; a,b,c: sequence of T): se-
quence of T; extensionmethod;
```

```
var sq1 := Range(1, 6)
           .Println;
var sq2 := Range(7, 14)
           .Println;
var sq3 := Range(15, 20)
           .Println;
var sq4 := Range(101, 116)
           .Println;

Println;
sq1.Interleave(sq2, sq3,sq4 )
   .Println;
```

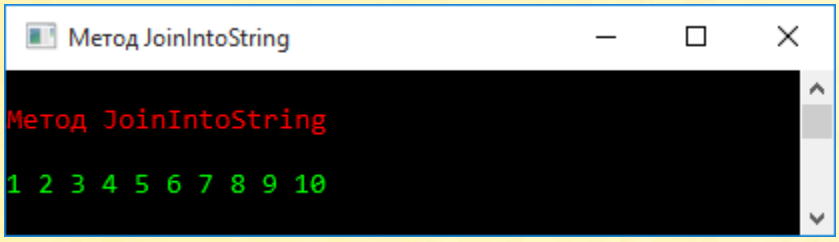


Метод *JoinIntoString*

Метод JoinIntoString без параметра возвращает строку, составленную из строкового представления элементов последовательности. Разделителем элементов служит пробел.

```
function JoinIntoString<T>(Self: sequence of T): string;
                           extensionmethod;
```

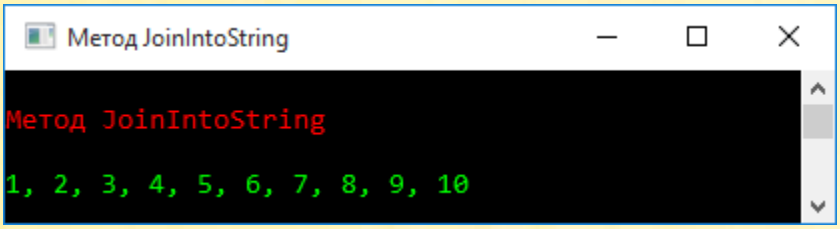
```
var sq := Range(1, 10);
sq.JoinIntoString.Println;
```



Метод JoinIntoString с параметром возвращает строку, составленную из строкового представления элементов последовательности. Разделителем элементов служит строка *delim*.

```
function JoinIntoString<T>(Self: sequence of T; delim: string): string;
                                                    extensionmethod;
```

```
var sq := Range(1, 10);
sq.JoinIntoString(', ').Println;
```



Сотая цифра (Range. JoinIntoString)

Задача 300 из книги *Математическая шкатулка* **[Нагибин88]**, страница 45:

> Все натуральные числа, начиная с 1, записаны в порядке их возрастания: 1 2 3 4 5 6 7 8 9 10 11...
>
> Какая цифра в этой записи стоит на сотом месте?

Проще всего решить задачу «строковым» способом.

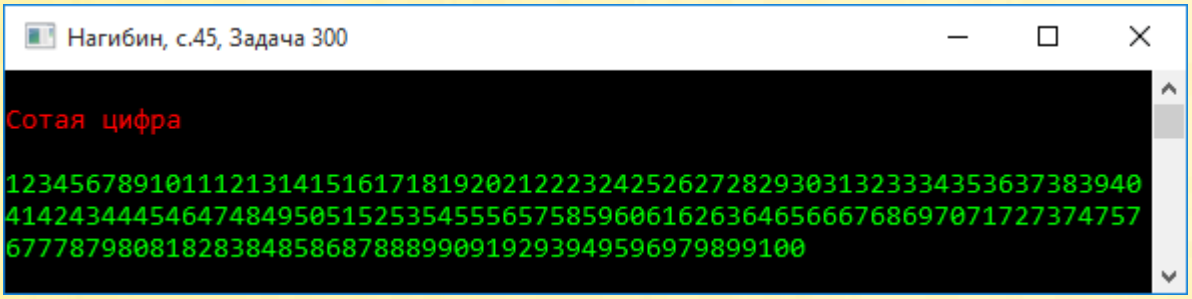
Мы не знаем, сколько чисел нам понадобится, поэтому в методе **Range** указываем заведомо большее число элементов в последовательности:

```
Range(1,100)
```

Так мы получим последовательность чисел от 1 до 100, а нам нужна **строка**. Метод расширения **JoinIntoString** объединяет все элементы в одну строку, но по умолчанию ставит между ними пробелы. Нам нужно записать все числа **вплотную** друг за другом, поэтому в скобках указываем пустую строку:

```
JoinIntoString('')
```

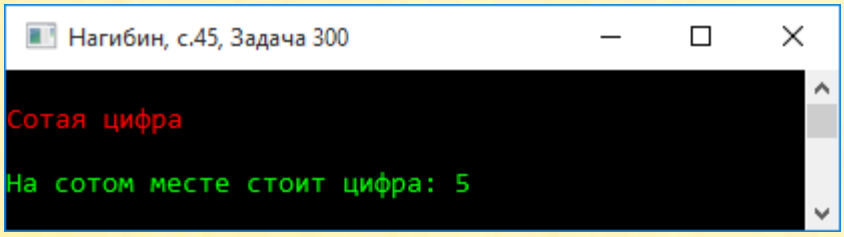
В итоге мы получим длинную строку из последовательных натуральных чисел:



Выделяем из неё сотую цифру – и задача решена:

```
uses
   System;

// Нагибин, с.45, Задача 300


begin
   // заголовок окна:
   Console.Title := 'Нагибин, с.45, Задача 300';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Сотая цифра');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   // решаем задачу:
   var res:= Range(1,100).JoinIntoString('');
   // печатаем ответ:
   Console.WriteLine('На сотом месте стоит цифра: {0}', res[100]);

   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

На рисунке вы видите, что сотое место в строке занимает цифра 5:



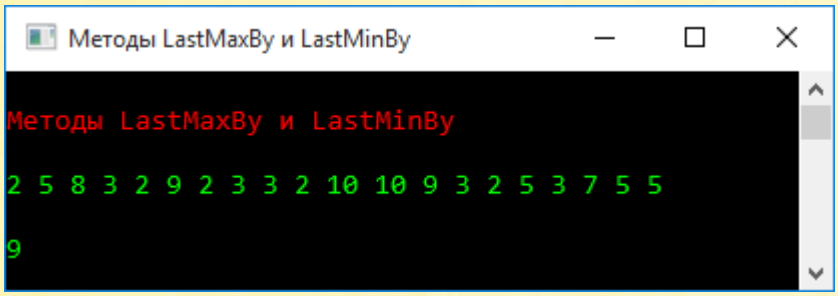
Решение «честным», числовым способом значительно сложнее!

Методы *LastMaxBy* и *LastMinBy*

Метод LastMaxBy возвращает последний элемент последовательности с *максимальным* значением ключа:

```
function LastMaxBy<T, TKey>(Self: sequence of T; selector: T -> TKey): T; extensionmethod;
```

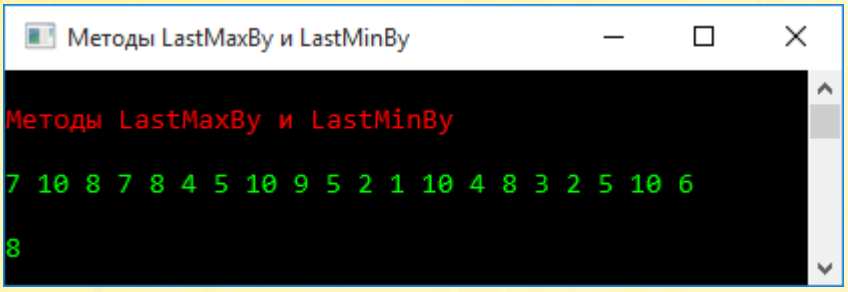
```
var sq := SeqRandom(20, 1, 10)
          .ToArray
          .Println;
Println;
var res := sq.LastMaxBy(n -> n > 8);
Println(res);
```



Метод LastMinBy возвращает последний элемент последовательности с *минимальным* значением ключа:

```
function LastMinBy<T, TKey>(Self: sequence of T;
                            selector: T -> TKey): T; extensionmethod;
```

```
var sq := SeqRandom(20, 1, 10)
          .ToArray
          .Println;
Println;
var res := sq.LastMinBy(n -> n mod 4);
Println(res);
```

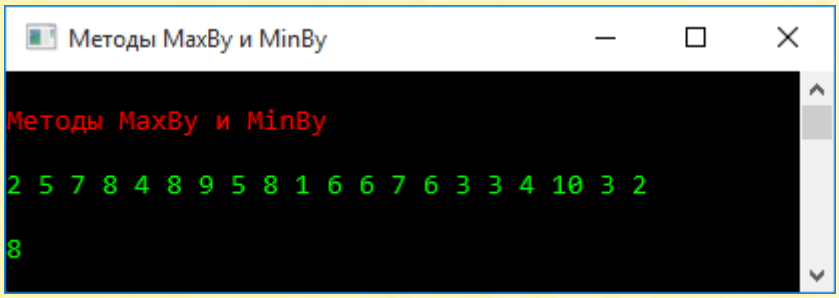


Методы *MaxBy* и *MinBy*

Метод MaxBy возвращает первый элемент последовательности с *максимальным* значением ключа:

```
function MaxBy<T, TKey>(Self: sequence of T;
                        selector: T -> TKey): T; extensionmethod;
```

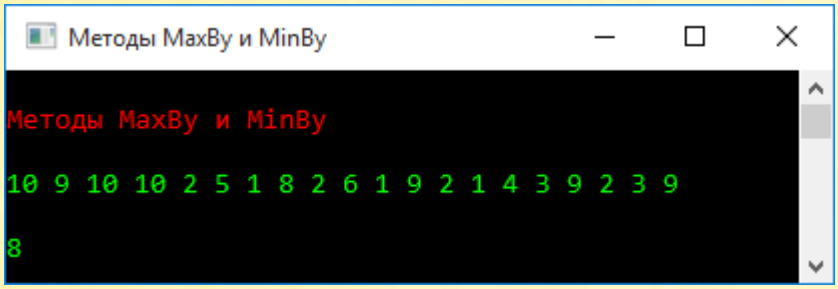
```
var sq := SeqRandom(20, 1, 10)
          .ToArray
          .Println;
Println;
var res := sq.MaxBy(n -> n > 7);
Println(res);
```



Метод MinBy возвращает первый элемент последовательности с *минимальным* значением ключа:

```
function MinBy<T, TKey>(Self: sequence of T;
                        selector: T -> TKey): T; extensionmethod;
```

```
var sq := SeqRandom(20, 1, 10)
          .ToArray
          .Println;
Println;
var res := sq.MinBy(n -> n mod 4);
Println(res);
```

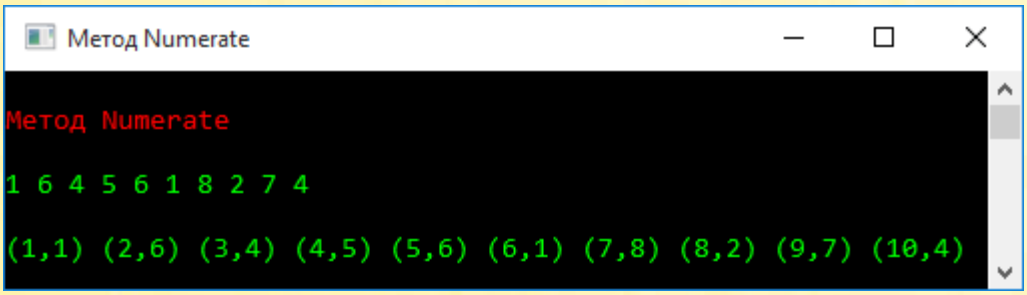


Метод *Numerate*

Метод Numerate возвращает последовательность кортежей. Первый элемент кортежа – номер элемента последовательности (счёт с 1), второй – сам элемент последовательности:

```
function Numerate<T>(Self: sequence of T): sequence of (integer,T);
                     extensionmethod;
```

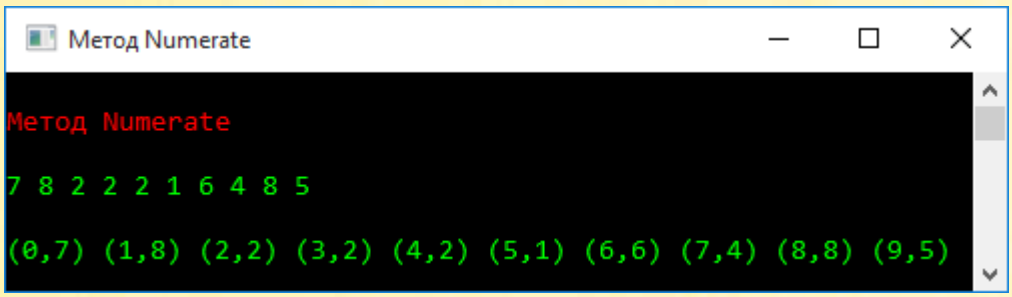
```
var sq := SeqRandom(10, 1, 10)
          .ToArray
          .Println;
Println;
var res := sq.Numerate().Println;
```



Второй метод Numerate возвращает последовательность кортежей. Первый элемент кортежа – номер элемента последовательности (счёт с *from*), второй – сам элемент последовательности:

```
function Numerate<T>(Self: sequence of T; from: integer):
                     sequence of (integer,T); extensionmethod;
```

```
var sq := SeqRandom(10, 1, 10)
          .ToArray
          .Println;
Println;
var res := sq.Numerate(0).Println;
```



Методы *Print* и *Println*

Эти методы мы уже рассмотрели раньше.

Метод *SkipLast*

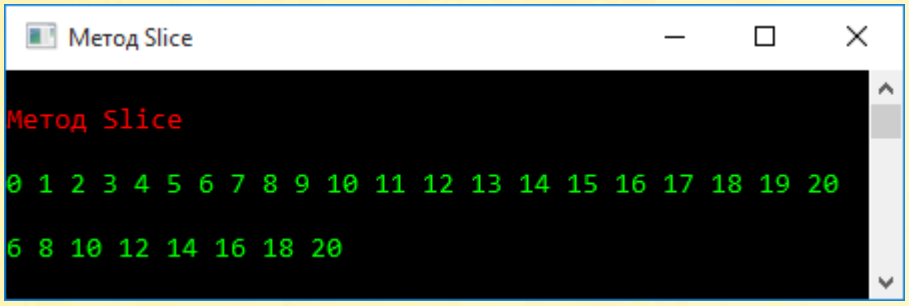
Этот метод мы уже рассмотрели раньше.

Метод *Slice*

Метод Slice возвращает элементы последовательности, начиная с индекса *from,* с шагом *step*:

```
function Slice<T>(Self: sequence of T; from,step: integer):
                  sequence of T; extensionmethod;
```

```
var sq := Range(0, 20)
          .Println;
Println;
sq.Slice(6, 2).Println;;
```

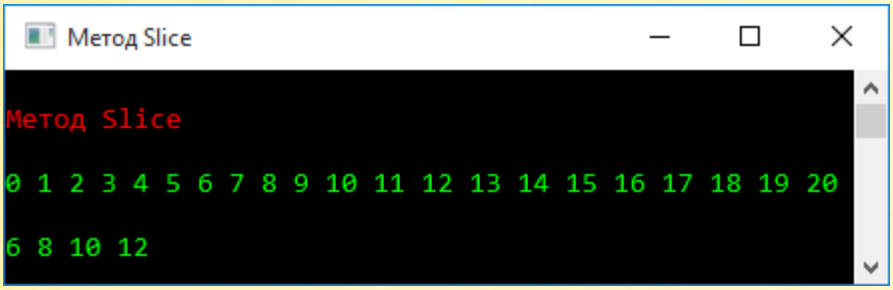


Шаг должен быть больше нуля!

Второй метод Slice возвращает элементы последовательности, начиная с индекса *from,* с шагом *step,* но не более *count*:

```
function Slice<T>(Self: sequence of T; from,step,count: integer):
                  sequence of T; extensionmethod;
```

```
var sq := Range(0, 20)
          .Println;
Println;
sq.Slice(6, 2, 4).Println;
```



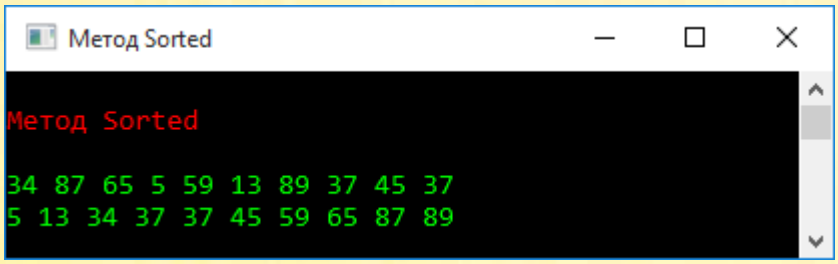
Методы *Sorted* и *SortedDescending*

Метод Sorted возвращает последовательность, отсортированную в *лексикографическом* порядке (по возрастанию):

```
function Sorted<T>(Self: sequence of T): sequence of T; extensionmethod;
```

Отсортируем случайную последовательность. Чтобы «зафиксировать» значения, преобразуем её в массив:

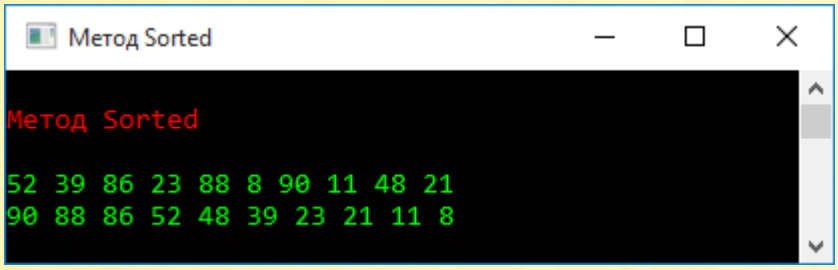
```
var sq := SeqRandom.ToArray.Println;
sq.Sorted.Println;
```



Метод SortedDescending возвращает последовательность, отсортированную в *антилексикографическом* порядке (по убыванию):

```
function SortedDescending<T>(Self: sequence of T):
                              sequence of T; extensionmethod;
```

```
var sq := SeqRandom.ToArray.Println;
sq.SortedDescending.Println;
```

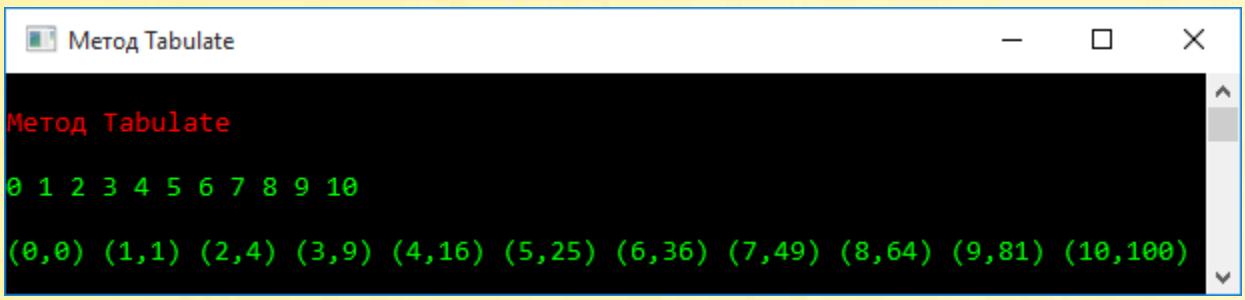


Метод *Tabulate*

Метод Tabulate возвращает последовательность кортежей. Первый элемент кортежа – элемент последовательности, второй – значение заданной функции для этого элемента:

```
function Tabulate<T,T1>(Self: sequence of T; F: T->T1):
                        sequence of (T,T1); extensionmethod;
```

```
var sq := Range(0, 10)
          .Println;
Println;
sq.Tabulate(x -> x * x).Println;
```



Метод *TakeLast*

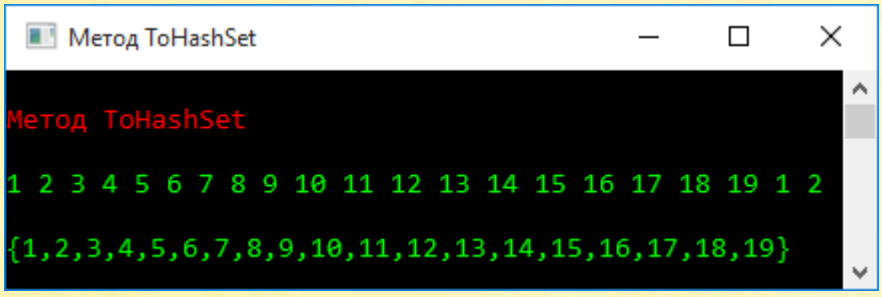
Этот метод мы уже рассмотрели раньше.

Метод *ToHashSet*

Метод ToHashSet возвращает множество типа *HashSet* из элементов последовательности:

```
function ToHashSet<T>(Self: sequence of T): HashSet<T>; extensionmethod;
```

```
var sq := (Range(1, 19) + Seq(1,2))
          .Println;
Println;
var hs:= sq.ToHashSet;
Println(hs);
```

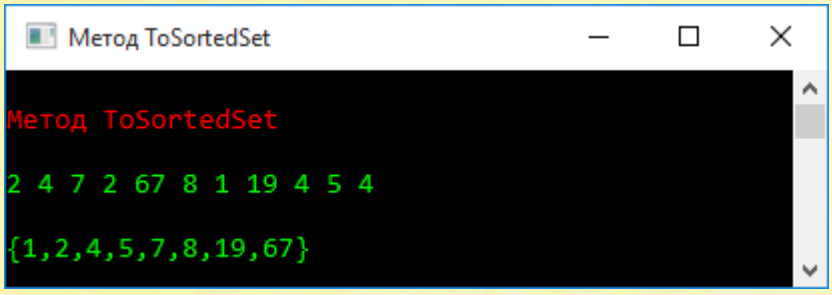


Метод *ToSortedSet*

Метод ToSortedSet возвращает множество типа *SortedSet* из элементов последовательности:

```
function ToSortedSet<T>(Self: sequence of T): SortedSet<T>; extension-
method;
```

```
var sq := Seq(2, 4, 7, 2, 67, 8, 1, 19, 4, 5, 4)
          .Println;
Println;
var ss:= sq.ToSortedSet;
Println(ss);
```

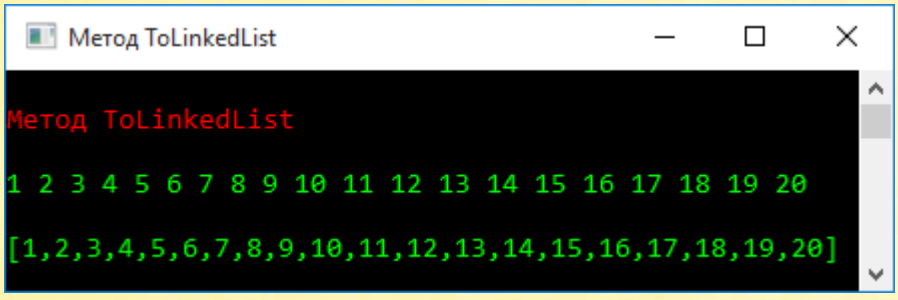


Метод *ToLinkedList*

Метод ToLinkedList возвращает связный список типа *LinkedList* из элементов последовательности:

```
function ToLinkedList<T>(Self: sequence of T): LinkedList<T>;
                         extensionmethod;
```

```
var sq := Range(1, 20)
          .Println;
Println;
var ll:= sq.ToLinkedList;
Println(ll);
```



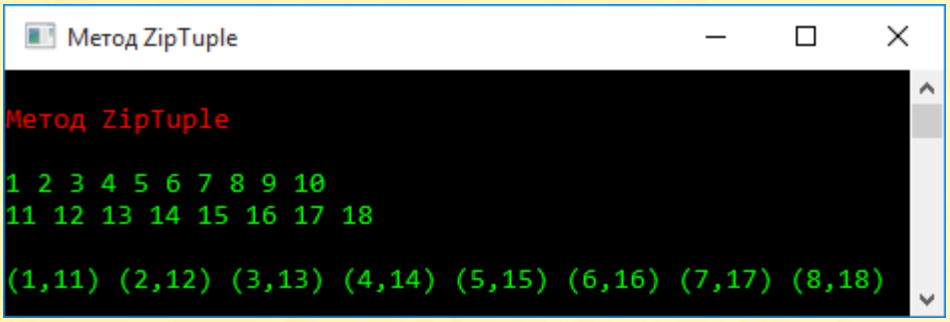
Метод *ZipTuple*

Метод ZipTuple возвращает последовательность *двухэлементных* кортежей. Первый элемент кортежа берётся из первой последовательности, а второй – из второй – с тем же индексом.

```
function ZipTuple<T,T1>(Self: sequence of T; a: sequence of T1):
                        sequence of (T,T1); extensionmethod;
```

```
var sq1 := Range(1, 10)
           .Println;
var sq2 := Range(11, 18)
           .Println;
Println;
sq1.ZipTuple(sq2)
   .Println;
```

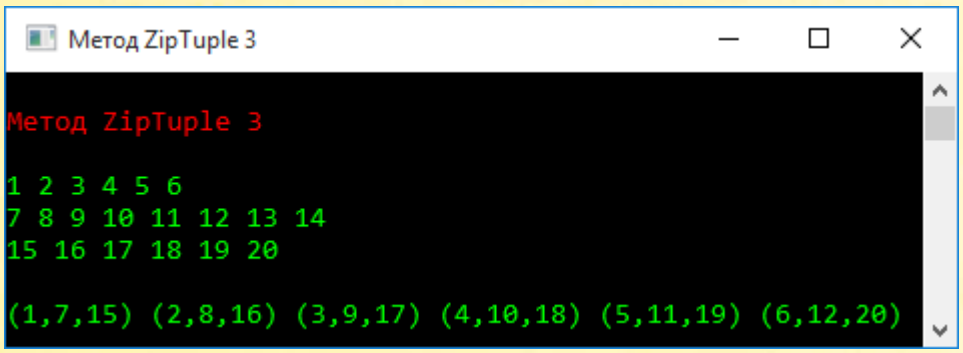
В результирующей последовательности столько элементов, сколько их в более короткой исходной последовательности:



Второй метод ZipTuple возвращает последовательность *трёхэлементных* кортежей. Первый элемент кортежа берётся из первой последовательности, второй – из второй, а третий - из третьей. Все элементы имеют одинаковый индекс:

```
function ZipTuple<T,T1,T2>(Self: sequence of T;
                           a: sequence of T1; b: sequence of T2):
                           sequence of (T,T1,T2); extensionmethod;
```

```
var sq1 := Range(1, 6)
           .Println;
var sq2 := Range(7, 14)
           .Println;
var sq3 := Range(15, 20)
           .Println;
Println;
sq1.ZipTuple(sq2, sq3)
   .Println;
```

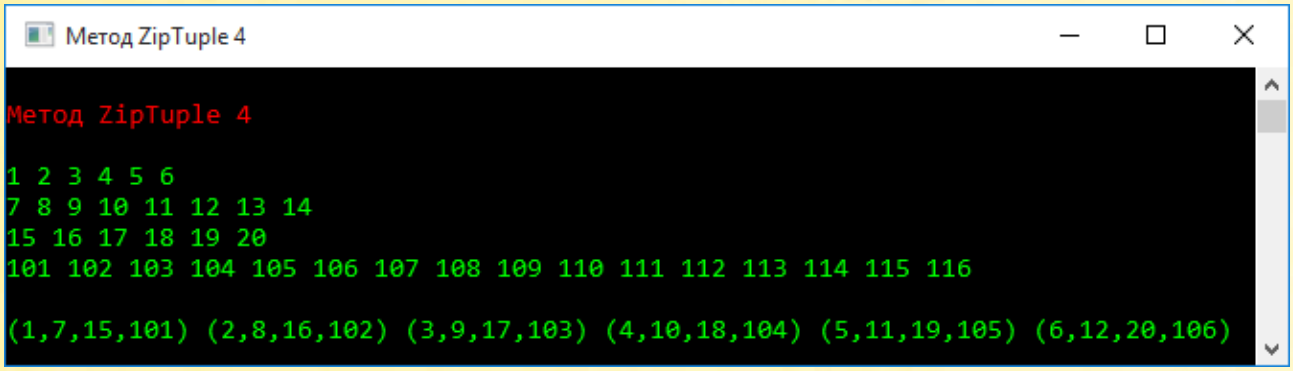


Третий метод ZipTuple возвращает последовательность *четырёхэлементных* кортежей. Первый элемент кортежа берётся из первой последовательности, второй – из второй, третий - из третьей, а четвёртый - из четвёртой. Все элементы имеют одинаковый индекс:

```
function ZipTuple<T,T1,T2,T3>(Self: sequence of T;
                              a: sequence of T1;
                              b: sequence of T2;
                              c: sequence of T3):
                              sequence of (T,T1,T2,T3);
                              extensionmethod;
```

```
var sq1 := Range(1, 6)
           .Println;
var sq2 := Range(7, 14)
           .Println;
var sq3 := Range(15, 20)
           .Println;
var sq4 := Range(101, 116)
           .Println;
```

```
Println;
sq1.ZipTuple(sq2, sq3,sq4 )
   .Println;
```

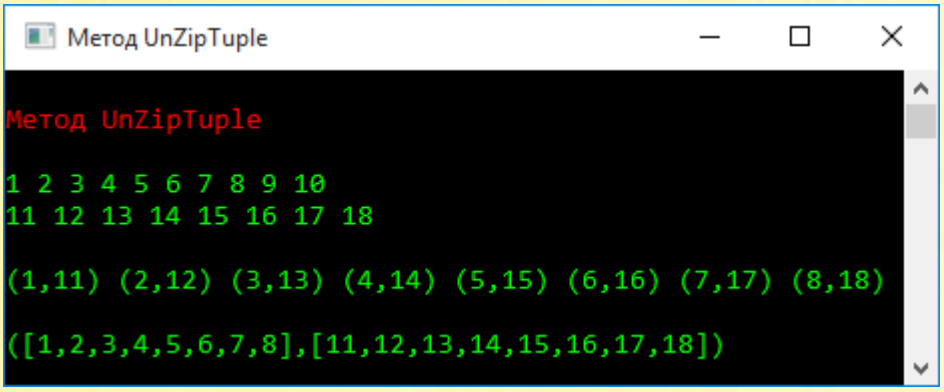


Метод *UnZipTuple*

Метод UnZipTuple возвращает *две* последовательности, созданные из последовательности *двухэлементных* кортежей. Первые элементы кортежей образуют первую последовательность, а вторые – вторую:

```
function UnZipTuple<T,T1>(Self: sequence of (T,T1)):
                          (sequence of T,sequence of T1); extensionmethod
```

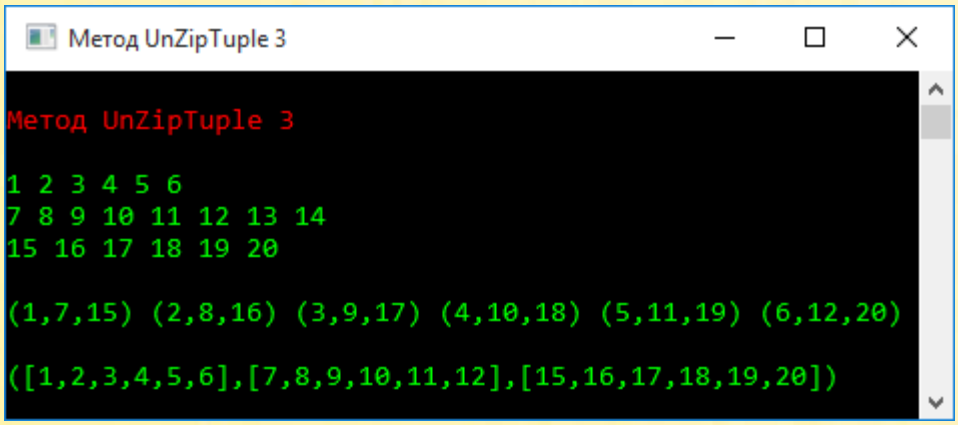
```
var sq1 := Range(1, 10)
           .Println;
var sq2 := Range(11, 18)
           .Println;
Println;
var zip:= sq1.ZipTuple(sq2)
          .Println;

Println;
var unzip:= zip.UnZipTuple;
Println(unzip);
```



Второй метод UnZipTuple действует так же, но для *трёхэлементных* кортежей:

```
function UnZipTuple<T,T1,T2>(Self: sequence of (T,T1,T2)):
                             (sequence of T,sequence of T1,
                              sequence of T2); extensionmethod;
```

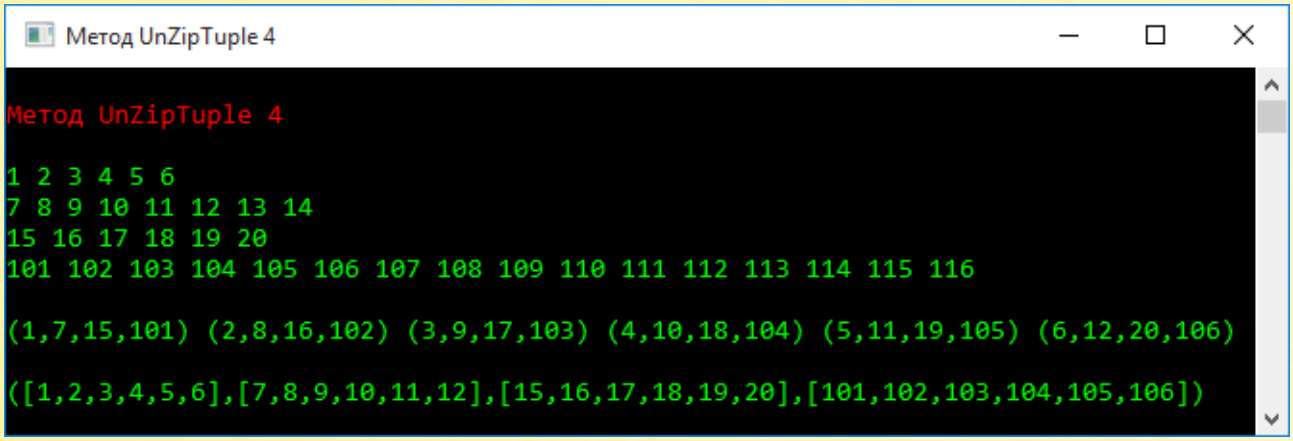
```
var sq1 := Range(1, 6)
           .Println;
var sq2 := Range(7, 14)
           .Println;
var sq3 := Range(15, 20)
           .Println;
Println;
var zip:= sq1.ZipTuple(sq2, sq3)
              .Println;
Println;
var unzip:= zip.UnZipTuple;
Println(unzip);
```



Третий метод UnZipTuple действует так же, но для *четырёхэлементных* кортежей:

```
function UnZipTuple<T,T1,T2,T3>(Self: sequence of (T,T1,T2,T3)):
                                (sequence of T,sequence of T1,
                                 sequence of T2,sequence of T3);
                                 extensionmethod;
```

```
var sq1 := Range(1, 6)
           .Println;
var sq2 := Range(7, 14)
           .Println;
var sq3 := Range(15, 20)
           .Println;
var sq4 := Range(101, 116)
           .Println;

Println;
var zip:= sq1.ZipTuple(sq2, sq3,sq4 )
              .Println;

Println;
var unzip:= zip.UnZipTuple;
Println(unzip);
```



Задания для самостоятельного решения

Из книги «Решение задач на языке *паскаль*»

Египетские коровы
Римские адвокаты
Персидские яблоки
Кахунский папирус
Берлинский папирус
Русские яблоки
Турецкие долгожители
Чешские сливы
Французский покупатель *
Болгарские сливы *
Болгарский парикмахер
Немецкий вопрос
Русские гуси
Насос Эдисона
Китайская арифметика
Задача Этьена Безу
Кому сколько лет?
Ноги и головы
Немецкая копилка
Ошибки
Коровы
Повторяющиеся цифры *
Гимнастический зал
Грузовые машины
Кувшин
Ещё один кувшин

Собственные методы расширения для последовательностей

Встроенных и дополнительных методов расширения очень много, но для решения конкретных задач вам могут потребоваться и такие, которых не в стандартном наборе. Тогда вы можете написать собственные.

Метод *RandomElement*

Метод RandomElement возвращает случайный элемент последовательности:

```
// СЛУЧАЙНЫЙ ЭЛЕМЕНТ
function RandomElement<T>(Self: sequence of T): T; extensionmethod;
begin
   var n := Self.Count();
   var id := PABCSystem.Random(n);
   Result := Self.ElementAt(id);
end;
```

Если метод что-то возвращает, значит, это функция.

Так как это метод расширения, то параметр должен называться Self. Это и есть та последовательность, для которой мы пишем расширение. Она может содержать элементы любого типа T.

Тип возвращаемого значения – это тип элементов последовательности, то есть T.

Заканчивается заголовок функции ключевым словом extensionmethod.

В теле функции переменная Self – это последовательность, для которой мы пишем расширение.

В остальном методы расширения ничем не отличаются от обычных функций.

Метод готов. Пора создавать последовательность.

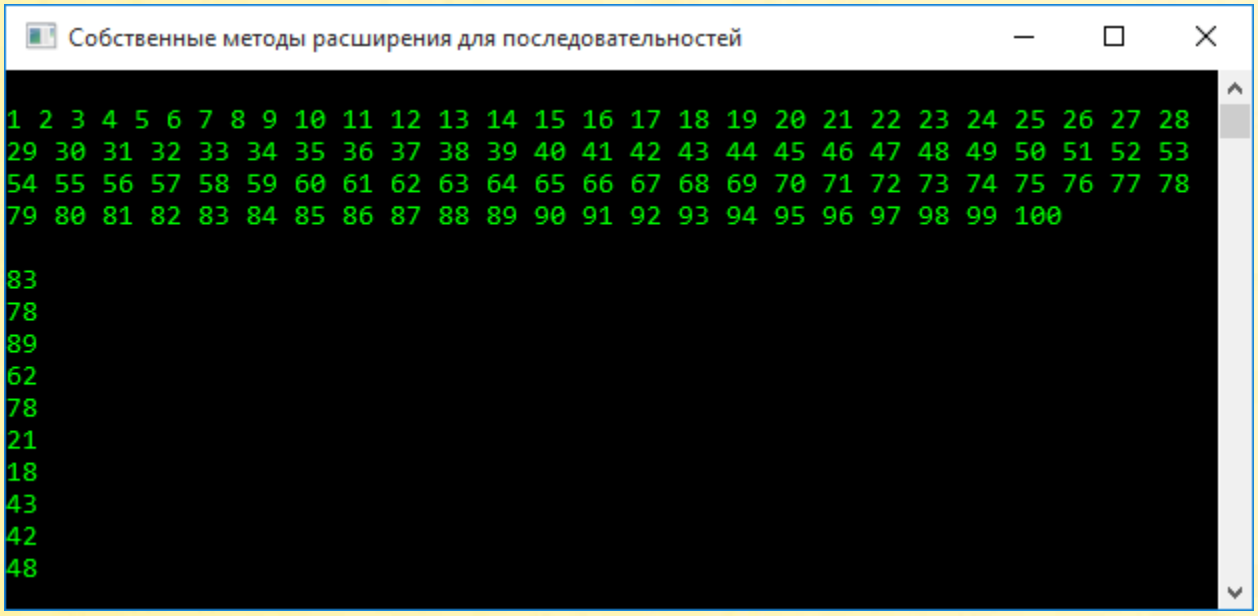
Пусть это будет ряд натуральных чисел:

```
// последовательность:
var sq:= Range(1,100)
        .Println;
```

В цикле for вызываем 10 раз наш метод для этой последовательности:

```
Println;
// случайный элемент:
for var i:= 1 to 10 do
begin
   var re:= sq.RandomElement();
   Println(re);
end;
```

10 случайных элементов последовательности получены:



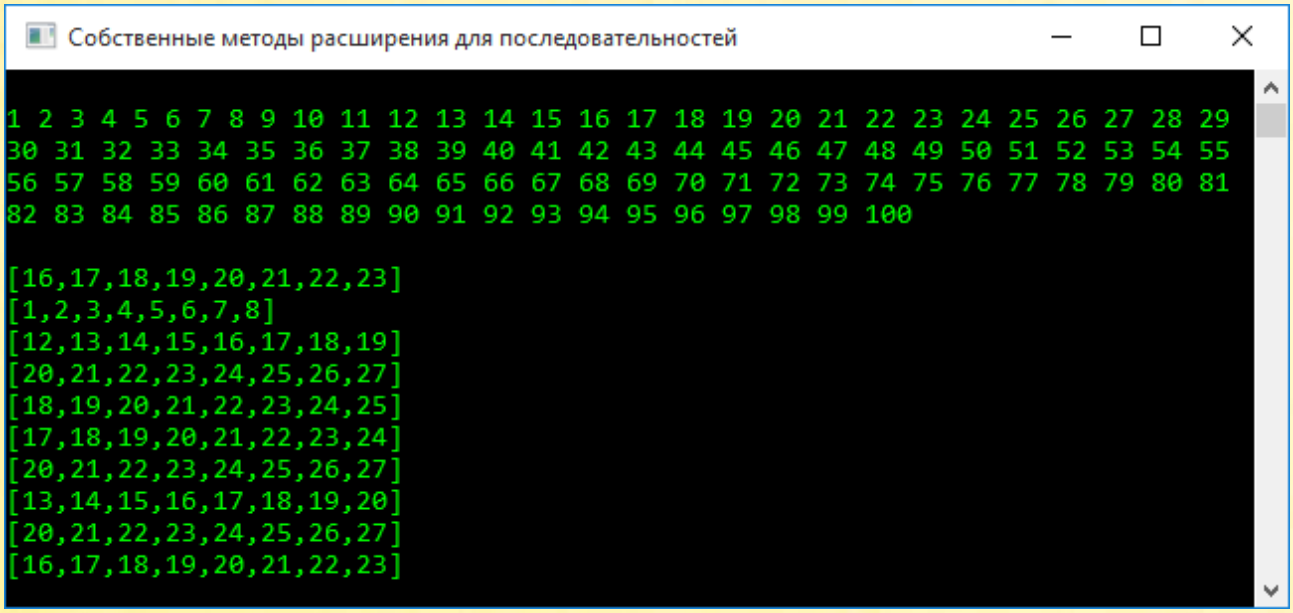
Метод *SubSequence*

Метод SubSequence возвращает часть последовательности длиной *length*, начиная с элемента с индексом *start*:

```
// ВЫРЕЗКА
function SubSequence<T>(Self: sequence of T;
            start, length: integer): sequence of T; extensionmethod;
begin
    Result := Self.Skip(start)
                  .Take(length);
end;
```

Опять применяем новый метод 10 раз:

```
Println;
// подпоследовательность:
for var i:= 1 to 10 do
begin
   var start:= PABCSystem.Random(20);
   var subsq:= sq.SubSequence(start, 8);
   Println(subsq);
end;
```



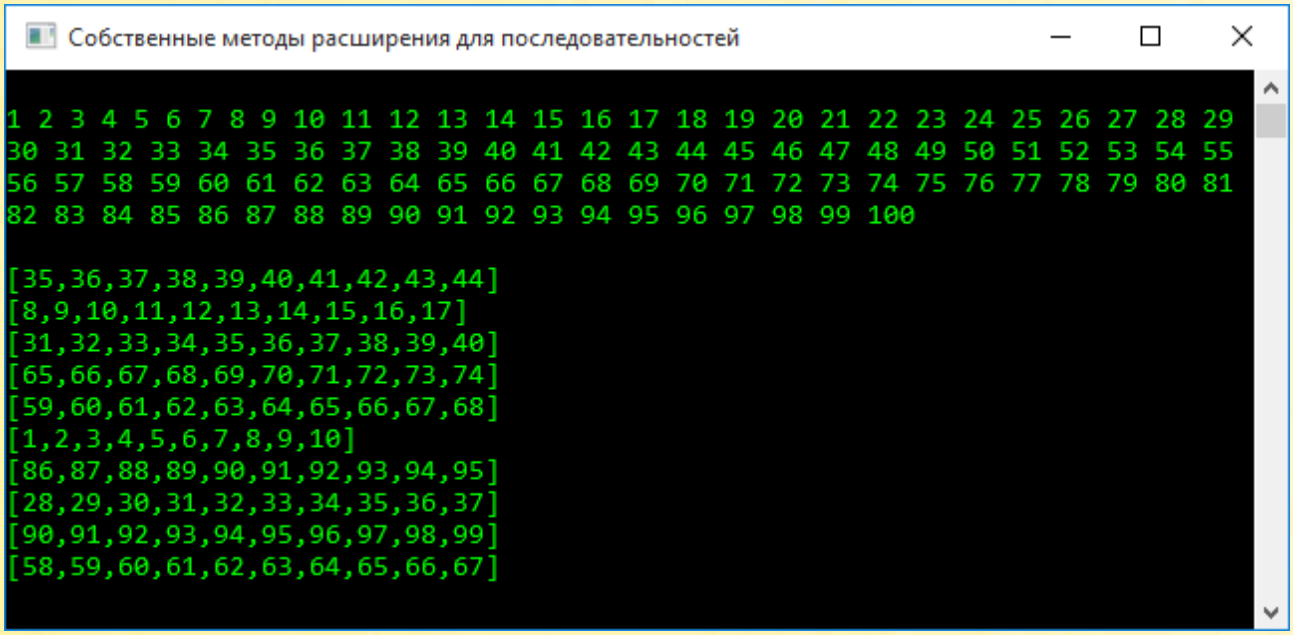
Метод *RandomSubSequence*

Метод RandomSubSequence объединяет два предыдущих метода – он возвращает случайную подпоследовательность заданной длины *length*:

```
// СЛУЧАЙНАЯ ПОДПОСЛЕДОВАТЕЛЬНОСТЬ
function RandomSubSequence<T>(Self: sequence of T;
                length: integer): sequence of T; extensionmethod;
begin
   var start := PABCSystem.Random(Self.Count-length+1);
   Result := Self.Skip(start)
                 .Take(length);
end;
```

Традиционная проверка нового метода в главном блоке:

```
Println;
// случайная подпоследовательность:
for var i:= 1 to 10 do
begin
   var rsubsq:= sq.RandomSubSequence(10);
   Println(rsubsq);
end;
```



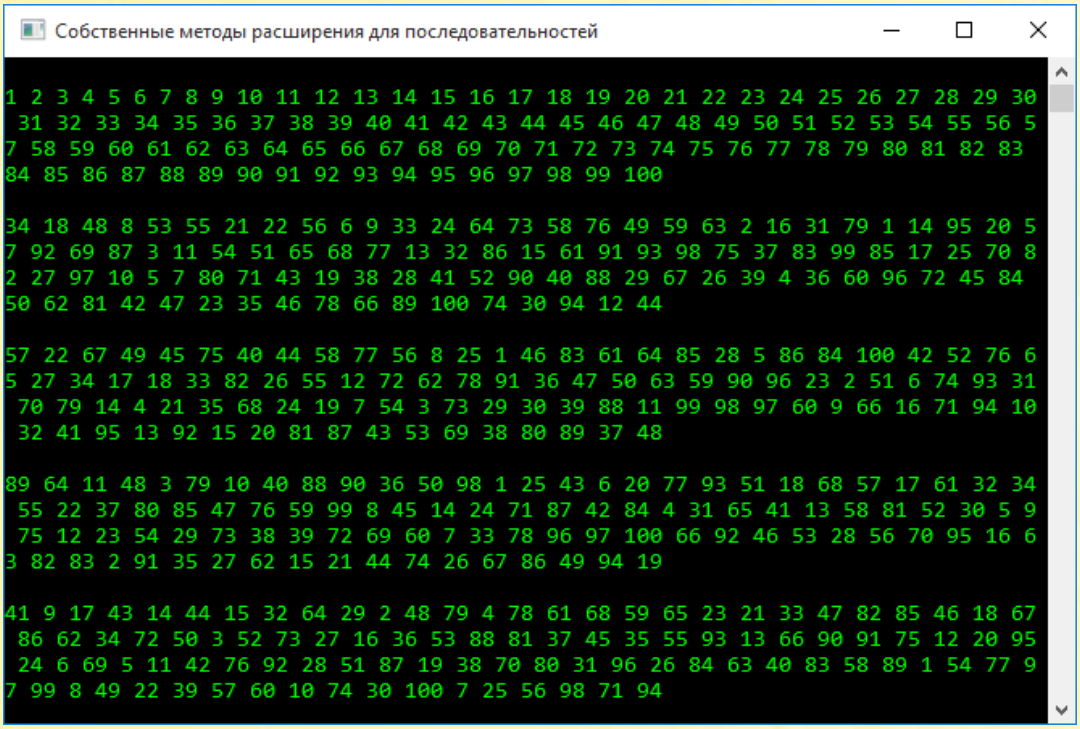
Метод *Shuffle*

Очень часто необходимо перемешать элементы отсортированной последовательности. Наш **метод расширения Shuffle** решает эту задачу:

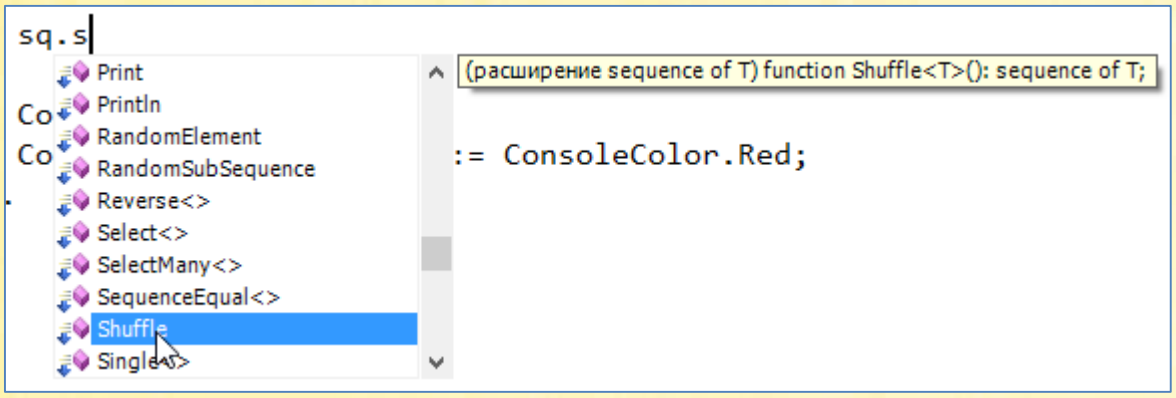
```
// ПЕРЕМЕШИВАЕМ ПОДПОСЛЕДОВАТЕЛЬНОСТЬ
function Shuffle<T>(Self: sequence of T): sequence of T;
                                          extensionmethod;
begin
   var sq:= Self.ToList;
   Result := sq.OrderBy(x -> PABCSystem.Random(sq.Count));
end;
```

10 замесов – и все случайные:

```
Println;
// перемешанная последовательность:
for var i:= 1 to 10 do
begin
   var sqsh:= sq.Shuffle().Println;
   Println;
end;
```



Новые методы расширения появляются в **подсказке**, так пользоваться ими очень удобно:



Аналогично вы можете написать и другие методы расширения для любых коллекций. Например, для массивов и списков.

Методы расширения можно писать непосредственно в файле программы или в отдельном модуле, чтобы пользоваться ими многократно.

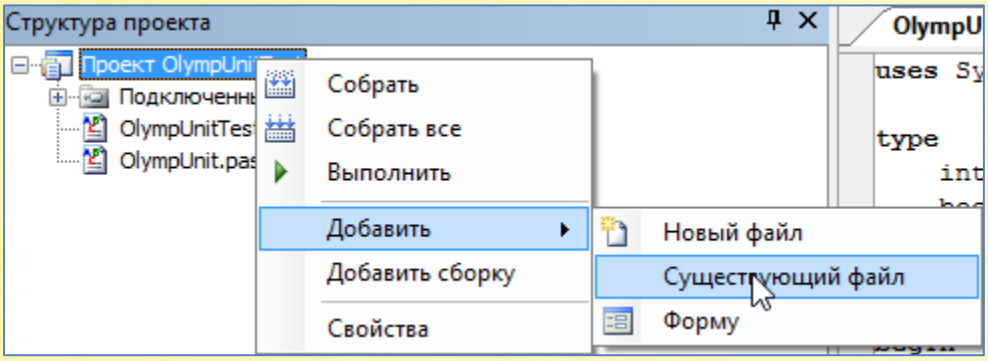
Модуль *OlympUnit*

При решении олимпиадных и многих других задач, очень часто нужны функции для вычисления факториалов, чисел Фибоначчи, для проверки чисел на палиндромность, простоту, и так далее. Вполне разумно поместить их в **отдельный модуль**, чтобы все нужные функции всегда были под рукой.

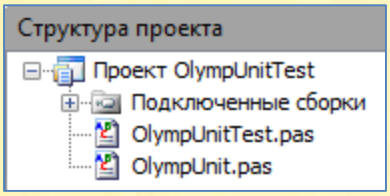
OlympUnit

Для отладки модуля необходима **тестовая программа**.

Создайте новый консольный проект **OlympUnitTest**, а также модуль **OlympUnit**. Сохраните их на диске и добавьте файл модуля к тестовому проекту:



Теперь все файлы удобно разместились в одном проекте:



Метод *ReverseNum*

Обычно в задачах используют числа типа integer. Мы напишем для них метод расширения, который возвращает исходное число, записанное в обратном порядке:

```
// ПЕРЕВОРАЧИВАЕМ ЧИСЛО
function ReverseNum(Self: integer): integer; extensionmethod;
begin
    var rev := 0;
    var num := self;
    while (num > 0) do
    begin
        rev := rev * 10 + num mod 10;
        num := num div 10;
    end;
    Result := rev;
end;
```

Для печати целых чисел нам пригодится метод расширения Println:

```
// ПЕЧАТАЕМ ЧИСЛО ТИПА integer
```

```
function Println(Self: integer): integer; extensionmethod;
begin
   Println(Self);
   Result := Self;
end;
```

Методы простые и понятные, поэтому сразу перейдём к их проверке.

А проверка тоже очень простая:

```
var i:= 1234567890.Println;
i.ReverseNum.Println;
```

Сначала мы печатаем заданное число *i*, затем переворачиваем его и снова печатаем.

Рисунок показывает, что новый метод работает верно:

Иногда в задачах встречаются и очень большие числа, которые имеют тип BigInteger. Для них также имеется метод расширения ReverseNum:

```
function ReverseNum(Self: BigInteger): BigInteger; extensionme-
thod;
begin
    var rev: BigInteger := 0;
    var num := self;
    while (num > 0) do
    begin
        rev := rev * 10 + num mod 10;
        num := num div 10;
    end;
    Result := rev;
end;
```

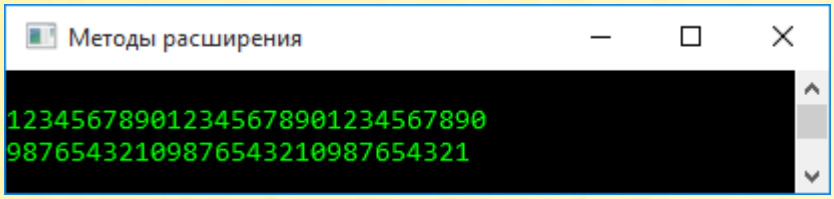
И метод печати:

```
// ПЕЧАТАЕМ ЧИСЛО ТИПА BigInteger
function Println(Self: BigInteger): BigInteger; extensionmethod;
begin
   Println(Self);
   Result := Self;
end;
```

Проверяем работу методов расширения для больших чисел:

```
Println;
var bi:= BigInteger.Parse('12345678901234567890123456789').Println;
bi.ReverseNum.Println;
```

И это испытание наши методы прошли успешно:



Метод *NumDigit*

Чисто олимпиадная задача – подсчитать число цифр в заданном числе (длину) – решается очень просто:

```
// НАХОДИМ ЧИСЛО ЦИФР В ЗАДАННОМ ЧИСЛЕ
function NumDigit(Self: integer): integer; extensionmethod;
begin
   var num := Abs(self);
   var nd := 0;
   while (num > 0) do
   begin
      nd += 1;
      num := num div 10;
   end;
```

```
   Result := Max(1, nd);
end;


function NumDigit(Self: int64): integer; extensionmethod;
begin
   var num := Abs(self);
   var nd := 0;
   while (num > 0) do
   begin
      nd += 1;
      num := num div 10;
   end;
   Result := Max(1, nd);
end;
```

Берём 10- и 1-значное числа и применяем к ним наш метод расширения:

```
Println;
var nd:= 1234567890.Println.NumDigit.Println;
0.Println.NumDigit.Println;
```

И этот метод считать умеет:

Метод *IsQuadrat*

И опять олимпиадная задача: проверить, **является ли заданное число полным квадратом**.

Извлекаем из числа квадратный корень, оставляем целую часть и возводим её в квадрат. Если получилось исходное число, значит, оно и есть полный квадрат:

```
function IsQuadrat(Self: integer): boolean; extensionmethod;
begin
   var r := Trunc(Sqrt(self));
   Result := r * r = self;
end;
```

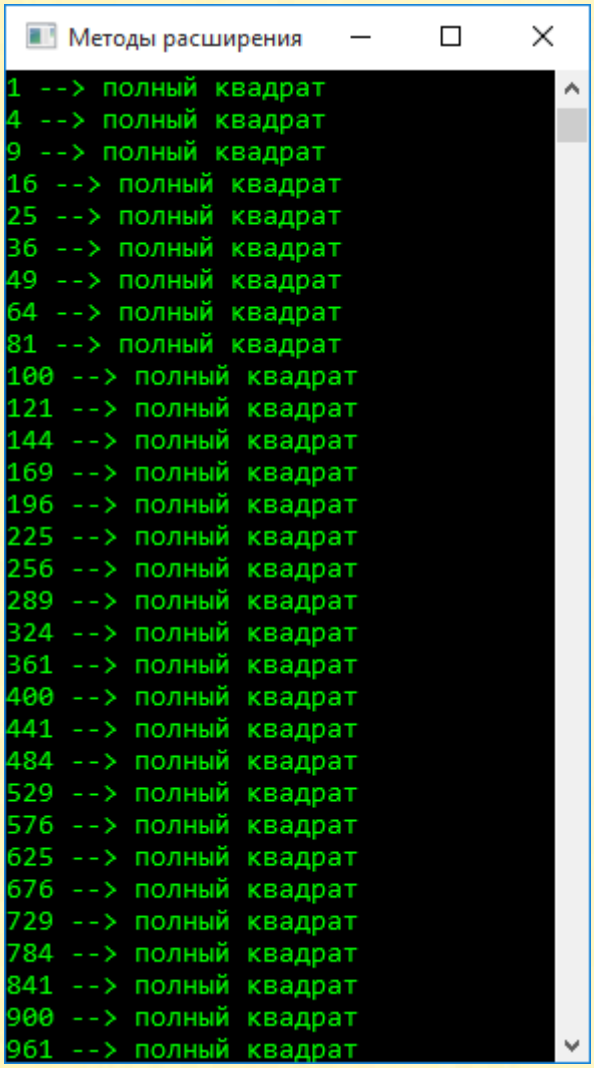
Проверяем метод на первой тысяче чисел. Классическим способом:

```
Println;
for var a := 1 to 1000 do
    if a.IsQuadrat then
       Println(a, ' --> полный квад-
рат');
```

Или функциональным:

```
Println;
Range(1,1000)
.Where(a -> a.IsQuadrat)
.Select(a -> NewLine + a + ' -->
полный квадрат')
.Println;
```

Результаты в обоих случаях мы получили одинаковые и правильные:



Метод *IsCube*

Подобная задача для кубов.

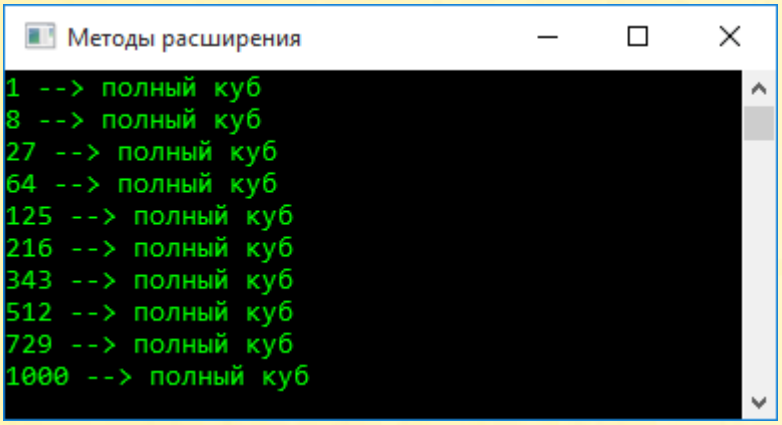
В этом случае извлечь кубический корень сложнее, но не гораздо:

```
function IsCube(Self: integer): boolean; extensionmethod;
begin
   var r := Trunc(Round(Power(self, 1.0 / 3.0)));
   Result := r * r * r = self;
end;
```

Снова проверяем первую тысячу чисел:

```
Println;
Range(1,1000)
.Where(i -> i.IsCube)
.Select(a -> NewLine + a + ' --> полный куб')
.Println;
```

И опять наш метод на высоте:



Метод *IsPalindrome*

И уже совсем олимпиадная задача – проверить, является ли заданное число палиндромом.

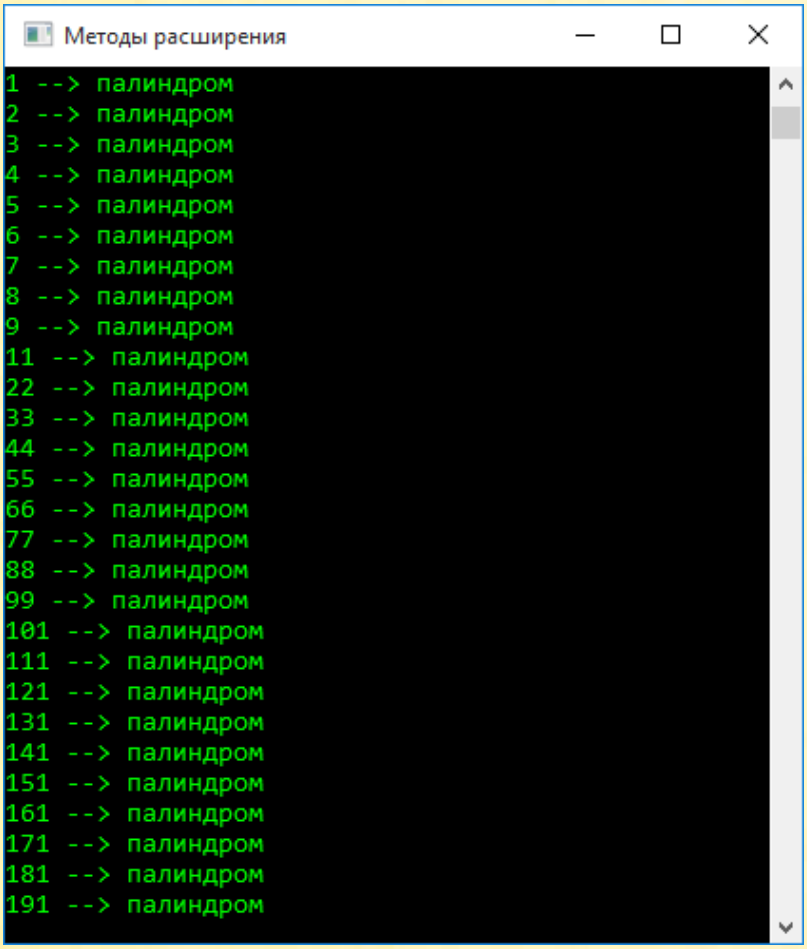
Эта задача – самая сложная:

```
function IsPalindrome(Self: integer): boolean; extensionmethod;
begin
   // избавляемся от чисел, заканчивающихся на нуль:
```

```
    if ((self <> 0) and (self mod 10 = 0)) then
    begin
       Result := false;
       exit;
    end;

    var rev:= 0;
    var num := self;
    while (num >= rev) do
    begin
       rev := 10 * rev + num mod 10;
       if (num = rev) then
       begin
          Result := true;
          exit;
       end;

       num := num div 10;
       if (num = rev) then
       begin
          Result := true;
          exit;
       end;
    end;
    Result := false;
end;
```

Проверяем две сотни чисел:

```
Println;
Range(1,199)
.Where(a -> a.IsPalind-
rome)
.Select(a -> NewLine + a
+ ' --> палиндром')
.Println;
```



И здесь всё сходится.

При запуске тестовой программы в папке создаётся файл **OlympUnit.pcu**. Это скомпилированный модуль, который нужно скопировать в папку с программой, которая его использует. А в начале самой программы добавьте модуль в раздел **uses**:

```
uses OlympUnit;
```

Здесь мы рассмотрели только те методы расширения, которые использовали при решении задач в этой книге. Но вы можете добавить в олимпиадный модуль и собственные методы. Тогда решение даже сложных задач станет более интересным и простым.